C.R.J.!
Today's Zoom-Chat Q'n...

What is the strongest pop music memory (middle-school or before) that you'd rather not remember?
Recursion

As close as CS gets to magic

a.k.a., CS's version of mathematical induction
Recursion example: $\text{numis}(s)$

total # of i's in 
'\text{\underline{i}}<3\text{five}'

is

# of i's in 
'\text{\underline{i}}'

+ 

# of i's in 
'\text{<3five}'

That's "num eyes"!
Recursion example: \( \text{numis}(s) \)

Total # of i's in \( '\text{alien}' \)

\[
\text{is} \quad \text{# of i's in} \quad 'a' \quad + \quad \text{# of i's in} \quad '\text{lien}'
\]

\text{first} \quad \text{rest}
Recursion example: \( \text{numis}(s) \)

\[
\text{total \# of i's in 'aliiien'}
\]

\[
\text{is}
\]

\[
\text{\# of i's in 'a'} + \text{\# of i's in 'liiien'}
\]

\( \text{first} \) \quad \text{rest} \)
Recursion example: \( \text{numis}(s) \)

Analysis...

\[
\text{total # of i's in } \quad s \\
\text{is} \\
\text{# of i's in } \quad s[0] \\
+ \\
\text{# of i's in } \quad s[1:] \\
\]

... via self-similarity!
Recursion example: $\text{numis}(s)$

Total # of i's in $s$

Analysis...

Vocab is super-helpful!

# of i's in $s[0]$ + # of i's in $s[1:]$
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!
Welcome back to CS 5!

Homework 2

Problem 0: Reading + response...

Problem 1: Recursion! Do these during lab... due today (Mon. ~ lab)

Problem 2: PythonBat functions! In lab or beyond... due Sunday evening

Our CS5 delegates

Wally

Alien
Welcome back to CS 5!

Homework 3

Problem 0: Reading + response...  
Problem 1: They're named 3+4...  
Problem 2: Sleepwalking simulation!  
Problem 3+4: Picobot Lab...  

Our CS5 delegates

Wally

Alien
Computation's Dual Identity

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

accessed through *functions*…

*Functions!* It's no coincidence this starts with *fun*!
C.R.J.!

me(number)
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!
Use variables!

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

OK: we humans work better with named variables...

... but why would even computers "prefer" the top version, too?

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

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...)
**Speaking of tests!**

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warmup-1</td>
<td>Simple warmup problems to get started, no loops (solutions available)</td>
</tr>
<tr>
<td>Warmup-2</td>
<td>Medium warmup string/list problems with loops (solutions available)</td>
</tr>
<tr>
<td>String-1</td>
<td>Basic python string problems -- no loops</td>
</tr>
<tr>
<td>String-2</td>
<td>Medium python string problems -- 1 loop.</td>
</tr>
<tr>
<td>List-1</td>
<td>Basic python list problems -- no loops.</td>
</tr>
<tr>
<td>List-2</td>
<td>Medium python list problems -- 1 loop.</td>
</tr>
</tbody>
</table>

**Python Help**

- Python Example Code
- Python Strings
- Python Lists
- Python If Boolean
- Code Badges

Problem 2 ~ due Thursday

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

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

an = dbl(20) + 2
    this is a value for further use!
    yes!
an = dblPR(20) + 2
    this turns lightbulbs on!
    ouch!
```

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

**return** yields the function call's **value** ...
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
def g(x):
    result = 4*x + 2
    return result

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

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

What is $f(2)$ here?

Submit screenshot:
day2quiz1
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 \texttt{demo(15)} here?

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

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

The stack

Call: demo(15)

Local variables:
x = 15
y = 5
z = ????

They stack.
**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)  stack frame
local variables:
  x = 15
  y = 5
  z = ???
returns 22

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

15

*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 = ?????

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...

"the stack"

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

How functions work...

15

call: demo(15)
local variables:
x = 15
y = 5
z = 22
return 42

"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...

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

they stack.
How functions work...

What's $f(2)$?

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

"the stack"
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...

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

The stack:

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

- **Call: f(1)**
  - Local variables: `x = 1`
  - Need `f(0)`

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

stack frame
x = 2
need f(1)

stack frame
x = 1
need f(0)

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

1
```

How do we compute the result?
How functions work...

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

---

<table>
<thead>
<tr>
<th>Stack Frame</th>
<th>Local Variables</th>
<th>Call</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="stack_frame1.png" alt="Stack Frame" /></td>
<td>x = 2</td>
<td>f(2)</td>
</tr>
<tr>
<td><img src="stack_frame2.png" alt="Stack Frame" /></td>
<td>f(0) = 12</td>
<td>f(1)</td>
</tr>
</tbody>
</table>

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

stack frame

x = 2
need f(1)

local variables:

stack frame

x = 1
f(0) = 12
result = 22

local variables:

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

How functions work...

What's this return value?
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
- f(1) = 22
- result = 42

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

call: f(2)

local: x = 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...
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 are software's cells ... each one is a self-contained computational unit!
How functions work...

Functions are software's **cells** ... ... each one is a **self-contained computational unit**!

```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...
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!**
sequential

iteration

self-similar

recursion

problem-solving paradigms
Thinking *sequentially*

**math**

\[ 5! = 120 \]

**cs**

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

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

**factorial**

\[ 5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \]

**math**

\[ 5! = 120 \]

**cs**

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

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

**next week + 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 fac w/ a smaller version of itself?**

\[ \text{fac}(N) = \]
Thinking recursively

**Recursion ~ self-similarity**

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

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

<table>
<thead>
<tr>
<th>Can we express ( \text{fac} ) w/ a smaller version of itself?</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{fac}(N) = N \times (N-1) \times \ldots \times 3 \times 2 \times 1 ]</td>
</tr>
<tr>
<td>[ \text{fac}(N) = N \times \text{fac}(N-1) ]</td>
</tr>
</tbody>
</table>

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:

```
import requests
import numpy as np
import pandas as pd
import sys
```
Recursion
the dizzying dangers of having no base case!

This "works" \textit{but doesn't work!}

\begin{verbatim}
def fac(N):
    return fac(N)
\end{verbatim}
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 ...
en.wikipedia.org/wiki/Recursion - Cached - Similar

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. ...
en.wikipedia.org/wiki/Recursion_(computer_science) - Cached - Similar

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 ...
mathworld.wolfram.com/Recursion.html - Cached - Similar

recursion
Definition of recursion, possibly with links to more information and implementations.
www.itl.nist.gov/div897/sqg/dads/HTML/recursion.html - Cached - Similar

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

How about an escape from recursion itself?!
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...

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

Conceptual

Acting recursively

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

Actual
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 * 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)
```

```
fac(5)
5 * fac(4)  
4 * fac(3)  
3 * fac(2)  
2 * fac(1)  
1.0
```

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 $N = 3$

$$
5 \times \text{fac}(4) \\
4 \times \text{fac}(3) \\
3 \times 2.0
$$
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...*

```python
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

Base Case

Self-similar design

problem-solving *paradigm*
Recursion

Base Case

Self-similar design

Next: recursive-function DESIGN
value of $5 \times 4 \times 3 \times 2 \times 1$ is $\text{fac}(x)$

Base case: 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 n times

value of 1+1+1+1+1

value of 1 +

Base case:
plusone(0) should return ___
plusone\( (n) \) adds 1 a total of \( n \) times

\[
\text{plusone}(5) = 1 + 1 + 1 + 1 + 1
\]

\[
\text{value of } 1 + \text{value of } 1+1+1+1+1
\]

Base case:
\[
\text{plusone}(0) \text{ should return } ___
\]
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)
The value of $2^5$ is $2*2*2*2*2$.

The value of $2^*$ is ___.

Base case:
$\text{pow}(2, 0)$ should return ___?
The value of $2^5$ is $2*2*2*2*2$.

The value of $2^2$ is $2*2$.

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...?
```python
def pow(b, p):
    """
b**p, defined recursively!
    ""
    if p == 0:
        return 1.0
    elif p < 0:
        return b * pow(b, p-1)
    else:
        return b * pow(b, p-1)
```
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

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 Smile to Happy Smile’ 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:

\[ s = 'aliilien' \]

- 'a'
- 'liilien'

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

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

\[ s = 'aliiien' \]

- handle the **first** Human!
- 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** Human!
- **recurse the rest** Machine!
Design patterns...

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

- Do the **first** piece: \[ L[0] \text{ or } s[0] \]
- Recurse with the **rest**: \[ L[1:] \text{ or } s[1:] \]
- Combine! Make sure all types match...
- Handle base cases, with **if** ...
Base case:
numis("") should return ___ ?
def numis(s):
    """ # of i's in s """
    if s == ' ':
        return
    elif s[0] == 'i':
        return
    else:
        return

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

# of i's in 'xlii' + # of i's in 'x' + # of i's in 'lii'
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?
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
Leap before you look!

Try these two...
# of vowels in 'eerie' is

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

Base case:
vwl('') should return ___ ?
def vwl(s):
    """ # of vowels in s """

    if s == '':
        return 0

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

    else:
        return 0 + vwl(s[1:])

What seven-letter s maximizes vwl(s)?
Python is... **in**

```python
>>> 'i' in 'team'
False

>>> 'cs' in 'physics'
True

>>> 'i' in 'alien'
True

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

>>> 42 in [[42], '42']
False
```

I guess Python's the **in** thing
def vwl(s):
    """ # of vowels in s
    """
    if s == '':
        return 0
    else:
        return __________

vwl("icier")

Try writing keepvwl(s):

keepvwl("slyly")

Submit screenshot: hw2quiz2
def vwl(s):
    """ # of vowels in s
    """
    if s == '':
        return 0

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

    else:
        return 0+vwl(s[1:])

def keepvwl(s):
    """ returns ONLY the vowels in s!
    """
    if s == '':
        return ________

    elif s[0] in 'aeiou':
        return ____________________

    else:
        return ____________________

Examples:

    keepvwl("icier")
    "iie"

    keepvwl("fjord")
    "o"

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

vwl("icier")
1+vwl('cier')
vwl('cier') = 0+vwl(ier)
vwl(ier)= 1+vwl(er)
vwl(er)=1+vwl(r)
vwl(r)=0+0
1+0+1+1+0

Try writing  keepvwl(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:])

Examples:

keepvwl("icier")
"iie"

keepvwl("fjord")
"o"

keepvwl("slyly")
"
```
Base case:
keepvwl("") should return ___ ?
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:])
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!
or - one layer up!?
Base case:
if \( \text{len}(L) == 1 \), what should \( \text{max}(L) \) return?
def max(L):
    """ returns the max of L! """
    if len(L) == 1:
        return M
    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]

either -7

is

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

Dive in ~ with lab #2

It's the eeriest!

tutors @ LAC + 4C's  Th/F/Sa/Su/Mon.
this week's *hw2pr0*

**Category:** U.S. Cities. **Clue:** *Its largest airport is named for a World War II hero, its second largest for a World War II battle.*

*Watson*