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!
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: $numis(s)$

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

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

$S[0]$

$S[1:]$

$+$

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

The total number of 'i's in 'alien' is

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

\( \text{numis}(s[0]) \)
Recursion example:  \textit{numis(s)}  \\

\begin{align*}
\text{total \# of i's in } & \text{'aliiiiien'} \\
\text{is} & \\
\text{\# of i's in } & \text{'a'} \\
\text{\# of i's in } & \text{'liiiien'}
\end{align*}
Recursion example: $\text{numis}(s)$

Analysis...

Total # of i's in 'aliien'

# of i's in 'a'

+ # of i's in 'liiien'

... 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

Is this Python??

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 **functions**…

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

```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!
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!

I'm happy about this, too!

OK: we humans work better with named variables.

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

Aargh!

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

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

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

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

call = dbl(20) + 2
print(ouch)
```

- `return` yields the function call's value...
- `print` changes pixels on the screen...
- `ans = dbl(20) + 2` is a value for further use! Yes!
- `ans = dblPR(20) + 2` turns lightbulbs on! Ouch!

...which the shell then prints!
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
**Quiz**

How f's work...

```python
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...
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

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 = ?????

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

15

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

result = 22

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...
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
```

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

"the stack"

they stack.
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

"the stack"

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

call: f(2)
local variables: x = 2
need f(1)
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

How functions work...

"the stack"

f(0)
f(1)
f(2)

stack frame

local variables:
x = 2
need f(1)

stack frame

local variables:
x = 1
need f(0)
How functions work...

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

![Stack diagram](image)
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 do we compute the result?
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)

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

f(1)

# stack frame
x = 1
f(0) = 12
result = 22

f(2)

# stack frame
x = 2
need f(1)
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

result = f(2)

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

How functions work...

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

How functions work...

"the stack"

stack frame

call: f(2)

for x in range(2):
x = 2

result = 42

the result then gets returned...

2
How functions work...

```python
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...

"the stack"
How functions work...

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

Pass these eastward!
... each one is a self-contained computational unit!

functions stack.
sequential

iteration

self-similar

recursion

problem-solving paradigms
Thinking *sequentially*

**factorial**

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

```csharp
fac(5) = 5*4*3*2*1
```

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

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

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

\[
\text{fac}(N) = N\times(N-1)\times\ldots\times3\times2\times1
\]

\[
\text{March + beyond...}
\]
Thinking *recursively*

**factorial**

\[ 5! = 120 \]

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

**CS**

\[ \text{fac}(5) = 5 \times \text{fac}(5-1) \]

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

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

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

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

Thinking recursively...

Recursive case (too short?)

Base case

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  Actual
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)
```

Stack frames with:
- \( N = 5 \)
- \( N = 4 \)
- \( N = 3 \)
- \( N = 2 \)
- \( N = 1 \)
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** \( N = 5 \)
  
  \( 5 \times \text{fac}(4) \)

- **Stack frame with** \( N = 4 \)
  
  \( 4 \times \text{fac}(3) \)

- **Stack frame with** \( N = 3 \)
  
  \( 3 \times \text{fac}(2) \)

- **Stack frame with** \( N = 2 \)
  
  \( 2 \times 1.0 \)
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 = 5

Stack frame with N = 4

Stack frame with N = 3
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 * fac(4)`

- **stack frame with N = 4**
  - `4 * 6.0`
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 $N = 5$

$5 \times 24.0$
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!

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

is

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

Base case:
fac(0) should return 1
def fac(x):
    """ factorial! Recursively! """
    if x == 0:
        return 1
    else:
        return x * fac(x-1)
value of $\underbrace{1+1+1+1+1}_5$ is plusone($n$) adds 1 a total of $n$ times

plusone(5) value of 1 + plusone(4)

Base case: plusone(0) should return ___
value of 1 + value of 1+1+1+1+1

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

value of 2 \(*\)

Base case:
pow(2,0) should return __ ?
value of \(2 \times 2 \times 2 \times 2 \times 2\) is \(\text{pow}(2,5)\)

value of 2 \* value of \(2 \times 2 \times 2 \times 2\)

Base case:
\text{pow}(2,0) \text{ should return } __ \text{ ?}
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 1/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!
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:

\[ s = '\textit{aliilen}' \]

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 = 'aliil lien'

's[0]' 'liilien'

handle the **first**

s[1:]'

recurse the **rest**

Human! Machine!
Design patterns...

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

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

- Handle the **first** \( L[0] \)
- Recurse the **rest** \( L[1:] \)

\( 3 \quad \rightarrow \quad [1, 4, 1, 5, 9] \)

*Human!*
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 \texttt{if} ...**
numis('xlii')

# of i's in 'xlii'

# of i's in 'x' + # of i's in '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 + ~
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 ___?
```python
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:])
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!?
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):
    """ # of vowels in s
    ''"
    if s == '':
        return ________________
    elif s[0] in 'aeiou'::
        return ________________
    else:
        return ________________
Python is...  

```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
```
keepvwl('pluto')

keep vowels in 'pluto'

is

keepvwl('p') + keepvwl('luto')

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

# Example usage:
print(keepvwl(''))  # Output: ''
print(keepvwl('abc'))  # Output: 'a'
print(keepvwl('hello'))  # Output: 'eol'
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 = ________________  # The max of the REST of L

  if L[0] > M:
    return ________________
  else:
    return ________________
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 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 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
```python
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.

It's the eeriest!
**pow(b,p)**  

**pow(b,p)** returns $b^{**p}$, but using only multiplication times $b$

---

**pow(2,4)** ~ $2*2*2*2$  
$= 16$

**pow(2,3)** ~ $2*2*2$  
$= 8$

**pow(2,2)** ~ $2*2$

**pow(2,1)** ~ 2

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

intuit

Anything to the zero power is 1.0

---

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

```python
if p == 0:
    return ____

# self-similarities 
# base cases
```

---

**base cases**

- $b^0 \cdot \text{pow}(b, 0) = 1$

---

**self-similarities**

- anything to the zero power is 1.0

---

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

---

**implement**
The function `vwl(s)` returns the number of vowels in the string `s`, `s` may or may not be empty.

**Example**

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

**Base Cases**

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

**What is the really simplest base case!?**

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

\[
\begin{align*}
pow(2,4) & \sim 2 \cdot 2 \cdot 2 \cdot 2 = 16 \\
pow(2,3) & \sim 2 \cdot 2 \cdot 2 = 8 \\
pow(2,2) & \sim 2 \cdot 2 \\
pow(2,1) & \sim 2 \\
pow(2,0) & \sim 1
\end{align*}
\]

\underline{intuit}

\[
pow(b, p) = pow(b, p-1) \cdot b
\]

\underline{identify}

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

\underline{base cases}

\underline{self-similarities}

\underline{implement}

\[
\begin{align*}
def pow(b, p): \\
    &\text{if } p == 0: \\
    &\quad \text{return } 1.0 \\
    &\text{else:} \\
    &\quad \text{return } pow(b, p-1) \cdot b
\end{align*}
\]

\underline{Extra!} See if you can also handle \textit{negative} powers...