C.R.J.!
This is the last CS 5 lecture you’ll ever "need"!

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!

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!
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!
Recursion example: \( \text{numis}(s) \)

- Total # of i's in \('i<3\text{five}'\)
- # of i's in \('i'\) + # of i's in \('<3\text{five}'\)
Recursion example: \( \textit{numis}(s) \)

- total \# of i's in 'alien'
- is
- \# of i's in 'a' \( S(0) \)
- +
- \# of i's in 'lien' \( S(1) \)
Recursion example: \( \text{numis}(s) \)

- total # of i's in 'aliiien'
- # of i's in 'a'
- # of i's in 'liiiien'

is

\[ \text{numis}(s) = \text{numis}(a) + \text{numis}(liiiien) \]
Recursion example: \( \text{numis}(s) \)

Analysis...

Total # of i's in 'aliienen' is

# of i's in 'a' + # of i's in 'liienen'

... via self-similarity!
This is the last CS 5 lecture you’ll ever "need"!

* 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!*
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

Illustration: Marien Banijes

"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.

THE PETABYTE AGE:
Sensors everywhere. Infinite 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.
Computation's Dual Identity

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

accessed through \textit{functions}…

Variables ~ boxes

It's no coincidence this starts with \texttt{fun}!
Functioning across disciplines

**procedure**

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

**CS's googolizer**

defined by *what it does*

+ what follows *behaviorally*

**structure**

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

**Math's googolizer**

defined by *what it is*

+ what follows *logically*
Giving names to data \textit{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 \textit{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'

4

'work'

'home'

Follow the data...
I'm happy about this, too!

OK: we humans work better with named variables.

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

Use variables!

Aargh!
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'
print(" petscar ~", flipside('carpets'))
print("    cs5! ~", flipside('5!cs'))

We provide tests (for now...)
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]

This program uses variables constantly!
Naming things!

name

signature line

docstring

code block

return statement

in-line comments – these are optional in CS 5
return vs. print

def dbl(x):
    """ dbls x? """
    return 2*x

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

ans = dbl(20)  # ans = dblPR(20)

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

dblPR(x):
    """ dble x? """
    print(2*x)

ans = dbl(20) + 2
ans = dblPR(20) + 2

return > print

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

print changes pixels on the screen... print

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

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

changes the pixels (little *lightbulbs*) on your screen
return > print

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 demo(15) here?

What is f(2) here?

I might have a guess at both of these...
```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…**

"the 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 = ?????

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)
local variables: x = 15
               y = 5
               z = ????

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

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

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

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)  # stack frame
local variables:
    x = 15
    y = 5
    z = ????

result = 22
returns 22

call: g(5)  # stack frame
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

15

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

How functions work...

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)  
stack frame: 
local variables:  
  x = 15
  y = 5
  z = 22 
return  42

they stack.
How functions work...

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

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

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

they stack.
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

How functions work...

stack frame

local variables:

f(0) + 10

stack frame

local variables:

x = 2
need f(1)

call: f(2)

local variables:

x = 1
need f(0)

call: f(1)
How functions work...

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

- **Call:** \( f(2) \)
  - **Local Variables:** \( x = 2 \)
  - **Stack Frame:**
    - **Call:** \( f(1) \)
      - **Local Variables:** \( x = 1 \)
      - **Stack Frame:**
        - **Call:** \( f(0) \)
          - **Local Variables:** \( x = 0 \)
          - **Returns:** 12
```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

How functions work...

"the stack"

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

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

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

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

x = 2
result = f(1) = 22

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

call: f(2)
local variables:
    x = 2
    f(1) = 22
result = 42

which then gets returned...
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

result = f(2)
result = 42

the result then gets returned...
How functions work...

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

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

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

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

Pass these eastward!

... each one is a self-contained computational unit!
sequential
iteration
self-similar
recursion

problem-solving paradigms
Thinking *sequentially*

**factorial**

\[ 5! = 120 \]

\[ \text{Cs} \quad \text{fac}(5) = 5\times4\times3\times2\times1 \]

\[ \text{fac}(N) = N\times(N-1)\times \ldots \times 3\times2\times1 \]
Thinking *sequentially*

**Math**  
5! = 120

**CS**  
fac(5) = 5*4*3*2*1

fac(N) = N*(N-1)*...*3*2*1
Thinking *recursively*

**factorial**

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

\[ \text{fac}(5) = 5! = 120 \]

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

\[ \text{fac}(N) = N! \]

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

**Recursion ~ self-similarity**

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

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

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

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

Can we express \( \text{fac} \) w/ a smaller version of itself?

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)
```
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?](https://stackoverflow.com/questions/...)) 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!*

```python
def fac(N):
    return fac(N)
```
Recursion: the dizzying dangers of having no base case!

This "works"... but doesn't work!
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
def facBad(N):
    return N * facBad(N-1)

calls to facBad will "never" stop: there's no BASE CASE

Make sure you have a base case

legal != recommended

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

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

Actual

Acting recursively

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

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
Behind the curtain: how recursion works...

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

Diagram:
- `fac(5)` leads to `5 * fac(4)`
- `4 * fac(3)`
- `3 * fac(2)`
- `2 * fac(1)`
- `1.0`

Stack frames:
- Stack frame with `N = 5`
- Stack frame with `N = 4`
- Stack frame with `N = 3`
- Stack frame with `N = 2`
- Stack frame with `N = 1`
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 * fac(4)

4 * fac(3)

3 * fac(2)

2 * 1.0
```

stack frame with N = 5

stack frame with N = 4

stack frame with N = 3

stack frame with N = 2
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
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)`
- `4 * 6.0`

*stack frame* with `N = 5`

*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*...

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

`fac(5)`

120.0

*complete!*

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

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

Nature loves recursion!

... *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 $5!$ is $5*4*3*2*1$.

Base case:
fac(0) should return 1.
def fac(x):
    """ factorial! Recursively! """
    if x == 0:
        return 1
    else:
        return x*fac(x-1)
The value of $1+1+1+1+1$ is plus one ($5$).

Base case: $\text{plusone}(0)$ should return $0$. 

$\text{plusone}(n)$ adds 1 a total of $n$ times.
plusone(5) is 

value of \(1+1+1+1+1\)

value of 1 + value of \(1+1+1+1\) 

plusone(4)

Base case: plusone(0) should return ___
def plusone(n):
    """
    returns n by adding 1's!
    """
    if n == 0:
        return
    else:
        return 1 + plusone(n-1)
def plusone(n):
    """
    returns n by adding 1's!
    """
    if n == 0:
        return 0
    else:
        return 1 + plusone(n-1)
value of $2*2*2*2*2$ is $\text{pow}(2,5)$

value of $2$ * $\text{pow}(2,4)$

Base case: $\text{pow}(2,0)$ should return __?
valor de $2^5$ es $2 \times 2 \times 2 \times 2 \times 2$

Base caso:
pow(2,0) debería retornar ___?
def pow(b, p):
    
    """b**p, defined recursively!"
    """

    if p == 0:
        return 1

    else:
        return b * pow(2, p)**(2**2**2)
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...?
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!

Pomona Sends Survey To Students To Find Out Why They Don’t Take Surveys

Ima Firstyear

Declining survey response rates at Pomona College prompted the administration to send students a new survey this week, which will assess students’ previous survey experiences and their survey preferences in hopes of explaining—and reversing—the decline.

“We know Pomona students have strong opinions about their education and their campus,” said Vice President and Dean of Students Miriam Feldblum. “But what we find is that when we offer students a chance to express those opinions via a general survey, we don’t get as many responses as we expect. We want to know why, and that’s why we’re sending out this survey.”

Students will be asked to self-identify at the start of the survey as a ‘frequent responder,’ ‘occasional responder’ or ‘forgot the password to my Pomona webmail account three months ago.’ According to Feldblum, these categories will help the administration create new strategies to engage more of the student population in responding to surveys.

The survey also addresses questions of methodology, incentive and access. It asks students to rank their preferences of survey provider, such as SurveyMonkey, Qualtrics and Google Forms, and to name their ideal survey prizes. It also asks students whether they would be more inclined to take school surveys via email, an iPhone app or voting machines in the dining halls complete with ‘I Surveyed!’ stickers.

Erika Bennett PO ’17 said she found some of the questions confusing.

“I had to pick my favorite assessment scale,” she said. “I had to rank ‘Scale of one to five,’ ‘Strongly Disagree to Strongly Agree’ and ‘Sad Face to Happy Face’ from least to most intuitive. But I’m not sure I did it correctly.’ Bennett added that she did appreciate the chance to critique previous surveys.

“Just last month I took a survey with no progress bar at the bottom of each page,” she said. “I felt lost and confused. I’m glad there’s a real See SURVEY page 2
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:

\[ s = 'aliiiien' \]

\[ s[0] \]

\[ s[1:] \]

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

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

\[ s = 'aliien' \]

\[ 'a' \quad 'liiien' \]

\[ s[0] \quad s[1:] \]

*handle the first*  
*recurse the rest*

**Human!**  
**Machine!**
Design patterns...

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

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

L[0] \(=\) \(3\)

L[1:] \(=\) \([1,4,1,5,9]\)

\(L[0]\) handle the **first**

\(L[1:]\) recurse the **rest**

*Human!* Machine!
Design patterns...

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

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

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

- **Handle base cases, with** \( \textbf{if} \) ...
numis('xlii')

# of i's in 'xlii'

numis('x') + numis('lii')

Base case:
numis('') should return ___?
def numis(s):
    """ # of i's in s
    ""
    if s == '':
        return
    elif s[0] == 'i':
        return 1 + numis(s[1:])
    else:
        return 0 + numis(s[1:])
```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 ___?
```python
def len(s):
    ""
    returns the length of s
    ""
    if s == '':
        return
    else:
        return
```

Extra! Can we also handle **LISTS**...?
def \texttt{len}(s):

\begin{verbatim}
    """
    returns the length of s
    """

    if s == ' ' or s == []:
        return 0

    else:
        return 1 + \texttt{len}(s[1:])
\end{verbatim}
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...
\[ \text{vwl('eerie')} \]

\# of vowels in 'eerie'

is

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

Base case:
\text{vwl('') should return ___ ?}
```python
def vwl(s):
    """ # of vowels in s
    """
    if s == '':
        return __________

    elif __________:
        return __________

    else:
        return __________
```

Python is... in

```python
>>> '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
```
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]

max(L)

L's biggest element

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

The max of the REST of L
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 = __________
    if abs(L[0]) < abs(Z):
        return __________
    else:
        return __________

The zeroest of the REST of L
```python
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's really being added here?

What seven-letter s maximizes \texttt{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:])

What's really being added here?
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

Good luck with Homework #1

tutors @ LAC + 4C's    Th/F/Sa/Su/Mon.
The function `pow(b, p)` returns $b^p$, but using only multiplication times $b$. To implement it, you can use a recursive approach:

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

Base cases are:
- $b^0 = 1$
- Anything to the zero power is 1.0

Self-similarities:
- $b^p = b \times b^{p-1}$

Examples:
- $\text{pow}(2,4) \sim 2 \times 2 \times 2 \times 2 = 16$
- $\text{pow}(2,3) \sim 2 \times 2 \times 2 = 8$
- $\text{pow}(2,2) \sim 2 \times 2 = 4$
- $\text{pow}(2,1) = 2$
- $\text{pow}(2,0) \sim 1$
The function `vwl(s)` returns the number of vowels in the string `s`, `s` may or may not be empty.

**Base Cases**
- `vwl('') ~ 0` (empty string)
- `vwl('a') ~ 1` (single vowel)
- `vwl('ca') ~ 1` (two vowels)
- `vwl('zy') ~ 1` (two vowels)
- `vwl('azy') ~ 2` (three vowels)
- `vwl('crazy') ~ 2` (three vowels)

**Self-similarities**
- `vwl('azy') = vwl('azy')` (two vowels)
- `vwl('zy') = vwl('zy')` (one vowel)
- `vwl('zy') = 1` (one vowel)
- `vwl('y') = 1` (one vowel)
- `vwl('y') = 1` (one vowel)

**Defining `vwl(s)`**
```
def vwl(s):
    if s == ''
        return 0
    elif s[0] in 'aeiou'
        return 1 + vwl(s[1:])
    else:
        return vwl(s[1:])
```

**Extra Question**
What 7-letter English word maximizes `vwl(s)`?
**Answer:** \( \texttt{pow} \)

\[
\text{pow}(2,4) \sim 2 \times 2 \times 2 \times 2 = 16
\]

\[
\text{pow}(2,3) \sim 2 \times 2 \times 2 = 8
\]

\[
\text{pow}(2,2) \sim 2 \times 2
\]

\[
\text{pow}(2,1) \sim 2
\]

\[
\text{pow}(2,0) \sim 1
\]

anything to the zero power is 1.0

---

**intuit**

**pow**

\[
\text{pow}(b,p) = \text{pow}(b,p-1) \times b
\]

**identify**

**self-similarities & base cases**

**self-similarities**

**base cases**

\[
\text{pow}(b,0) = 1.0
\]

**implement**

**def** \( \texttt{pow}(b,p) \):

\[
\text{if } p == 0:
\]

\[
\text{\quad return } 1.0
\]

\[
\text{else:}
\]

\[
\text{\quad return } \text{pow}(b,p-1) \times b
\]

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