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!
Next Monday, I'm out of Claremont...

Recursion

As close as CS gets to magic

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

Hw #1 due this Monday, May 29, at 11:59 pm

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

*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.'
Next Monday, I'm out of Claremont...

So, the next homework is due Monday night – and next Monday evening will be a chance to work on it, with tutoring support!

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

xFE 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"!

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...

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

As close as CS gets to magic

Jack Ma's Picobot "magic"

Next Monday, I'm out of Claremont...

So, the next homework is due Monday night -- and next Monday night will START.

And, next Wed. will START.

Monday night -- and next there will be a bit of turtle graphics! So, today background on that...
Picobot, *Aargh!* 

Hello Zach,

We would like to thank you for this great project idea. The project is over and it was a success. We got more than 100 different Java implementations of the simulator from student groups, and a couple of groups did have a lot of fun in solving the 7 maps.

Interestingly, the best solution for each map (in terms of number of rules) was found by different groups. Here is what they found:

- map 1: 6 rules
- map 2: 8 rules
- map 3: 10 rules
- map 4: 25 rules
- map 5: 34 rules (two different solutions)
- map 6: 23 rules
- map 7: 41 rules (12 groups solved this last map with up to 958 rules!)
- universal solution: 58 rules

Also, one group designed a solution generator, i.e. an algorithm which takes a map as input and outputs a set of rules that works for any initial position. It seems to work, but we have to test it comprehensively.
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!

Hw #1 due this Monday, 1/30, at 11:59 pm
Python is... **in** MAGIC!

```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
```
if you attended lab and submit pr1+pr2:
you get full credit for hw1pr1 and hw1pr2

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

either way: submit pr1 + pr2
complete and submit hw1pr3

Extra Credit: Pig Latin / CodingBat

DNA transcription
Piazza!

Questions?

+ add'l tutoring times?

needed?
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: Marian 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.

Petabytes? This article is old-school!
Computation's Dual Identity

-but what does the stuff on this side look like?
Functions!

It's no coincidence this starts with *fun*!

accessed through *functions*…
Functioning across disciplines

**procedure**

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

**structure**

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

**CS's googolizer**

**Math's googolizer**

defined by **what it does**

+ what follows **behaviorally**

defined by **what it is**

+ what follows **logically**
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!
Use variables!

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

I'm happy about this, too!

```python
def flipside(s):
    return s[len(s) // 2:] + s[:len(s) // 2]
```

Why would computers "prefer" the top version, too?
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!
#
print("workhome ~", flipside('homework'))
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 number 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]
return vs. print

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

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

In[1]: ans = dbl(20)

In[1]: ans = dblPR(20)

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

>>> ans = dbl(20)+2

>>> ans = dblPR(20)+2

print changes pixels on the screen...

return yields the function call's value ...
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

Quiz

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

What is \texttt{f(2)} here?
```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"

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.
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 = ?????

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

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...

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

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

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

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...**

- **Call:** `demo(15)`
- **Stack Frame:**
  - **Local Variables:**
    - `x = 15`
    - `y = 5`
    - `z = 22`
  - `return 42`

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...

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

they stack.
How functions work...

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

2

what's f(2)?
How functions work...

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

1

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

stack frame

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

How functions work...

0

<table>
<thead>
<tr>
<th>call: f(2)</th>
<th>stack frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>local variables:</td>
<td></td>
</tr>
<tr>
<td>x = 2</td>
<td></td>
</tr>
<tr>
<td>need f(1)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>call: f(1)</th>
<th>stack frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>local variables:</td>
<td></td>
</tr>
<tr>
<td>x = 1</td>
<td></td>
</tr>
<tr>
<td>need f(0)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>call: f(0)</th>
<th>stack frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>local variables:</td>
<td></td>
</tr>
<tr>
<td>x = 0</td>
<td></td>
</tr>
<tr>
<td>returns 12</td>
<td></td>
</tr>
</tbody>
</table>
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 \( f(x-1) + 10 \times x \)

How do we compute the result?
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 = 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

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

2

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

which then gets returned...
How functions work...

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

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

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

functions stack.

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

Functions are like cells, i.e., software's building blocks... ...each one, a self-contained computational unit!

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

**factorial**

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

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

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

\[\text{factorial} \]

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

\[
\text{cs} \quad \text{fac}(5) = 5*4*3*
\]

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

*June + beyond...*
Thinking *recursively*

factorial

$$5! = 120$$

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

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

**can we express \(\text{fac}\) w/ 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} with a smaller version of itself?
output value of $5\times4\times3\times2\times1$

**Recursion, the process**

starting from 5, decreasing to 1

integers from 5 down to 1, multiplied together

overall goal

$\text{fac}(5)$

$\text{fac}(5)$
Recursion, *the process*

- Output value of $5 \times 4 \times 3 \times 2 \times 1$
- Value of $5$
- One, small piece
- Combined with $4 \times 3 \times 2 \times 1$
- Self-similar "rest"
- Overall goal
- $\text{fac}(5)$
Recursion, *the process*
Recursion's *conceptual* challenge?

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

*Nature loves recursion!*
Recursion, *the code*!
def fac(N):
    return N * fac(N-1)

I wonder how this code will STACK up!?

def facBad(N):
    print("N is", N)
    return N * facBad(N-1)
\textit{legal} \neq \textit{recommended}

def \texttt{facBad}(N):  
    return N * \texttt{facBad}(N-1)

calls to \texttt{facBad} will never stop: there's no \textbf{BASE CASE}

\textbf{Make sure you have a base case}  
a.k.a. "\textit{escape hatch}"
Recursion... the dizzying dangers of having no base case!

This "works"—but doesn't work!
Recursion
the dizzying dangers of having no base case!

This "works" ~ but doesn't work!

```python
def fac(N):
    return fac(N)
```
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
base case: output for \( \text{fac}(0) \) is 1

output value of
\[ N \times (N-1) \times \ldots \times 2 \times 1 \]

is

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

value of
\[ N \]

is

one, small piece

combined with

output value of
\[ (N-1) \times \ldots \times 2 \times 1 \]

fac(N-1)

self-similar "rest"

overall goal

fac(N)

smallest possible input

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

Thinking recursively...
Aside!

**Acting recursively**

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

**Conceptual**

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

**Actual**

this recursion happens first!

hooray for variables!
Behind the curtain: *how recursion works*...

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

- $\text{fac}(5)$
- $5 \times \text{fac}(4)$
- $4 \times \text{fac}(3)$
- $3 \times \text{fac}(2)$
- $2 \times \text{fac}(1)$
- $1.0$
Behind the curtain: *how recursion works*...

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

**stack frame with N = 1**

**stack frame with N = 2**

**stack frame with N = 3**

**stack frame with N = 4**

**stack frame with N = 5**
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...

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

- **fac(5)**
  - **5 \times fac(4)**
    - **4 \times fac(3)**
      - **3 \times 2.0**

Stack frames:
- 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) → stack frame with N = 5

5 * fac(4) → 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)
```

Stack frame with $N = 5$

$$5 \times 24.0$$
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!**
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!

Recursion, *the design process*
Recursive Design: the 3 i's

identify
self-similarities & base cases

intuition

implement
Recursive Design: the 3 i's

- Examples!
- Intuition
- Identify self-similarities & base cases
- What's the self-similarity?
- Python...
- Implement
The overall goal is to compute the factorial of 5, denoted as $\text{fac}(5)$.

Starting from 5 and decreasing to 1, the integers are multiplied together:

$5 \times 4 \times 3 \times 2 \times 1$
output value of $5 \times 4 \times 3 \times 2 \times 1$

is

value of 5

combined with

output value of $4 \times 3 \times 2 \times 1$

Examples!

What's the self-similarity?

overall goal

fac(5)

fac(4)
The overall goal is $\text{fac}(N)$.

The output value of $\text{fac}(N)$ is $N \cdot (N-1) \cdot \ldots \cdot 2 \cdot 1$.

The value of $N$ is one, small piece

$\ast$ combined with self-similar "rest"

The output value of $(N-1) \cdot \ldots \cdot 2 \cdot 1$ is $\text{fac}(N-1)$.

What's the self-similarity?

What's the base case?

Base case: The output for $\text{fac}(0)$ is 1, which is the smallest possible input and its output.
def fac(N):
    if N == 0:
        return 1
    else:
        return N*fac(N-1)
Recursive Design: the 3 i's

**Identify**

fac(N) is self-similarities & base cases

**Base Cases**

- fac(0) is 1
- fac(1) is 1
- fac(2) is 2 * 1
- fac(3) is 3 * 2 * 1
- fac(4) is 4 * 3 * 2 * 1
- fac(5) is 5 * 4 * 3 * 2 * 1

**Intuit**

- fac(5) is 5 * 4 * 3 * 2 * 1
- fac(4) is 4 * 3 * 2 * 1
- fac(3) is 3 * 2 * 1
- fac(2) is 2 * 1
- fac(1) is 1
- fac(0) is 1

**Implement**

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

Aye, aye, eye!
Recursive Design: the 3 i's

- Fac(5) is $5*4*3*2*1$
- Fac(4) is $4*3*2*1$
- Fac(3) is $3*2*1$
- Fac(2) is $2*1$
- Fac(1) is 1
- Fac(0) is 1

**Identify**

- Fac(N) is $N*fac(N-1)$

**Self-Similarities & Base Cases**

- The order of base cases and recursive cases usually doesn't matter.

**Recursive Design**

- Def fac(N):
  - If N > 0:
    - Return N * fac(N-1)
  - Else:
    - Return 1.0

**Implement**

- Rec.
- Base case
Recursive Design: the 3 i's

fac(5) is $5 \times 4 \times 3 \times 2 \times 1$

fac(4) is $4 \times 3 \times 2 \times 1$

fac(3) = $3 \times 2 \times 1$

fac(2) = $2 \times 1$

fac(1) = 1

fac(0) is 1

Let's try two more examples together – and then try 2-3 on your own...

def fac(N):
    if N > 0:
        return N * fac(N-1)
    else:
        return 1.0
Recursive Design: the 3 i's

recursive

def plusone(N):
    if ______:
        return ___
    else:
        return _______________

plusone(4) is 1+1+1+1
plusone(3) is 1+1+1
plusone(2) is 1+1
plusone(1) is 1
plusone(0) is 0

recursion?
base case(s)
intuit

identify
self-similarities
base cases

implement
self-similarities & base cases

rec.

base case
rec.
Output value of $1+1+1+1$ plusone(4) overall goal

$1+1+1+1$ added together, 4 times

plusone(4) 4 times
output value of $1+1+1+1$ is

value of 1

combined with

output value of $1+1+1$

equal to

plusone(4)

plusone(3)

example with concrete values
output value of $1+1+\ldots+1$ is $\text{plusone}(N)$

value of $1$ is one, small piece

combined with $1+\ldots+1$ is self-similar "rest"

base case: output of $\text{plusone}(0)$ is $0$

overall goal

$\text{plusone}(N)$

smallest possible input its output
Recursive Design: the 3 i's

plusone(4) is 1+1+1+1
plusone(3) is 1+1+1
plusone(2) is 1+1
plusone(1) is 1
plusone(0) is 0

implementation
self-similarities & base cases
recursive

intuit
base case(s)

recursion?

plusone(N)
def plusone(N):
    if ______:
        return ___
    else:
        return ______________
Recursive Design: the 3 i's

plusone(4) is 1+1+1+1
plusone(3) is 1+1+1
plusone(2) is 1+1
plusone(1) is 1
plusone(0) is 0

identify

plusone(N) is 1+plusone(N-1)

rec.

self-similarities

base cases

plusone(0) is 0

intuit

def plusone(N):
    if N == 0:
        return 0
    else:
        return 1 + plusone(N-1)

implement

base case
rec.

base cases
**Recursive Design:**

The function `leng(L)` calculates the length of a list `L`. We can naturally break down the problem into smaller subproblems, identifying and solving the base cases and implementing the recursive cases.

### Intuit

- `leng([5,42,7])` is 3
- `leng([42,7])` is 2
- `leng([7])` is 1
- `leng([])` is 0

### Identify

- **Base cases:**
  - `leng([])`
  - `leng([item])`

- **Self-similarities & base cases:**

### Implement

```python
def leng(L):
    if ______________:
        return ___
    else:
        return ____________
```

```python
def leng(L):
    if len(L) == 0:
        return 0
    else:
        return 1 + leng(L[1:])
```
**Recursive Design:**

1. **def** \( \text{len}(L) \):
   
   - **if** \( L == [] \) or \( L == '' \):
     - **return** 0
   
   - **else**:
     - **return** 1 + \( \text{len}(L[1:]) \)

2. Identify self-similarities & base cases

3. Implement base case(s)

4. Intuit: 
   - \( \text{len}( [5,42,7] ) \) is 3
   - \( \text{len}( [42,7] ) \) is 2
   - \( \text{len}( [7] ) \) is 1
   - \( \text{len}( [ ] ) \) is 0
   - \( \text{len}( '' ) \) is 0
   - \( \text{len}( \text{'and me?'}) \)
Design patterns...

- Handle base cases, with `if` ...
- Do one piece of work: `L[0] or s[0]`
- Recurse with the rest: `L[1:] or s[1:]`
- Combine + make sure the types match!

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

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!**

As a hat, I'm recursive, too!

https://www.youtube.com/watch?v=8PhiSSnaUKk @ 1:11
https://www.youtube.com/watch?v=ybX9nVLtNi4   @ 0:08
romanesco broccoli
Yes... and no. Are these rules for real?
There still has to be a base case...
or else!
def pow(b, p):
    if __________:
        return ________
    else:
        return _______________

Extra! See if you can also handle negative powers...
def vwl(s):
    if s:
        return vwl(s)
    elif s == '':
        return 0
    else:
        return 1

vwl('crazy') ~ 2
vwl('razy') ~ 2
vwl('azy') ~ 2
vwl('zy') ~ 1
vwl('y') ~ 1
vwl('') ~ 0

What is the really simplest base case!?
max(L) return the biggest element in L, given that L has at least one element!

Intuit:

max([9,8,5]) \sim 9

max([8,5]) \sim 8

max([2,8,5]) \sim 8

max([8,5]) \sim 8

max([2,8,5]) \sim 8

max([9,8,5]) \sim 9

max([5]) \sim 5

Implement:

def max(L):
    return the biggest element in L, given that L has at least one element!
zeroest(L)

return the element of L closest to 0; L will have at least one element

Consider these examples:

zeroest([4,-3,42]) ~ -3

zeroest([1,-3,42]) ~ 1

zeroest([-3,42]) ~ -3

zeroest([42]) ~ 42

Note: The function abs is built-in to Python....
20 minute recursive work + break
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
```

I guess Python's the *in* thing.
**Answer:** `pow`  

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

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

\[
pow(2,2) \sim 2 \times 2
\]

\[
pow(2,1) \sim 2
\]

\[
pow(2,0) \sim 1
\]

Anything to the zero power is 1.0

**intuit**

**pow(b,p)**  

return \(b^{p}\), but using only multiplication times \(b\)

**pow(b,p) = pow(b,p-1) * b**

**base cases**

\[
pow(b,0) = 1.0
\]

**identify**

**self-similarities**

**def** `pow(b,p)`:

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

\[
\text{return } 1.0
\]

\[
\text{else:}
\]

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

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

**implement**
**def vwl(s):**

if s == '':
    return 0

elif s[0] in 'aeiouy':
    return 1 + vwl(s[1:])

else:
    return vwl(s[1:])

**vwl(s)** return the number of vowels in the string s, s may or may not be empty.

**vwl( ) = 0** base cases

let's count y!

vwl( 'crazy') ~ 2  
vwl( 'razy' ) ~ 2  
vwl( 'azy' ) ~ 2  
vwl( 'zy' ) ~ 1  
vwl( 'y' ) ~ 1  
vwl( '' ) ~ 0  

**Answer: vwl**

What is the really simplest base case!?

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

```python
def max(L):
    if len(L) == 1:
        return L[0]
    elif L[0] > max(L[1:])
        return L[0]
    else:
        return max(L[1:])
```

max(L) is \( L[0] \), if \( L[0] > \max(L[1:]) \)

max(L) is \( \max(L[1:]) \), otherwise

self-similarities

base cases

identify

implement

max( [9,8,5] ) ~ 9

max( [8,5] ) ~ 8

max( [2,8,5] ) ~ 8

max( [8,5] ) ~ 8

max( [5] ) ~ 5

max(L) return the biggest element in L, given that L has *at least one element*!
zeroest( L )

Consider these examples:

zeroest( [4, -3, 42] ) ~ -3

zeroest( [1, -3, 42] ) ~ 1

zeroest( [-3, 42] ) ~ -3

zeroest( [42] ) ~ 42

don't forget the base case!

intuit

zeroest(L)

Zeroest(L)

zeroest( L ) is L[0]

Note: The function abs is built-in to Python....

identify

Implement

Self-similarities & base cases

def zeroest(L):
    if len(L) == 1:
        return L[0]
    rest = zeroest(L[1:])
    if abs(L[0]) < abs(rest):
        return L[0]
    else:
        return rest

Zeroest(L)

return the element of L closest to 0;
L will have at least one element

base cases

self-similarities

The key to understanding recursion is, first, to understand recursion.

- former CS 5 student

Good luck with Homework #1

but wait, there's more!

tutors – Montana and Taylor! – Mon eve.
Etch-a-Sketch?

Wednesday's lab – and the first problem in hw#2...

No way this is real… but it is!
Etch-a-Sketch?

No way this is real... but it is!
more usual etch-a-sketch work...
import time
from turtle import *

def draw():         # define it!
    shape('turtle')
    # pause
time.sleep(2)
    # drawing...
    width(5)
    left(90)
    forward(50)
    right(90)
    backward(50)
    down() or up() is the pen on the "paper"?
    color('darkgreen')
    tracer(1) or tracer(0)
    width(5)

# run it and done!
draw(); done()

http://docs.python.org/library/turtle.html
Turtle happiness?

you can always run

```python
done()
```

after turtle drawing

This releases control of the turtle window to the computer (the operating system).

*don't* use it in your functions, however

To be honest, it seems that *not* every machine needs this...
**Single-path recursion**

1. Let's try this with recursion:

   ```python
   def tri(n):
       # draws a triangle
       if n == 0:
           return
       else:
           forward(100)  # one side
           left(120)     # turn 360/3
           tri(n-1)      # draw rest
   
   def poly(n, N):
       # draws a triangle
       if n == 0:
           return
       else:
           forward(100)  # one side
           left(360.0/N)  # turn 360/N
           poly(n-1, N)   # draw rest
   ```

2. How about *any* regular N-gon?

   ```python
   def tri():  # define it!
       """ a triangle! """
       forward(100)
       left(120)
       forward(100)
       left(120)
       forward(100)
       left(120)

   # run + done
   tri(); done()
   ```

I don't know about `tri`, but there sure is NO `return` ...!
Be the turtle!

(1) What would `chai(100)` draw?

```python
def chai(dist):
    """ mystery fn! """
    if dist < 5:
        return
    forward(dist)
    left(90)
    forward(dist/2.0)
    right(90)
    right(90)
    forward(dist)
    left(90)
    left(90)
    forward(dist/2.0)
    right(90)
    backward(dist)
```

Extra! How could you make this a bull (or a bear) market?

Extra #2! What if the line `chai(dist/2)` were placed between the two right(90) lines? And/or between the two left(90) lines?

Have `rwalk` draw a "stock-market" path:

N steps of 10 pixels each. Use recursion.

```python
from random import *

def rwalk(N):
    """ make N 10-pixel steps, NE or SE """
    if N == 0:
        return
    elif choice(['left','right']) == 'left':
        left(45)
        forward(10)
    else:
        # this handles 'right'
        ?
```

one possible result of `rwalk(20)`
from random import *

def rwalk(N):
    """ make N 10-px steps, NE or SE """
    if N == 0: return

    elif choice(['left','right'])=='left':
        left(45)
        forward(10)
        right(45)
        rwalk( N-1 )

    else: # 'right'
        right(45)
        forward(10)
        left(45)
        rwalk( N-1 )

rwalk(N) is a random "stock market" walk...

What if we didn't turn back to face east each time?

Single-path (or counting) recursion
Cyriak: *conceptually disruptive* recursion...

is the *branching*, not the *single-path* variety.
def chai(dist):
    """mystery!""
    if dist<5:
        return
    forward(dist)
    left(90)
    forward(dist/2.0)
    right(90)
    right(90)
    forward(dist)
    left(90)
    left(90)
    forward(dist/2.0)
    right(90)
    backward(dist)

How could you add more to each T's tips?

Why are there two identical commands in a row ~ twice!?
**Branching recursion**

Now, what does `chai(100)` do?

```python
def chai(dist):
    """ mystery! ""
    if dist<5:
        return
    forward(dist)
    left(90)
    forward(dist/2.0)
    right(90)
    chai(dist/2)
    right(90)
    forward(dist)
    left(90)
    chai(dist/2)
    left(90)
    forward(dist)
    left(90)
    forward(dist/2.0)
    right(90)
    backward(dist)
```
fractal art

spiral(100, 90, 0.8)

spiral( initLength, angle, multiplier )
lab ~ hw2pr1

fractal art

spiral(100, 90, 0.8)

spiral(100, 90, 0.8)

spiral(initLength, angle, multiplier)
svtree(trunkLength, levels)

svtree(100, 5)

levels == 5
levels == 4
levels == 3
levels == 2
levels == 1
levels == 0
(no drawing)
What steps does the turtle need to take before recursing?
svtree( trunkLength, levels )

Be sure the turtle always returns to its starting position!

step #1: go forward...

step #2: turn a bit...

step #3: draw a smaller svtree!

step #4: turn to another heading

step #5: draw another smaller svtree!

step #6: get back to the start by turning and moving!

svtree( 100, 5 )

levels == 5

levels == 4

levels == 3

levels == 2

levels == 1

levels == 0
(no drawing)
svtree( trunkLength, levels )

Be sure the turtle always returns to its starting position!

that means it will finish the recursive call right here!

so that it can change heading and draw another one...

levels == 0
(no drawing)
The Koch curve

snowflake(100, 0)  snowflake(100, 1)  snowflake(100, 2)

snowflake(100, 3)  snowflake(100, 4)  snowflake(100, 5)
Recursive art? Create your own...

Happy turtling in lab on Wed!

seven-cornered confetti

What? Way too happy to be art...
My recursive compositions burninate even CyriaK's brain!