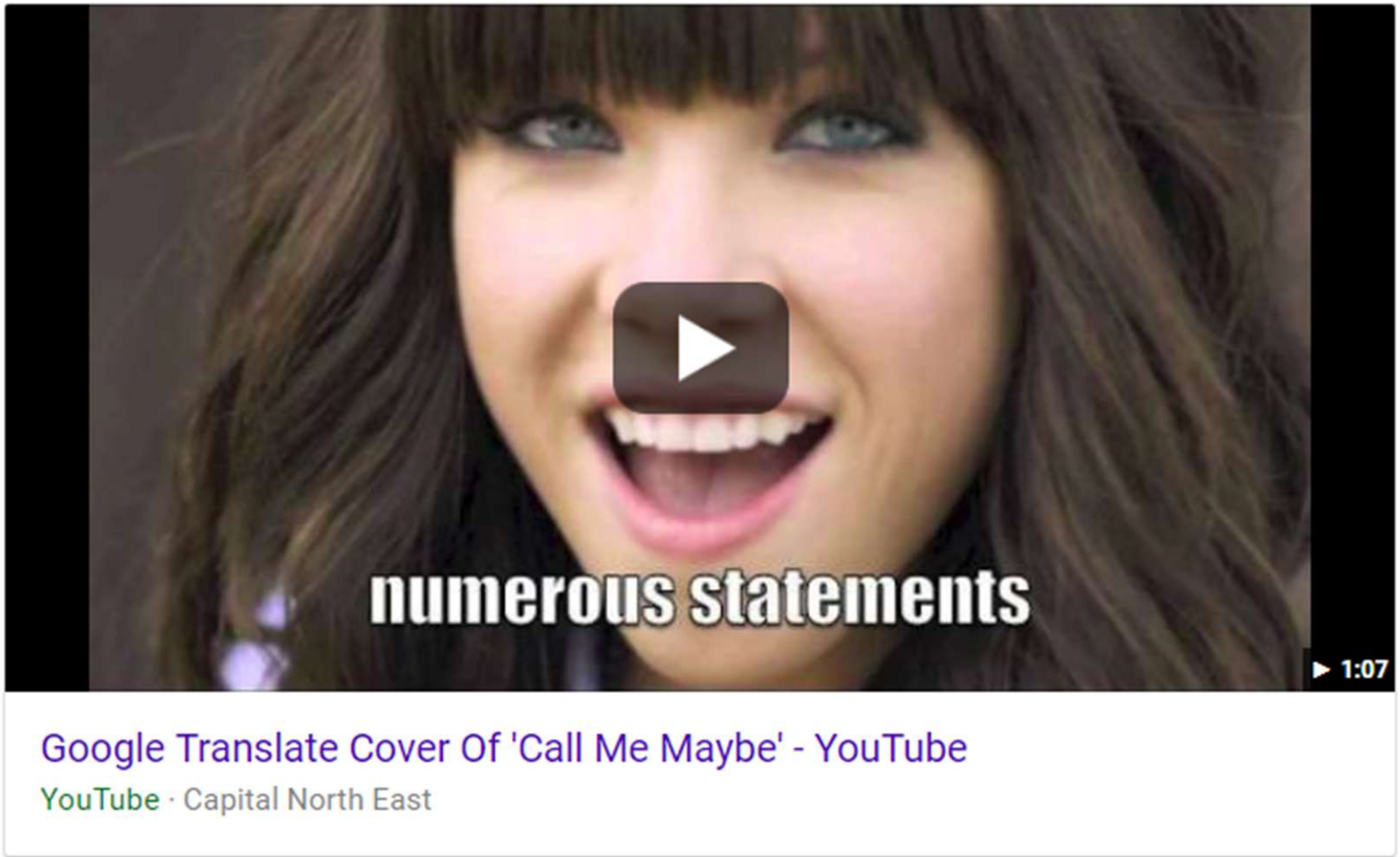
C.R.J.!



Google Translate Cover Of 'Call Me Maybe' - YouTube

YouTube · Capital North East

CS 5 … Today!

MAGIC!?

Recursi

s CS
gic

a

h

Need to do!

First four examples == given
Last four examples == question

PRINTING VERSIONS w/ BRANCHES!

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)*

total # of i's in
`'i<3five'`

s

is

# of i's in
`'i'`

s[0]

+

# of i's in
`'<3five'`

s[1:]

Recursion example: *numis*(`s`)

total # of i's in
`'alien'`

is

# of i's in
`'a'`
s[0]

+

# of i's in
`'lien'`

Recursion example: *numis(s)*

total # of i's in
`'aliiien'`

is

# of i's in
`'a'`

+

# of i's in
`'liiien'`

Recursion example: *numis(s)*

total # of i's in
`'aliiien'`

Analysis...

is

# of i's in
`'a'`

+

# of i's in
`'liiien'`

... via *self-similarity!*

CS 5 ... Today!

Jack Ma's Picobot "magic"

Recursion

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

As close as CS gets to magic

Tutoring hours: LOTS!

Hw #1 due this Monday, 9/17, at 11:59 pm

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

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

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

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

Career Fair + CS5!

AwkwardSelfie(Day) ...

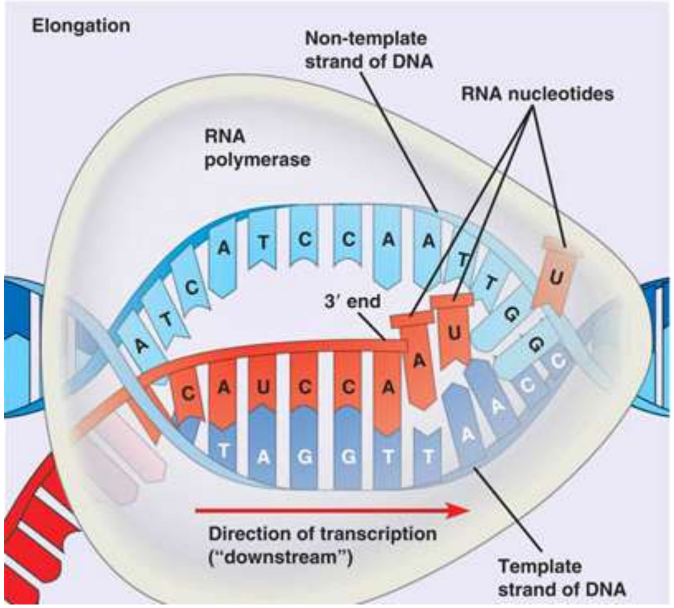
hw1

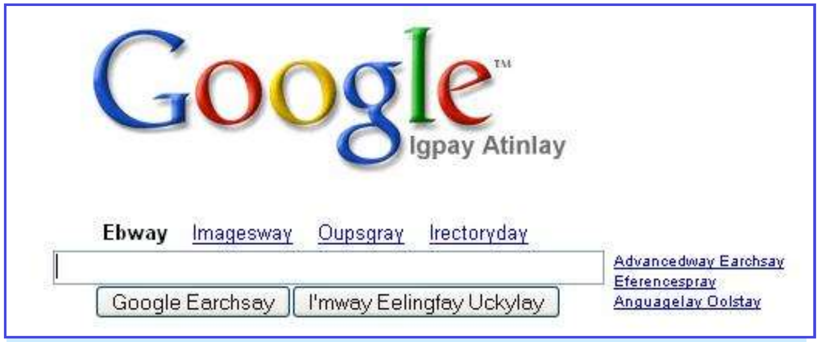
`if` you attended lab and submit pr1+pr2:
you get full credit for hw1pr1 and hw1pr2

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

Is this Python??

either way: submit pr1 + pr2
complete and submit hw1pr3

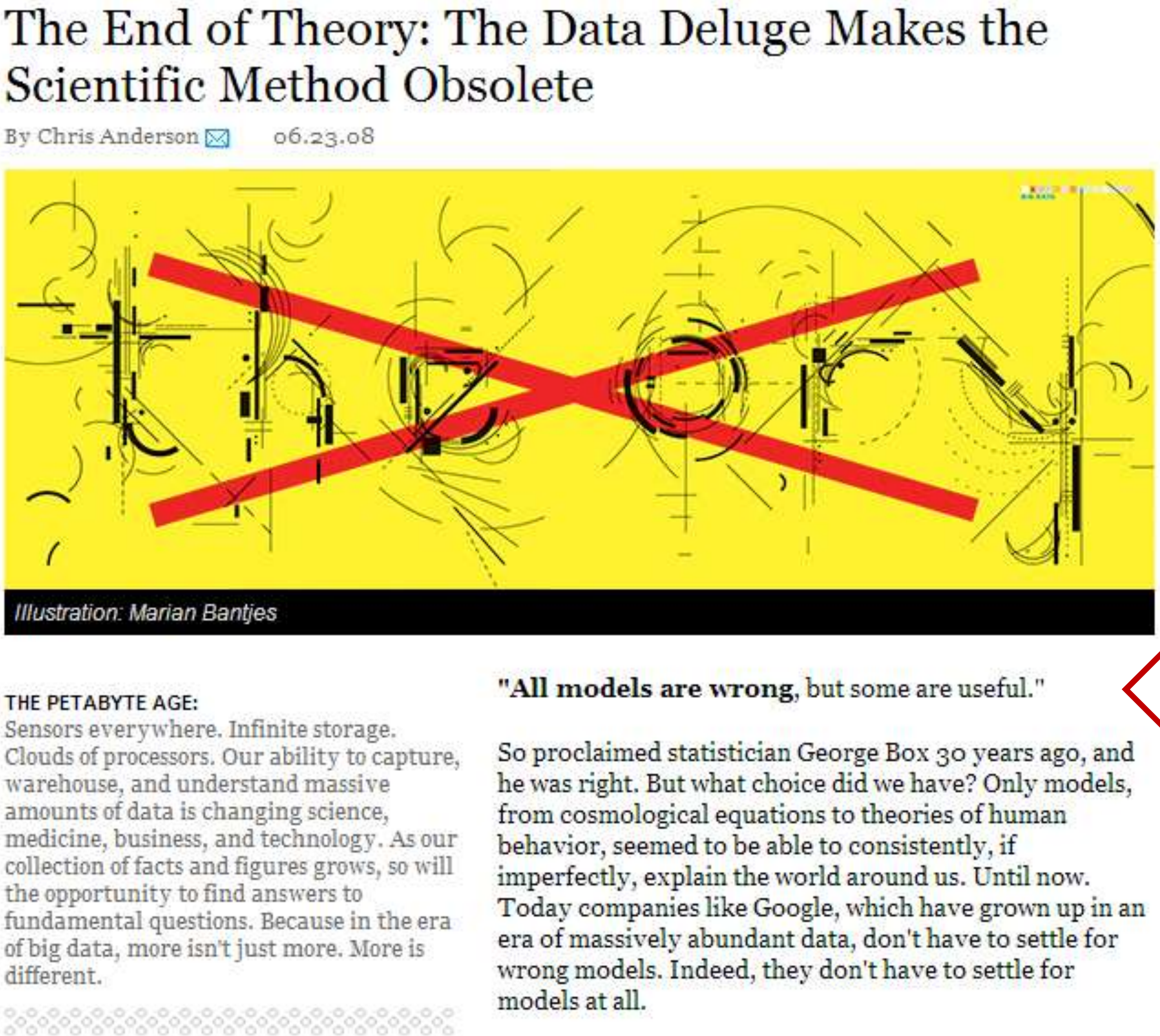




Extra Credit: *Pig Latin / CodingBat*

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: Marian Bantjes

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.

"**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.

Petabytes? This article is old-school!

Computation's Dual Identity

Computation

Data Storage

41

name: x
type: int
LOC: 300

memory location 300

42

name: y
type: int
LOC: 304

memory location 304

variables ~ boxes

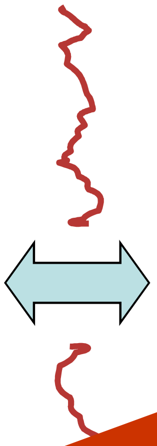
Last time

But what does the
stuff on this side
look like ?

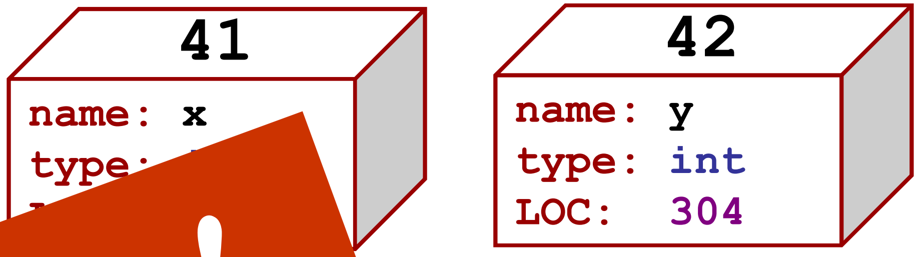
Computation's Dual Identity

accessed through ***functions***…

Computation



Data Storage



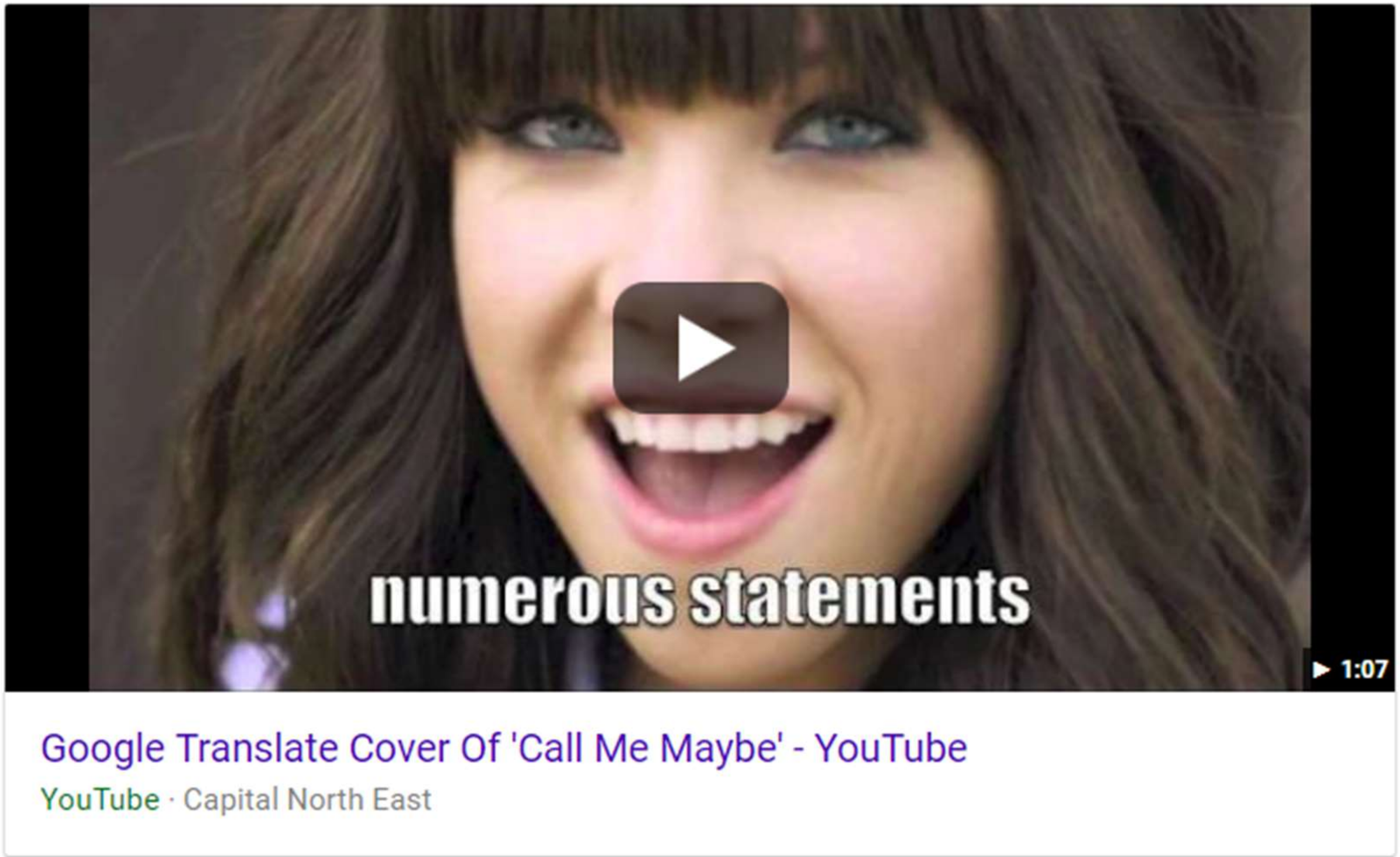
memory location 304

variables ~ boxes

Functions!

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

C.R.J.!



Google Translate Cover Of 'Call Me Maybe' - YouTube

YouTube · Capital North East

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 *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*

follow the data...

`'homework'`

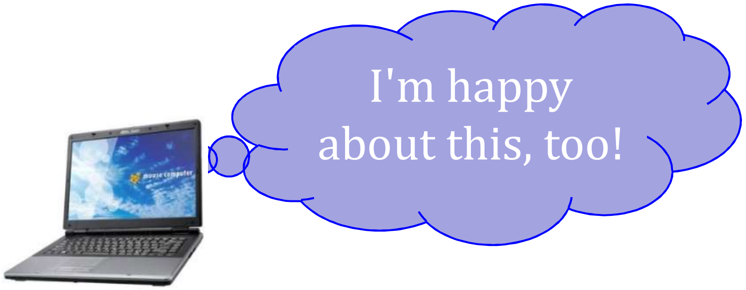
```
def flipside(s):
    """ flipside(s): swaps s's sides!
        input s: a string
    """
    x = len(s)//2
    return s[x:] + s[:x]
```

`4`

`'work'` `'home'`

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

Use variables!



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

these two functions
do the same thing...

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!

Test!

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

(1) function
definition

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

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

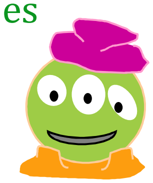
`assert`

(2) function
tests

`print`

We provide tests *(for now...)*

Redefining variables...

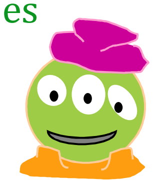


This program uses variables ***constantly***!

```
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]
```

Naming things!

This program uses variables ***constantly***!



name

signature line

```
def convertFromSeconds(s):
    """ convertFromSeconds(s): Converts an
          integ             nto a list of
          [days                   seconds]
        input s: an int
    """
    days = s // (24*60*60)   # total days
    s = s % (24*60*60)       # remaind
    hours =              ))  # total h
    s = s                    # remaind
    minute                   # total m
    s = s % 60               # remaind
    return [days, hours, minutes, s]
```

docstring

code block

in-line comments – these are optional in CS 5

return statement

`return` vs. `print`

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


ans = dbl(20)
```

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


ans = dblPR(20)
```

What's the difference ?!

return > print

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

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

this is a value for further use!

yes!

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

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

this turns lightbulbs on!

ouch!

`print` changes pixels on the screen...

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

... which the shell then prints!

`return` > `print`

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

changes the pixels (little ***lightbulbs***) on your screen

`return` > `print`

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

changes the pixels (little ***lightbulbs***) or

Quiz

How f'ns ***work***...

Name(s):

15

What is **`demo(15)`** here?

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

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

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

What is **`f(2)`** here?

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

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
```

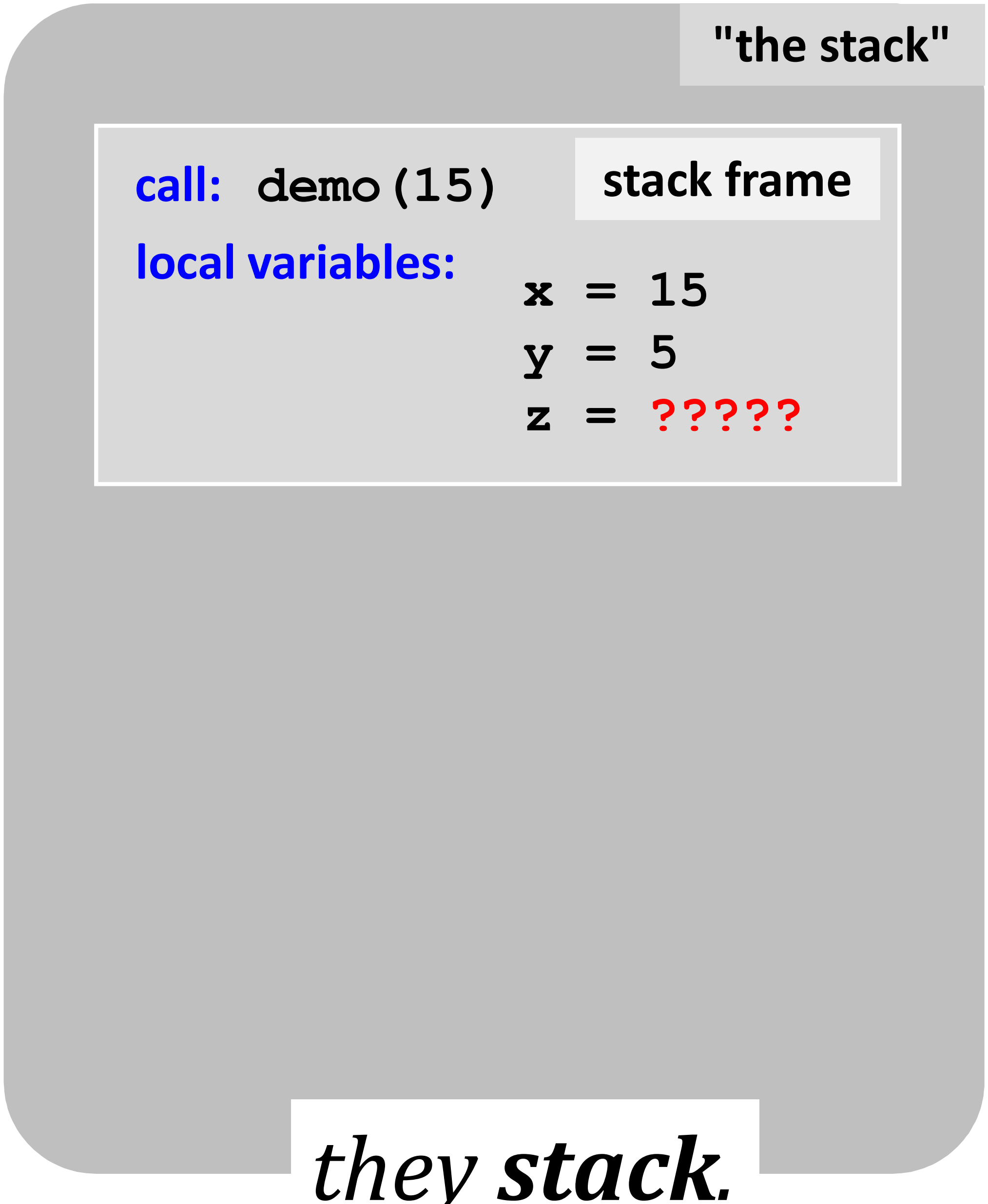
"the stack"

*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"

stack frame

call: `demo(15)`

local variables:

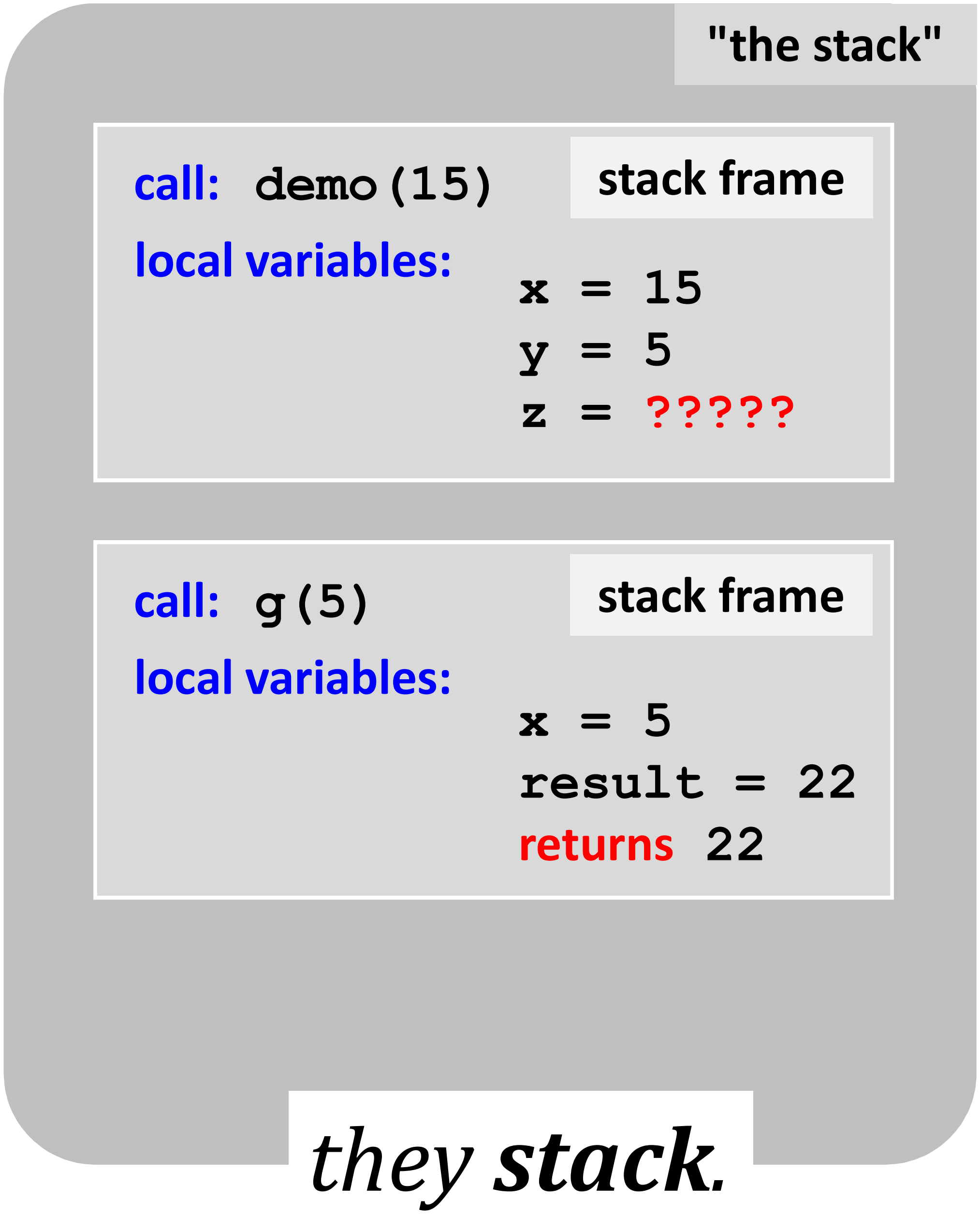
```
x = 15
y = 5
z = ?????
```

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"

call: `demo(15)` **stack frame**

local variables:

```
x = 15
y = 5
z = ?????
```

call: `g(5)` **stack frame**

local variables:

```
x = 5
result = 22
```

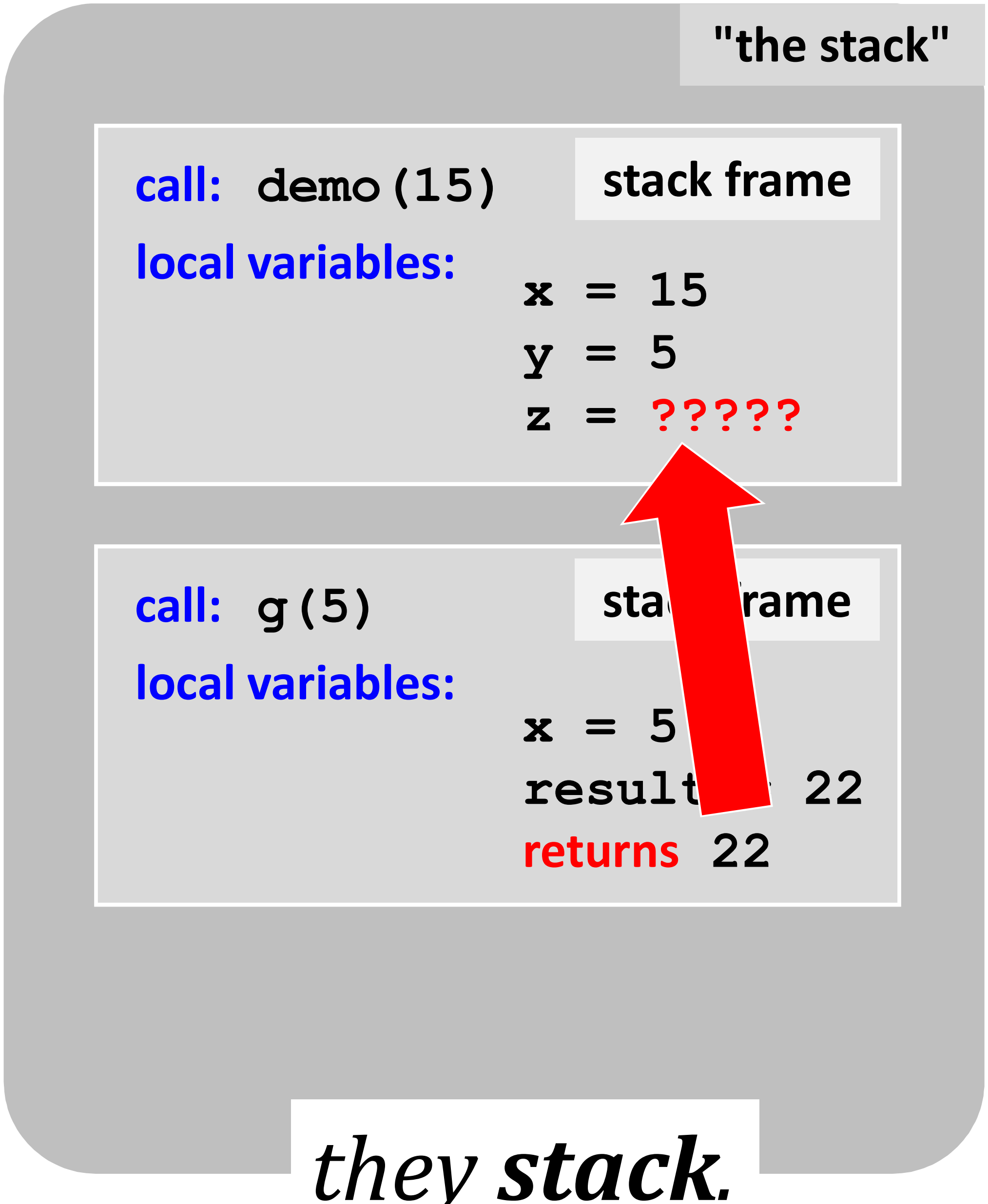
returns 22

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"

call: demo(15)

stack frame

local variables:

x = 15
y = 5
z = ?????

call: g(5)

sta rame

local variables:

x = 5
result 22
returns 22

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"

stack frame

call: `demo(15)`

local variables:

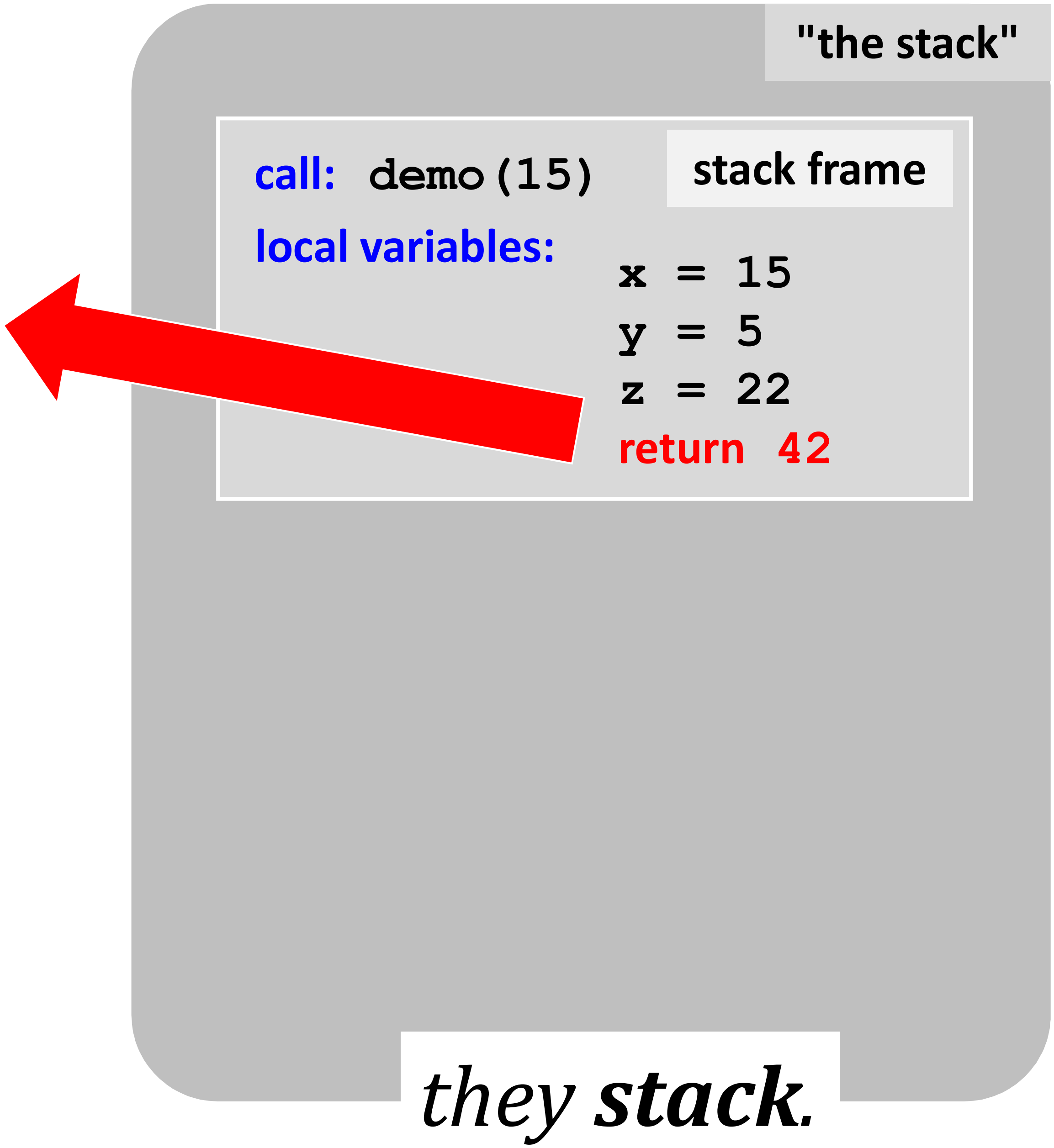
`x = 15`
`y = 5`
`z = 22`

*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"

stack frame

call: demo(15)

local variables:

x = 15
y = 5
z = 22
return 42

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
```

42

output

"the stack"

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

they ***stack.***

How functions work...

what's **`f(2)`** *?*

2

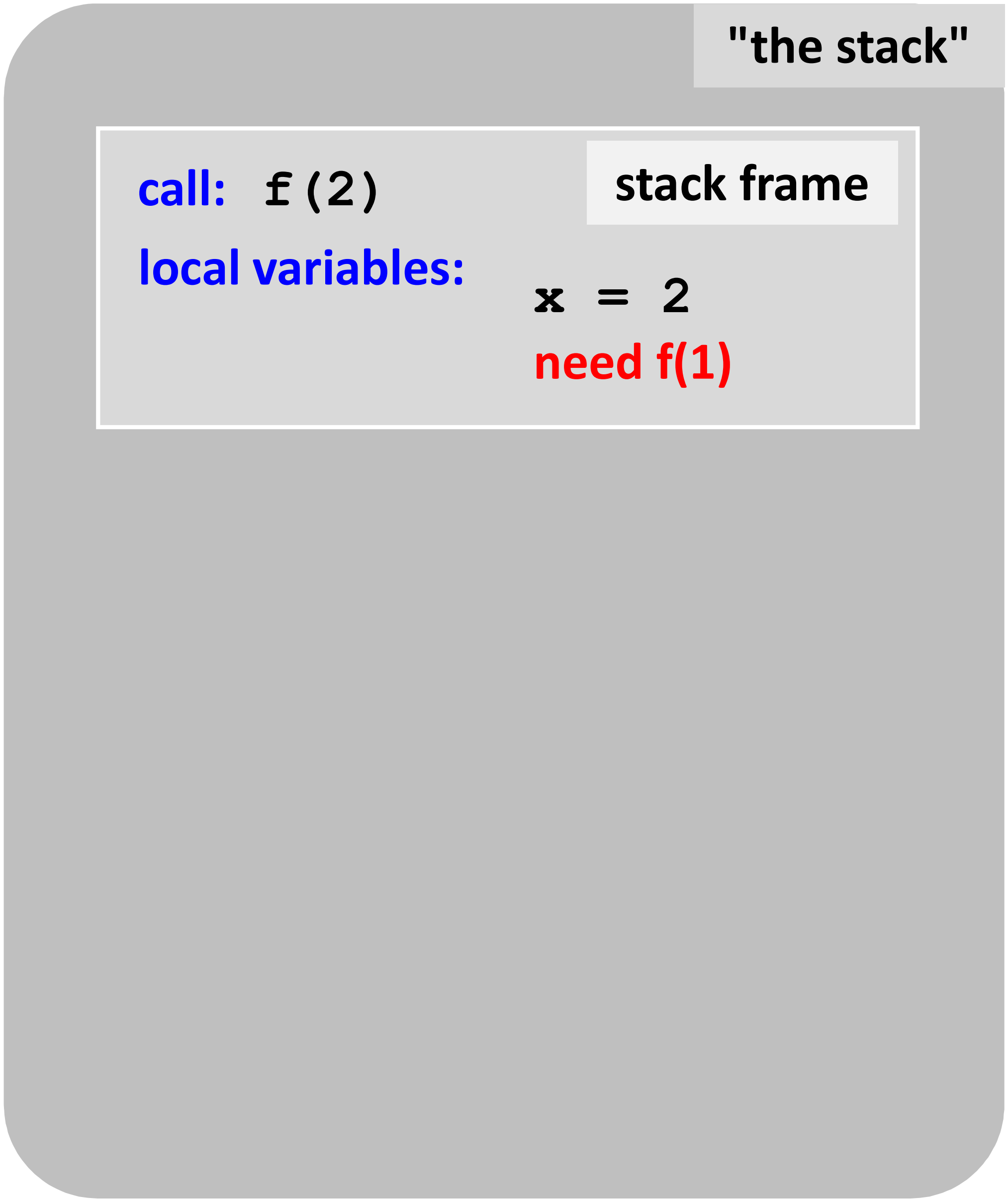
```
def f(x):
  if x == 0:
    return 12
  else:
    return f(x-1) + 10*x
```

"the stack"

How functions work...

2

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



"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

need f(1)

How functions work...

1

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

"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)

How functions work...

0

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

"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:

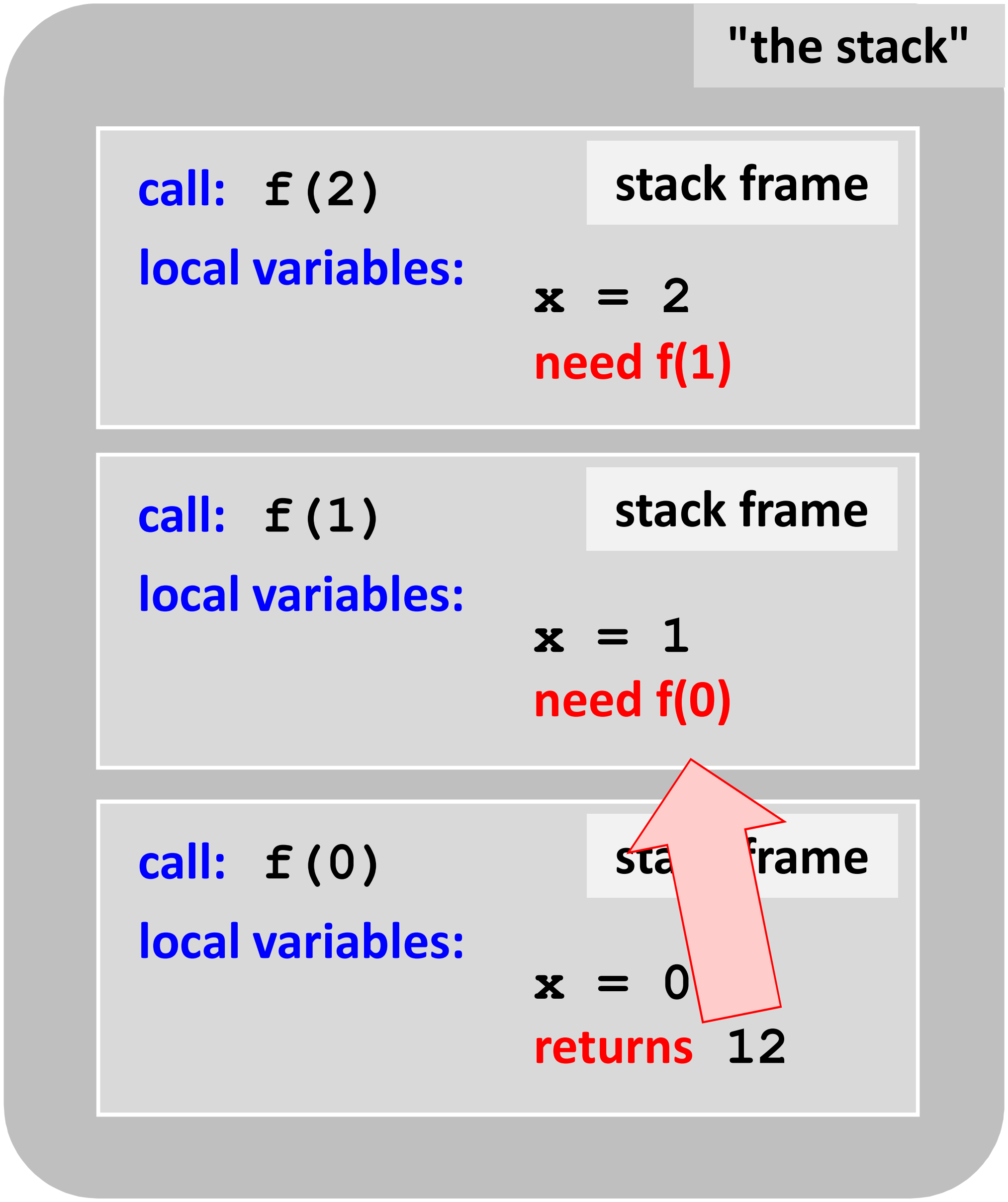
`x = 0`

returns 12

How functions work...

0

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



"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)

sta frame

call: `f(0)`

local variables:

`x = 0`

returns `12`

How functions work...

1

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

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

need f(1)

stack frame

call: `f(1)`

local variables:

`x = 1`

f(0) = `12`

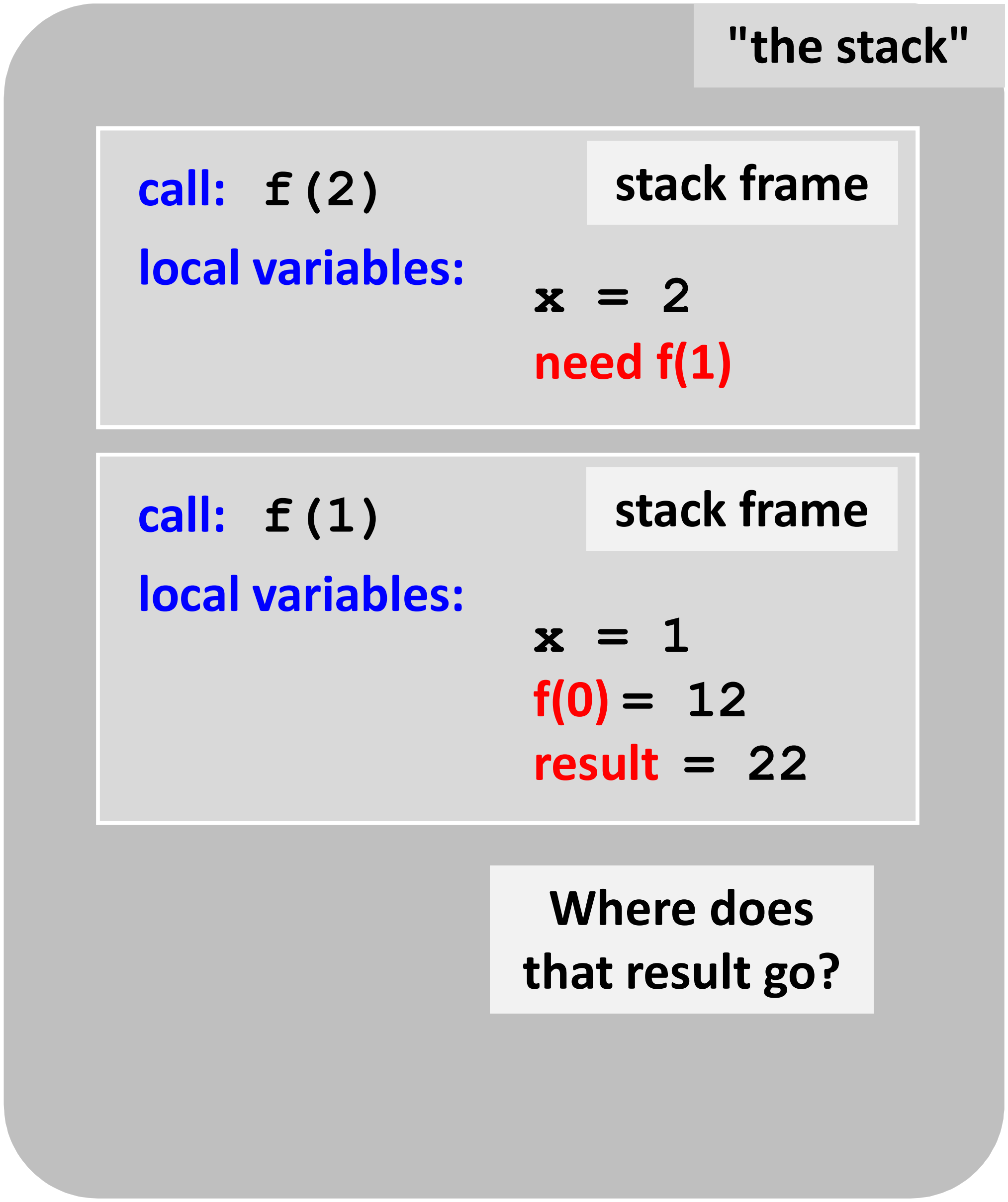
result =

How do we compute the result?

How functions work...

1

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



"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

need f(1)

stack frame

call: `f(1)`

local variables:

`x = 1`

f(0) = `12`

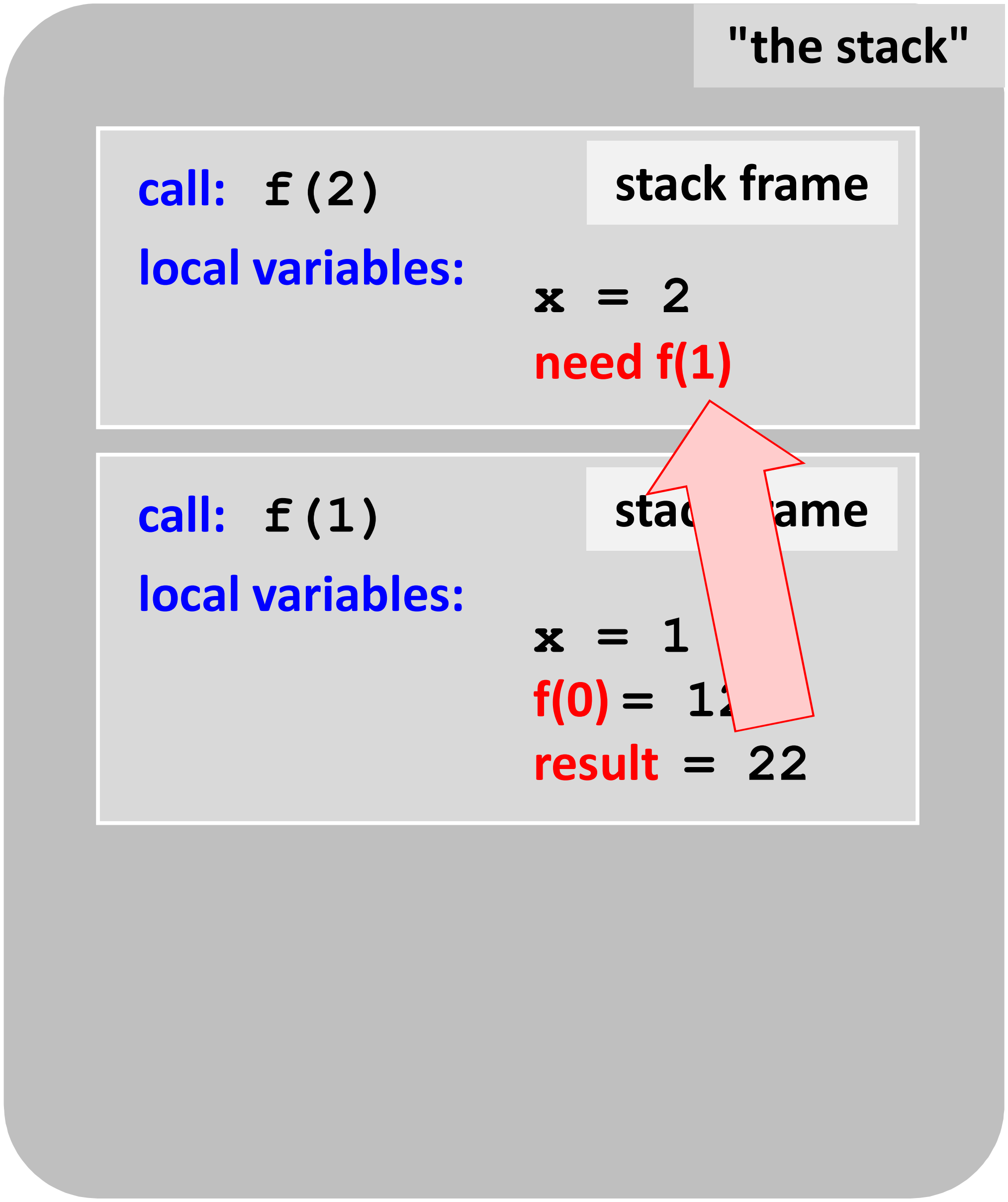
result = `22`

Where does that result go?

How functions work...

1

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



"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

need f(1)

stac ame

call: `f(1)`

local variables:

`x = 1`

f(0) = 12

result = 22

How functions work...

2

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

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

f(1) = `22`

result =

What's *this* return value?

How functions work...

2

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

"the stack"

stack frame

call: `f(2)`

local variables:

`x = 2`

f(1) = `22`

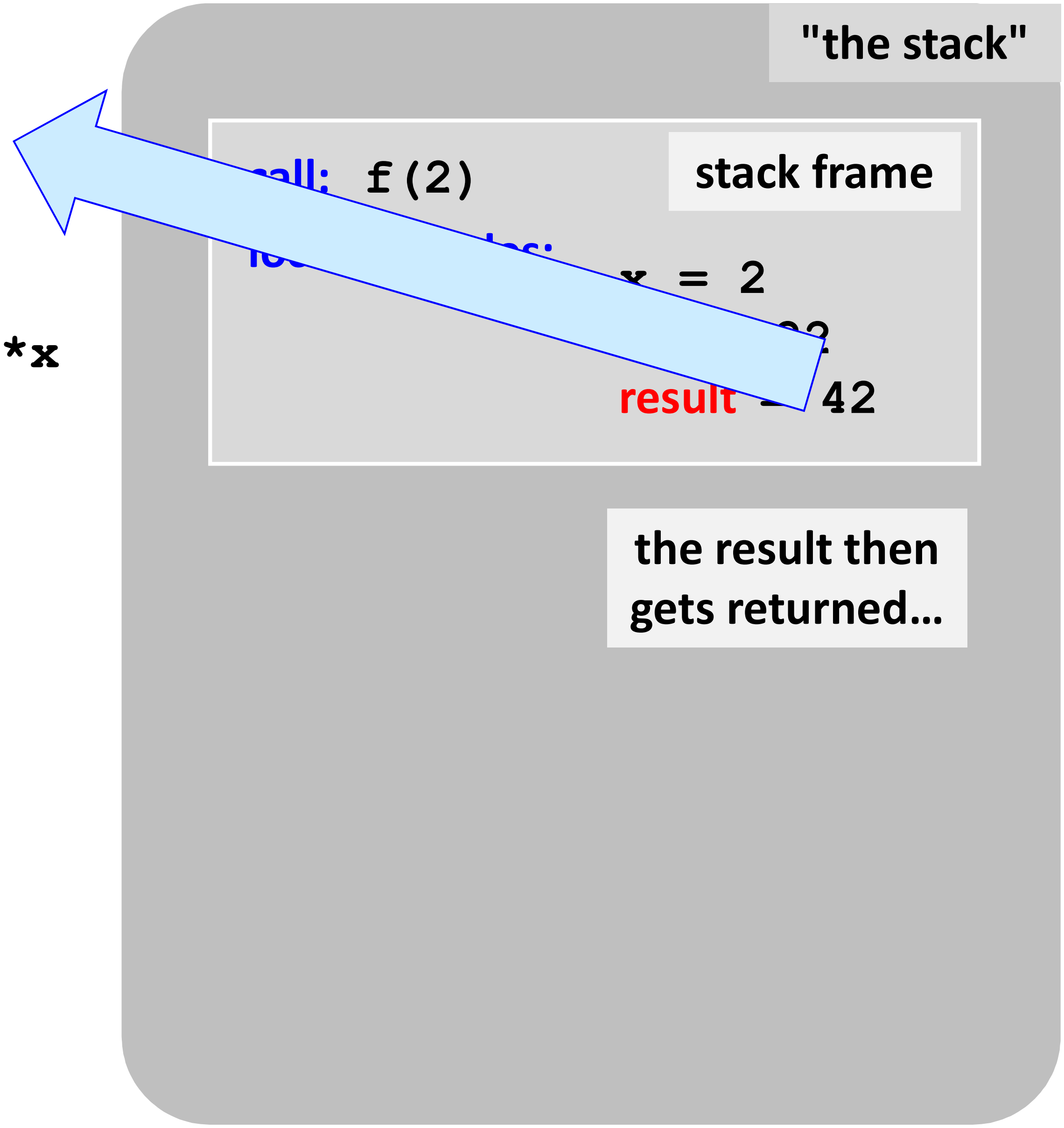
result = `42`

which then
gets returned...

How functions work...

2

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



"the stack"

stack frame

call: `f(2)`

locals:

`x = 2`

[illegible]

result = `42`

the result then gets returned...

How functions work...

2

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

42

output

"the stack"

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

functions ***stack.***

How functions work...

2

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

42

output

"the stack"

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!*

functions ***stack.***

How functions work...

2

```
def f(x):
  if x == 0:
```

42

"the stack"

Pass these eastward!

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

*functions **stack.***

sequential

iteration

self-similar

recursion

problem-solving *paradigms*

Thinking ***sequentially***

factorial

math `5! = 120`

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

`fac(N) = N*(N-1)* … *3*2*1`

Thinking ***sequentially***

factorial

math `5! = 120`

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

October + beyond...

`fac(N) = N*(N-1)* … *3*2*1`

Thinking ***recursively***

factorial

math `5! = 120`

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

cs `fac(5) =`

can we express **`fac`** *w/ a smaller version of itself?*

`fac(N) = N*(N-1)* … *3*2*1`

`fac(N) =`

Thinkin ly

Recursion ~ self-similarity

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

```
fac(5) = 5 * fac(4)
```

can we express **`fac`** *w/ a smaller version of itself?*

```
fac(N) = N*(N-1)* … *3*2*1
```

```
fac(N) = N * fac(N-1)
```

We're done!?

Warning: ***this is legal!***

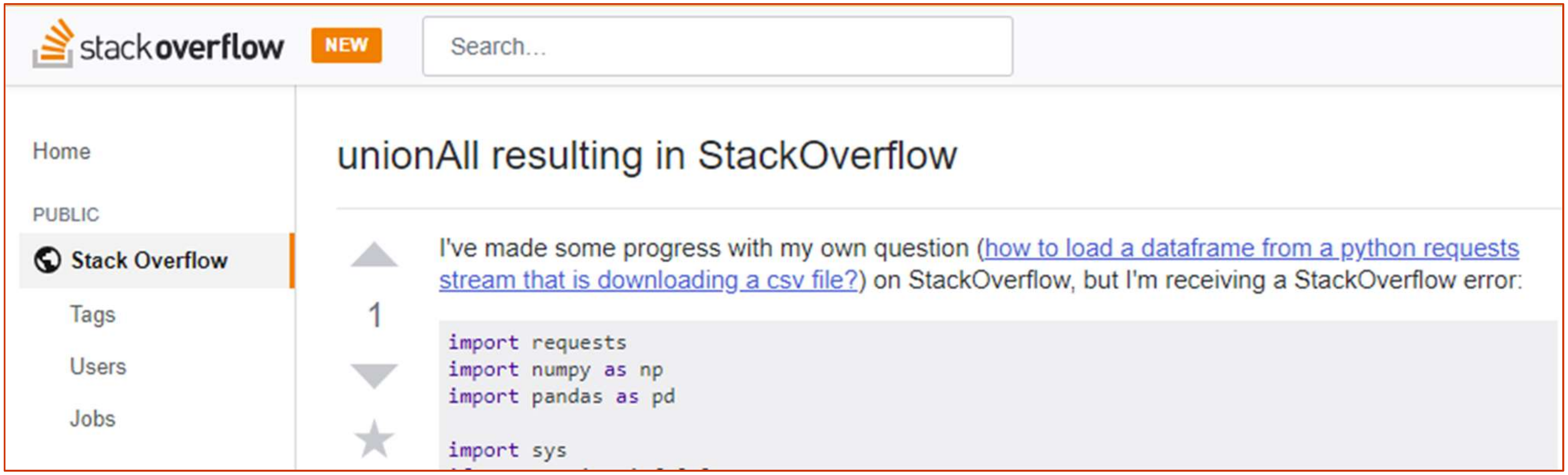
```
def fac(N):
    return N * fac(N-1)
```

I wonder how this code
will **STACK** up!?

```
def facBad(N):
    return N * facBad(N-1)
```

stack **overflow**

stack**overflow**



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:

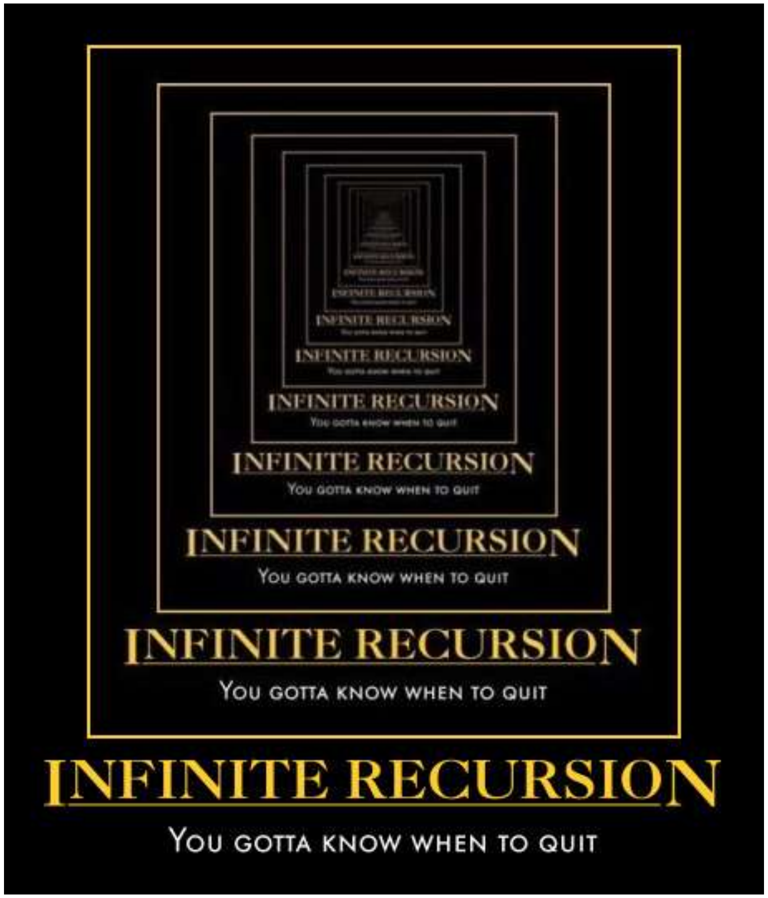
```
import requests
import numpy as np
import pandas as pd

import sys
```

Recursion

the dizzying dangers of having no **base case**!

This "works" ~ *but doesn't work!*

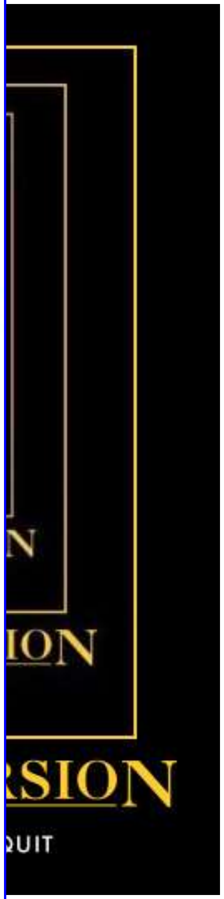
```
def fac(N):
    return fac(N)
```



n't work!

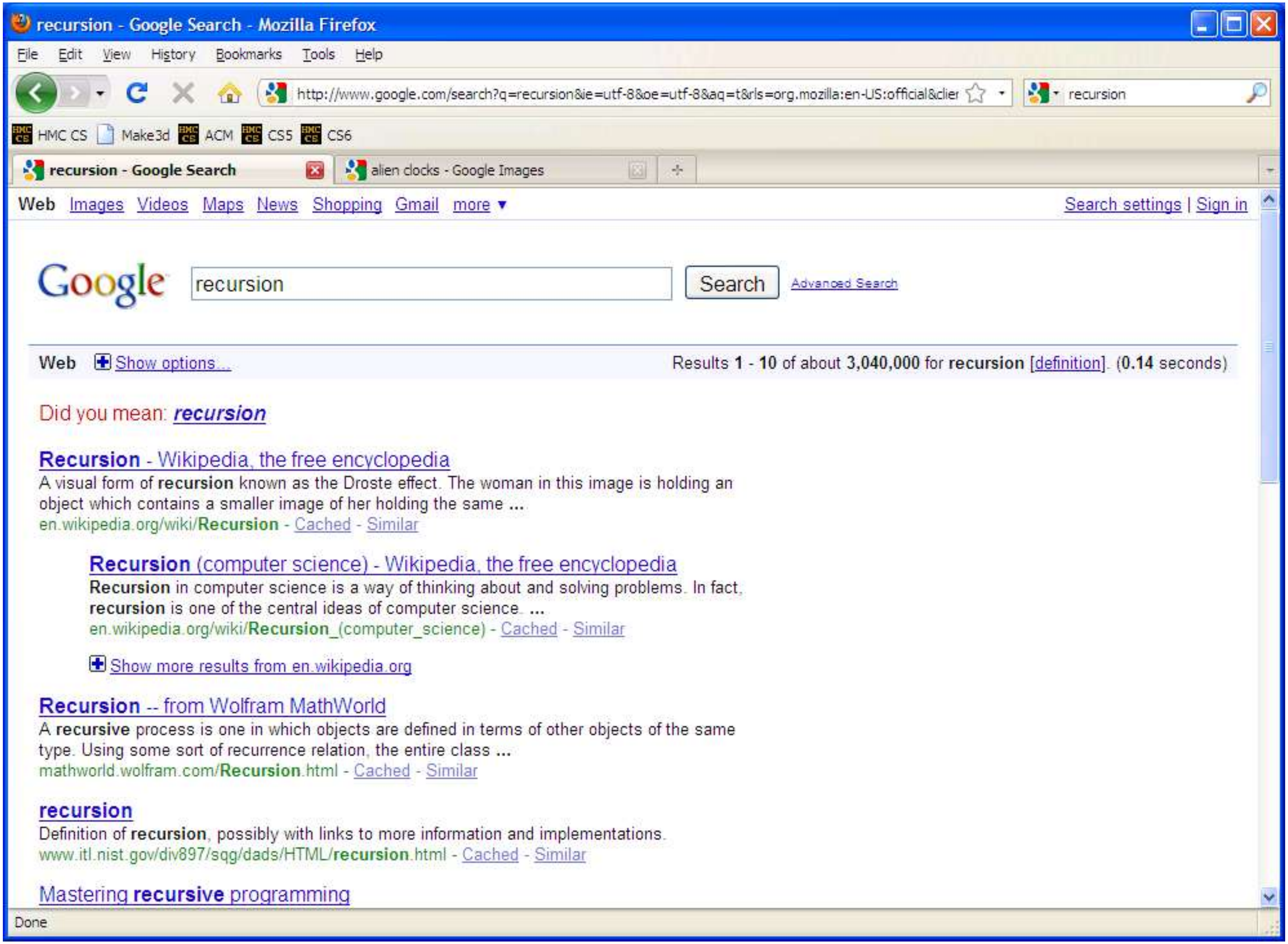
`c (N)`

OH MY GOD 😂😂



R

the diz
havin



http://www.google.com/search?q=recursion&ie=utf-8&oe=utf-8&aq=t&rls=org.mozilla:en-US:official&clier

Search box: recursion

Google recursion

Results **1** - **10** of about **3,040,000** for **recursion** [definition]. (**0.14** seconds)

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

Show more results from en.wikipedia.org

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

legal != recommended

```
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"*

How about an escape
from recursion itself!?!

Thinking recursively...

```
def fac(N):

    if N == 0:
        return 1

    else:
        return N * fac(N-1)
                   ↑
```

Base case

Recursive case

(*too* short?)

Thinking recursively...

```
def fac(N):

    if N == 0:
        return 1

    else:
        return N * fac(N-1)
```

Base case

Recursive case

(*too* short?)

How can this multiply N by something that hasn't happened yet!?!!

Acting recursively

```
def fac(N):

    if N <= 1:
        return 1

    else:

        return N*fac(N-1)
```

this recursion happens first!

Conceptual

```
def fac(N):

    if N <= 1:
        return 1

    else:
        rest = fac(N-1)
        return N*rest
```

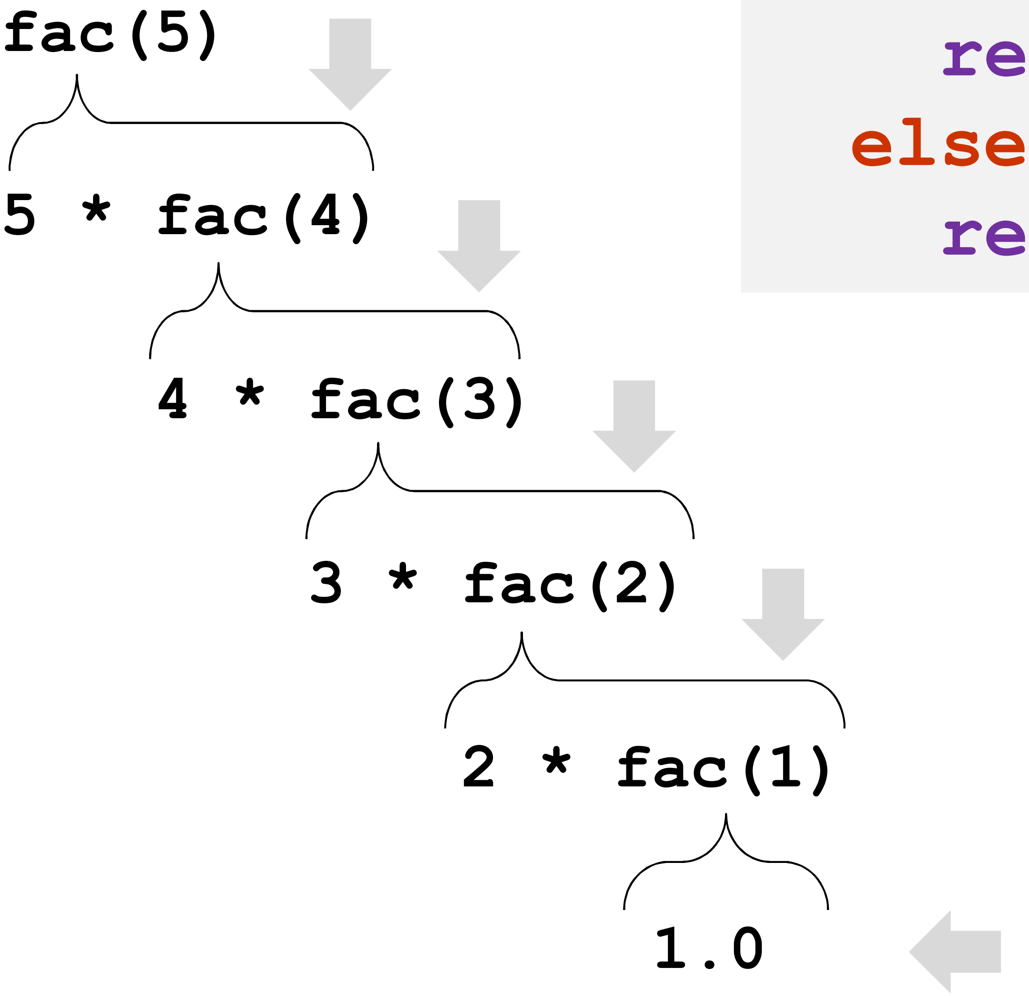
hooray for variables!

Actual

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 * fac(3)

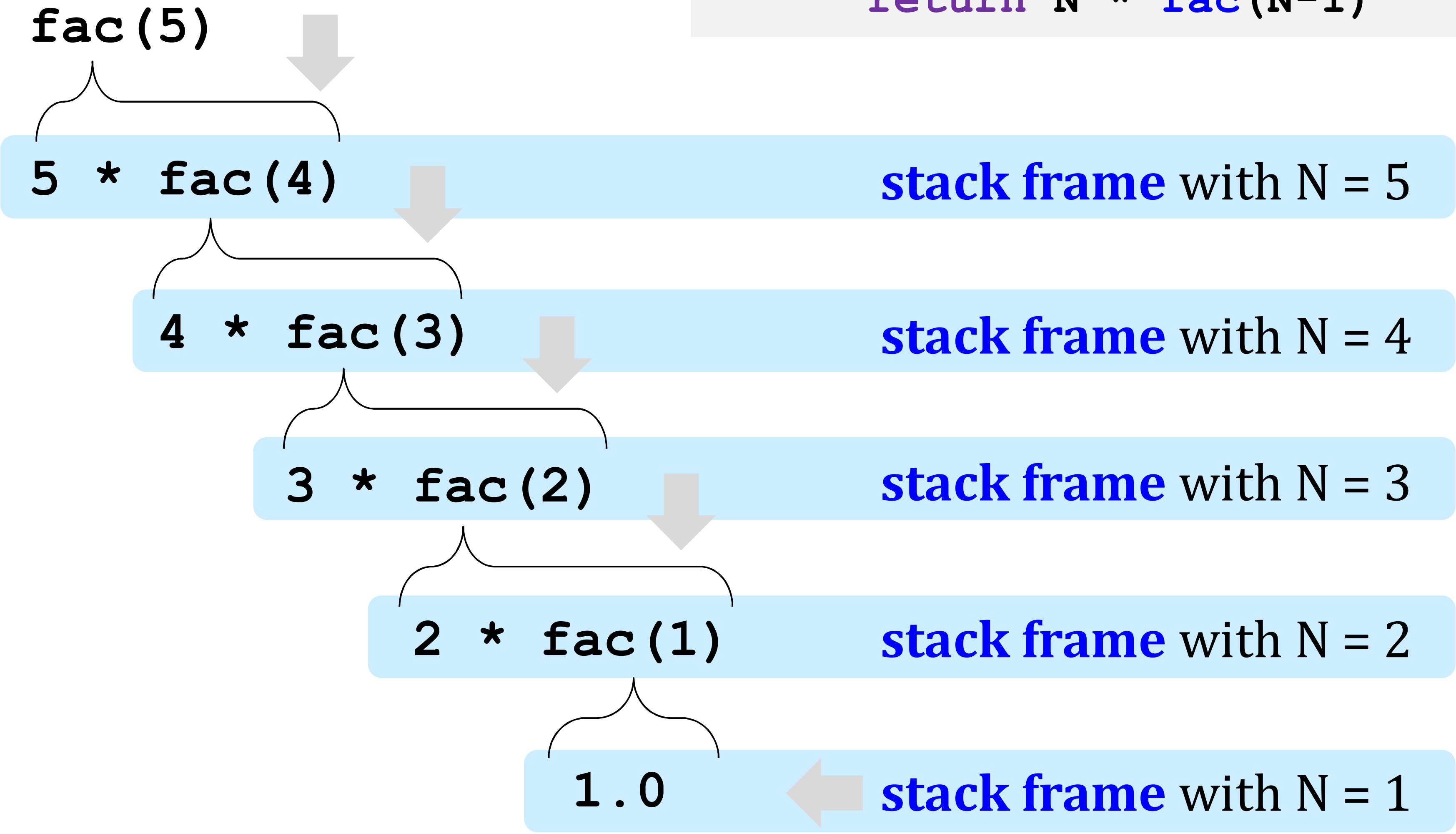
3 * fac(2)

2 * fac(1)

1.0

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

`3 * fac(2)` **stack frame** with N = 3

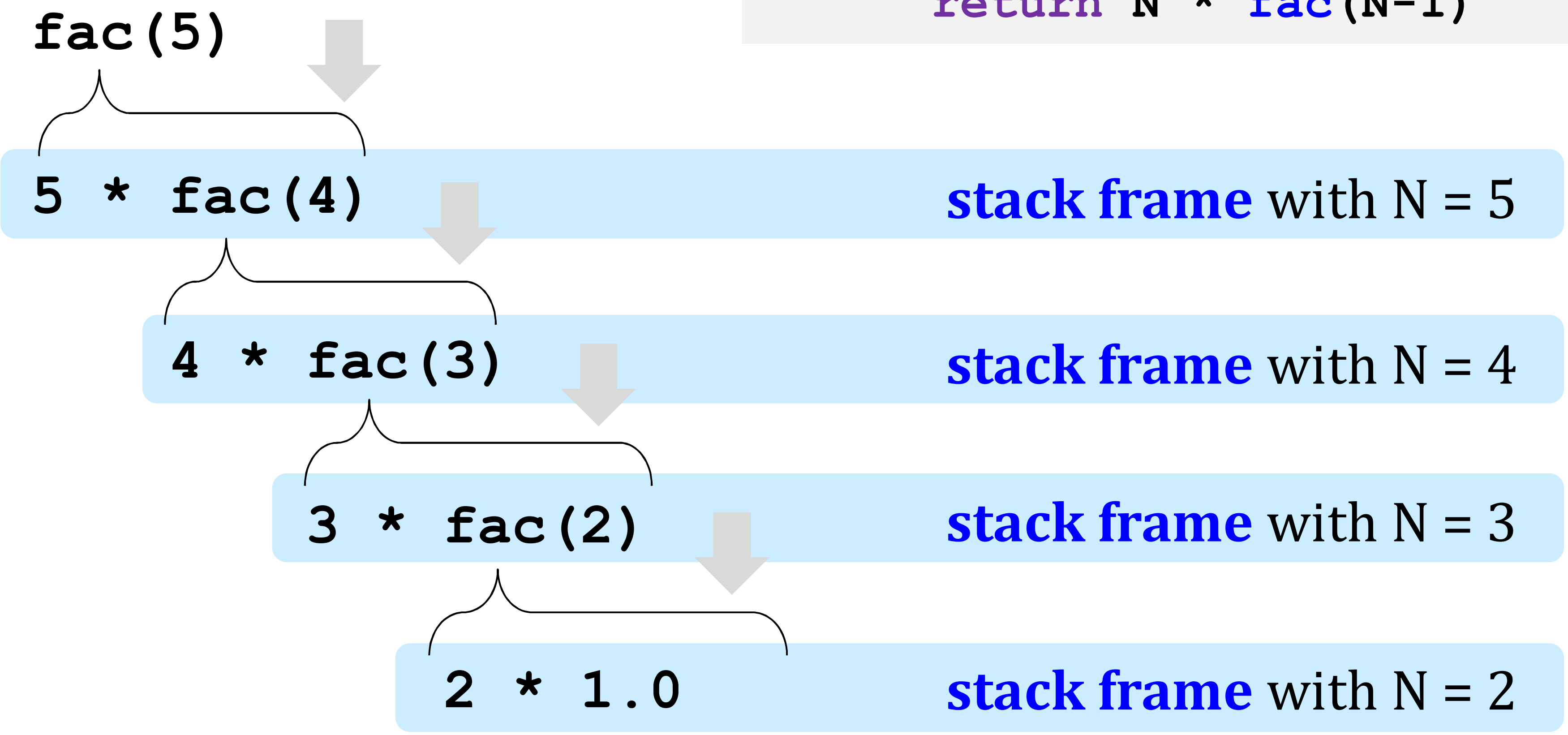
`2 * fac(1)` **stack frame** with N = 2

`1.0` **stack frame** with N = 1

Behind the curtain:

how recursion works...

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



`fac(5)`

`5 * fac(4)` **stack frame** with N = 5

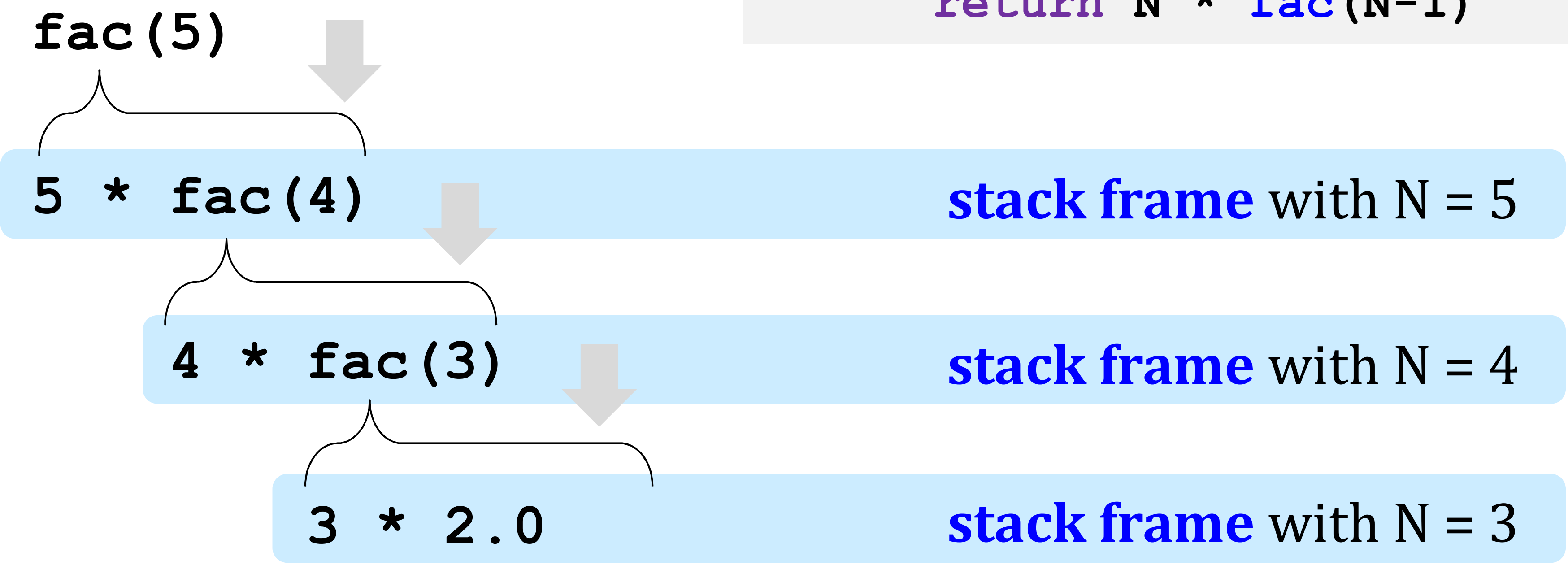
`4 * fac(3)` **stack frame** with N = 4

`3 * fac(2)` **stack frame** with N = 3

`2 * 1.0` **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)`

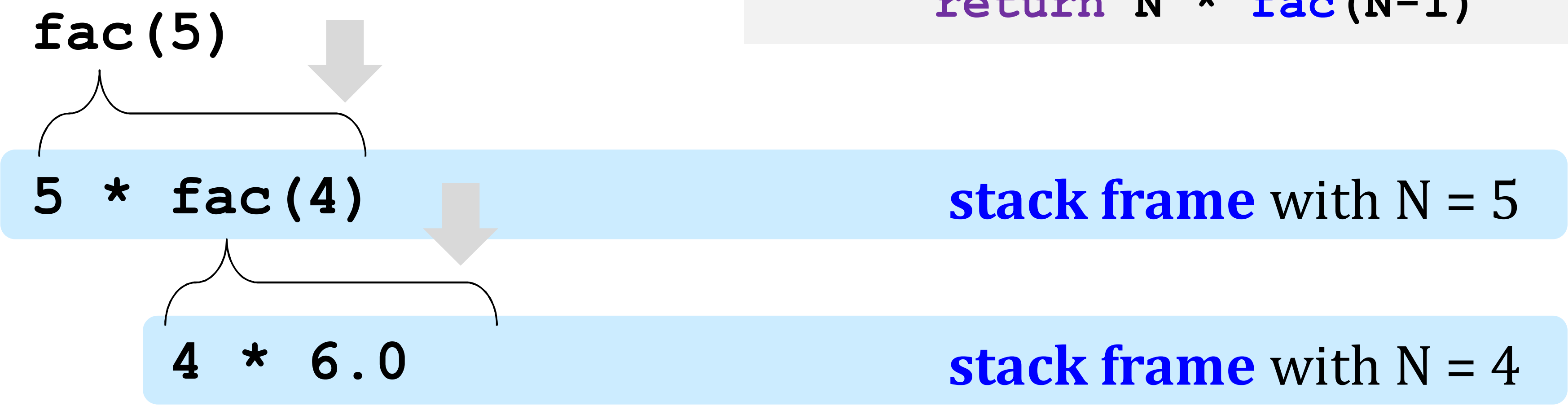
`5 * fac(4)` **stack frame** with N = 5

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

`3 * 2.0` **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)`

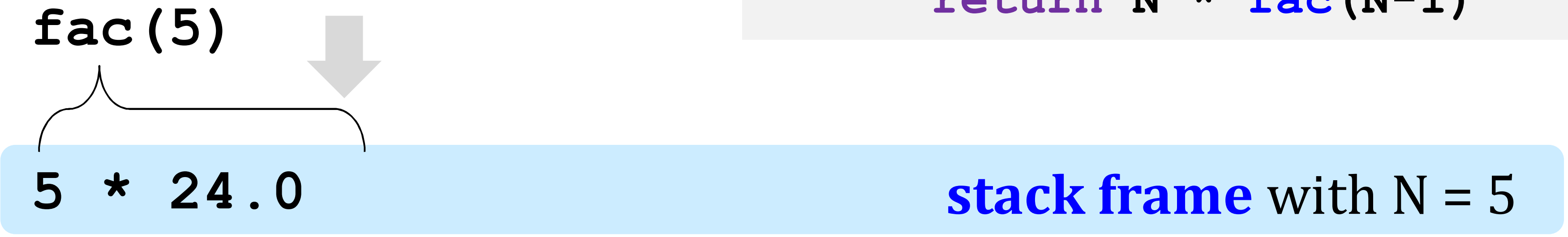
stack frame with N = 5

`4 * 6.0`

stack frame with N = 4

Behind the curtain:
how recursion works...

```
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)
```

$$\underbrace{\texttt{fac(5)}}_{\texttt{120.0}}$$

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* ➡

fac(x)

factorial of x

fac(5)

value of
5*4*3*2*1

is

value of 5 * value of
4*2*3*1

fac(4)

Base case:

fac(0) should return 1

```
def fac(x):
    """ factorial! Recursively!
    """
    if x == 0:
        return 1


    else:
        return x*fac(x-1)
```

`plusone(n)`

adds 1 a total of n times

`plusone(5)`

value of

1+1+1+1+1

is

value of **1** +

Base case:

plusone(0) should return ___

`plusone(n)`

adds 1 a total of n times

`plusone(5)`

value of

1+1+1+1+1

is

value of 1

+

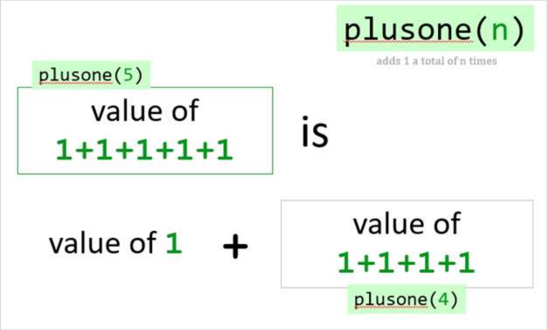
value of

1+1+1+1

`plusone(4)`

Base case:

plusone(0) should return ___



plusone(n)

adds 1 a total of n times

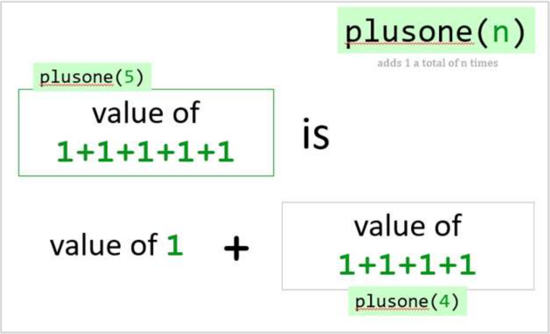
plusone(5)

value of 1+1+1+1+1 is

value of 1 + value of 1+1+1+1

plusone(4)

```
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)
```

`pow(b,p)`

b to the p'th power

`pow(2,5)`

value of

2*2*2*2*2

is

value of **2** *

Base case:

pow(2,0) should return __ ?

`pow(b,p)`

b to the p'th power

`pow(2,5)`

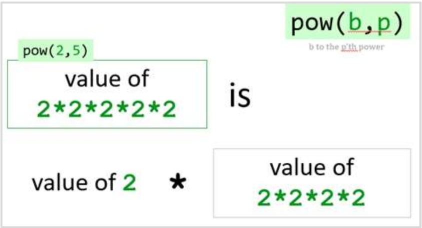
value of
`2*2*2*2*2`

is

value of `2` * value of
`2*2*2*2`

Base case:

pow(2,0) should return __ ?



pow(b,p)

b to the p'th power

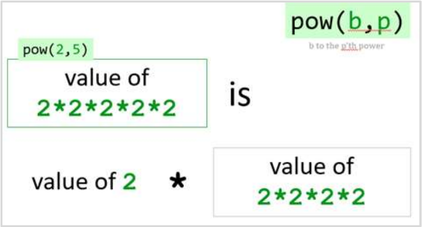
pow(2,5)

value of 2*2*2*2*2 is

value of 2 * value of 2*2*2*2

```
def pow(b,p):
    """
    b**p, defined recursively!
    """
    if p == 0:
        return ____________



    else:
        return ____________________
```

Extra! Can we also handle ***negative*** powers... ?


pow(b,p)

b to the p'th power

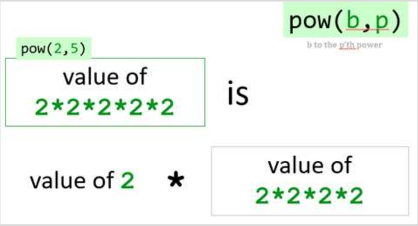
pow(2,5)

value of 2*2*2*2*2 is

value of 2 * value of 2*2*2*2

```
def pow(b,p):
    """
    b**p, defined recursively!
    """
    if p == 0:
        return 1.0
    elif  p < 0:



    else:
        return b*pow(b,p-1)
```

Extra! Can we also handle ***negative*** powers... ?



pow(b,p)

b to the p'th power

pow(2,5)

value of 2*2*2*2*2

is

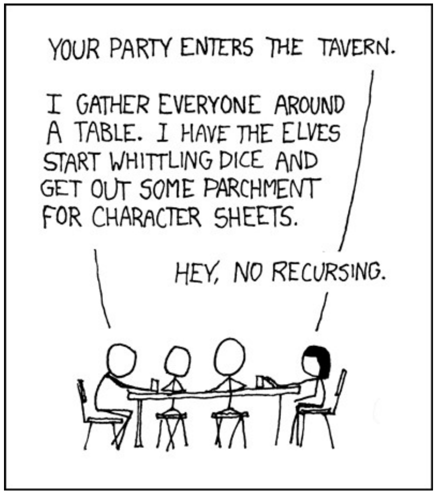
value of 2 * value of 2*2*2*2

```
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... ?

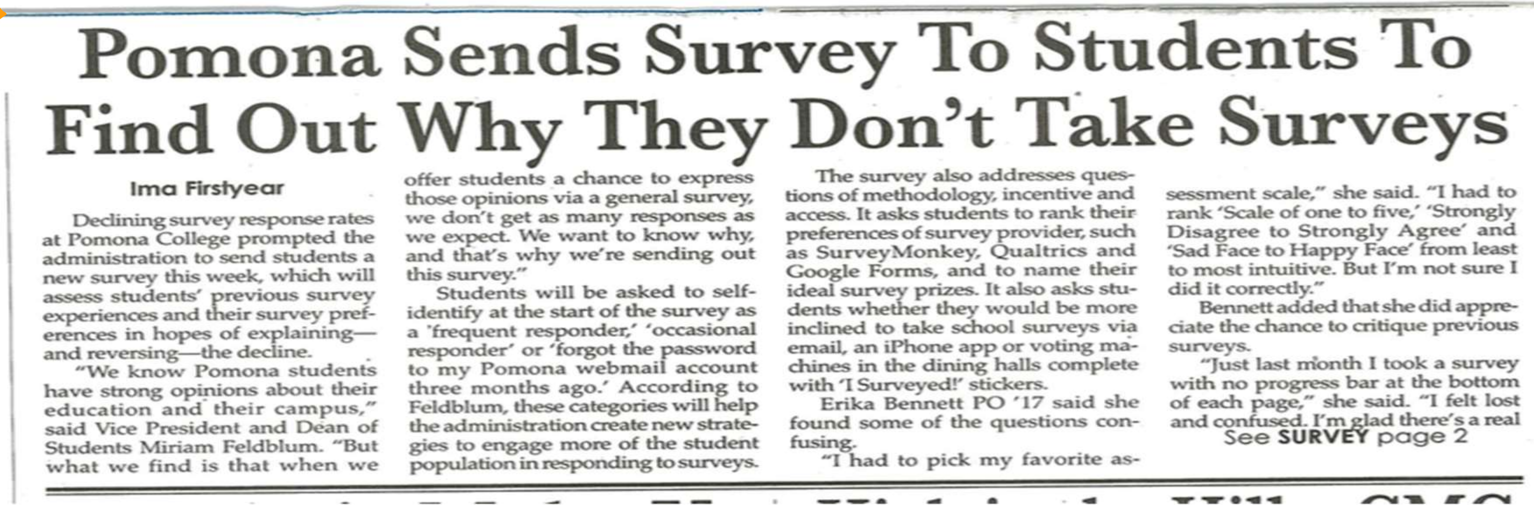
Recursion's advantage:

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



https://www.youtube.com/watch?v=ybX9nVLtNi4 @ 0:08
https://www.youtube.com/watch?v=8PhiSSnaUKk @ 1:11

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***!

As a

Justin Timberlake | Jimmy Fallon | Ultimate Inception | Mug
$15.35 - Etsy
No tax

Jimmy Fallon & Justin Timberlake Funny Coffee Mug. Ultimate Inception Coffee Mug. Great ...
$11.99 - Etsy
No tax

ts To
urveys

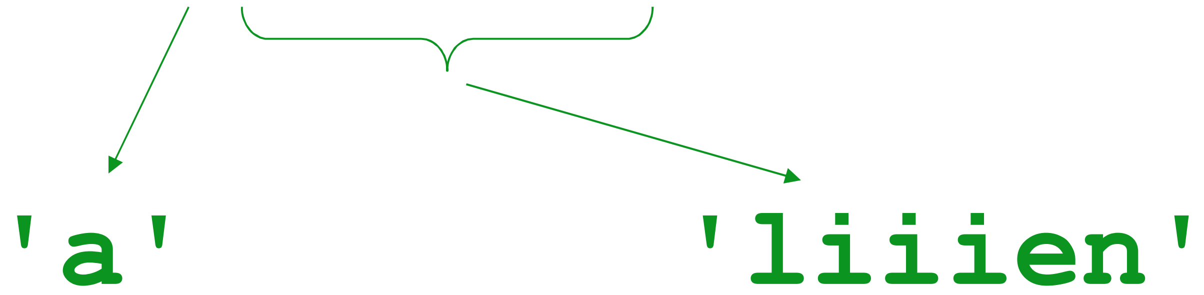
le," she said. "I had to
f one to five,' 'Strongly
Strongly Agree' and
Happy Face' from least
tive. But I'm not sure I
y."
ided that she did appre-
nce to critique previous

month I took a survey
gress bar at the bottom
," she said. "I felt lost
l. I'm glad there's a real
URVEY page 2

Design patterns...

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

```
s = 'aliiien'
```



`'a'`

`'liiien'`

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 = 'aliiien'`

`'a'`

`'liiien'`

`s[0]`

*handle the **first***

Human!

`s[1:]`

*recurse the **rest***

Machine!

Design patterns...

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

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

`3`

`[1,4,1,5,9]`

`L[0]`

*handle the **first***

Human!

`L[1:]`

*recurse the **rest***

Machine!

Design patterns...

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

- *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* **`if`** ...

`numis(s)`

# of i's in s

`numis('xlii')`

# of i's in **`'xlii'`**

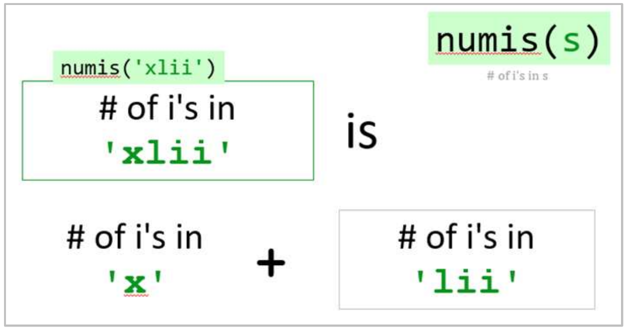
is

# 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                :
        return ________________

    else:
        return ________________
```



numis('xlii')

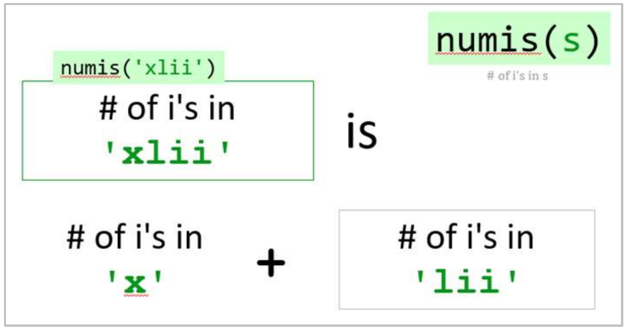
# of i's in 'xlii'

is

numis(s)

# of i's in s

# of i's in 'x' + # of i's in 'lii'



numis(s)

# of i's in s

numis('xlii')

# of i's in 'xlii' is

# 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?

`len(s)`

length of s

`len('yaycs')`

# of chars in `'yaycs'`

is

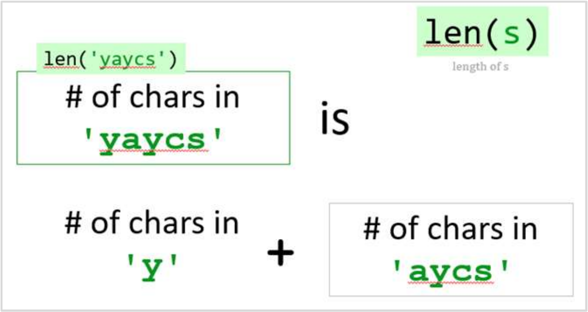
# of chars in `'y'`

+

# of chars in `'aycs'`

Base case:

len('') should return ___ ?



len('yaycs')

# of chars in 'yaycs' is

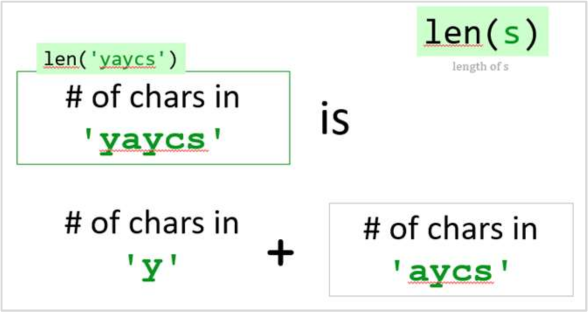
len(s)

length of s

# of chars in 'y' + # of chars in 'aycs'

```
def len(s):
    """
    returns the length of s
    """
    if s == '':
        return ____________


    else:
        return _______________________
```

Extra! Can we also handle ***LISTS***... ?



len(s)

length of s

len('yaycs')

# of chars in 'yaycs'

is

# of chars in 'y' + # of chars in 'aycs'

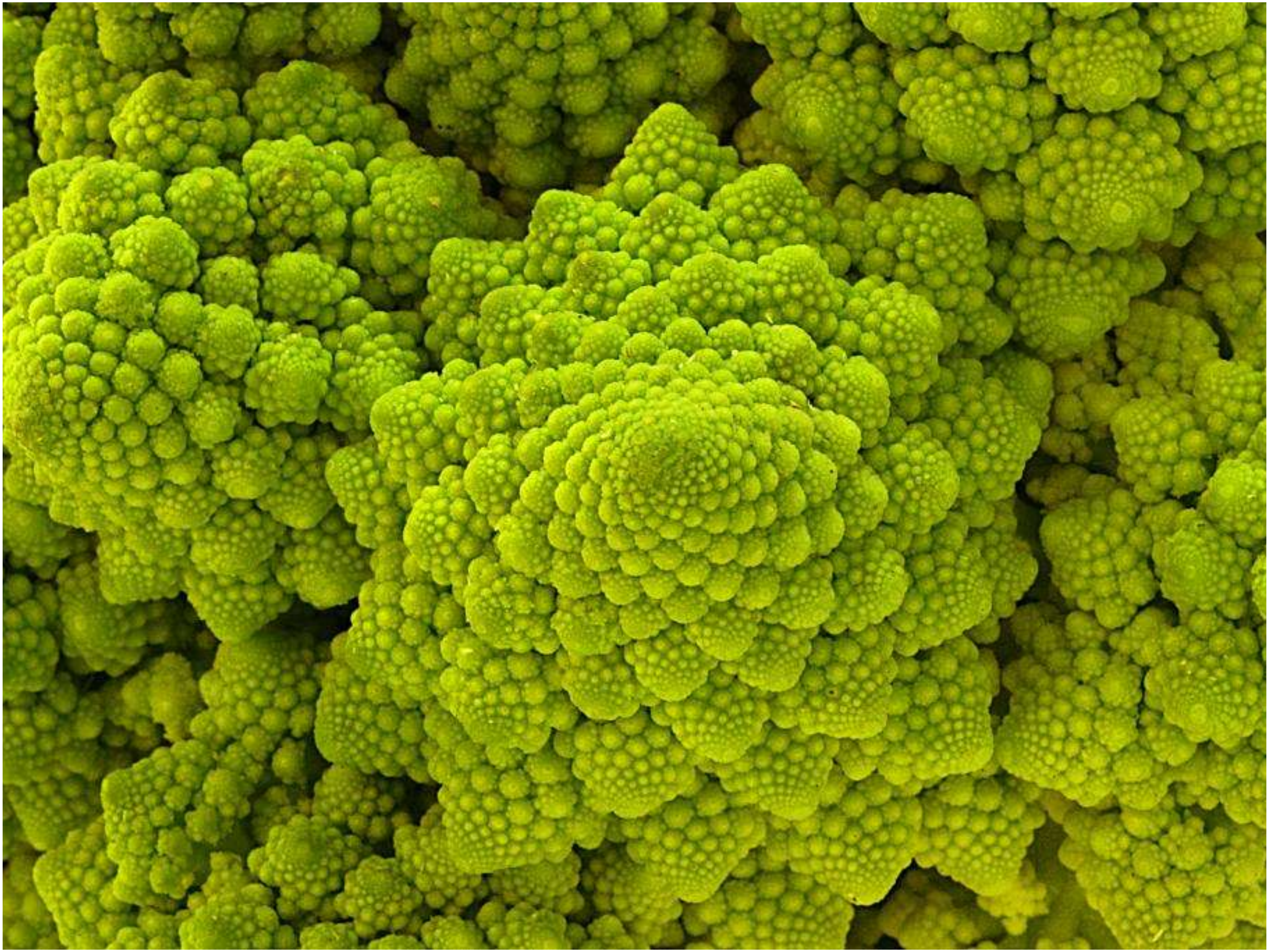
```
def len(s):
    """
    returns the length of s
    """
    if s == '' or s == []:
        return 0



    else:
        return 1 + len(s[1:])
```

one, plus... ... the length of the *rest* of s

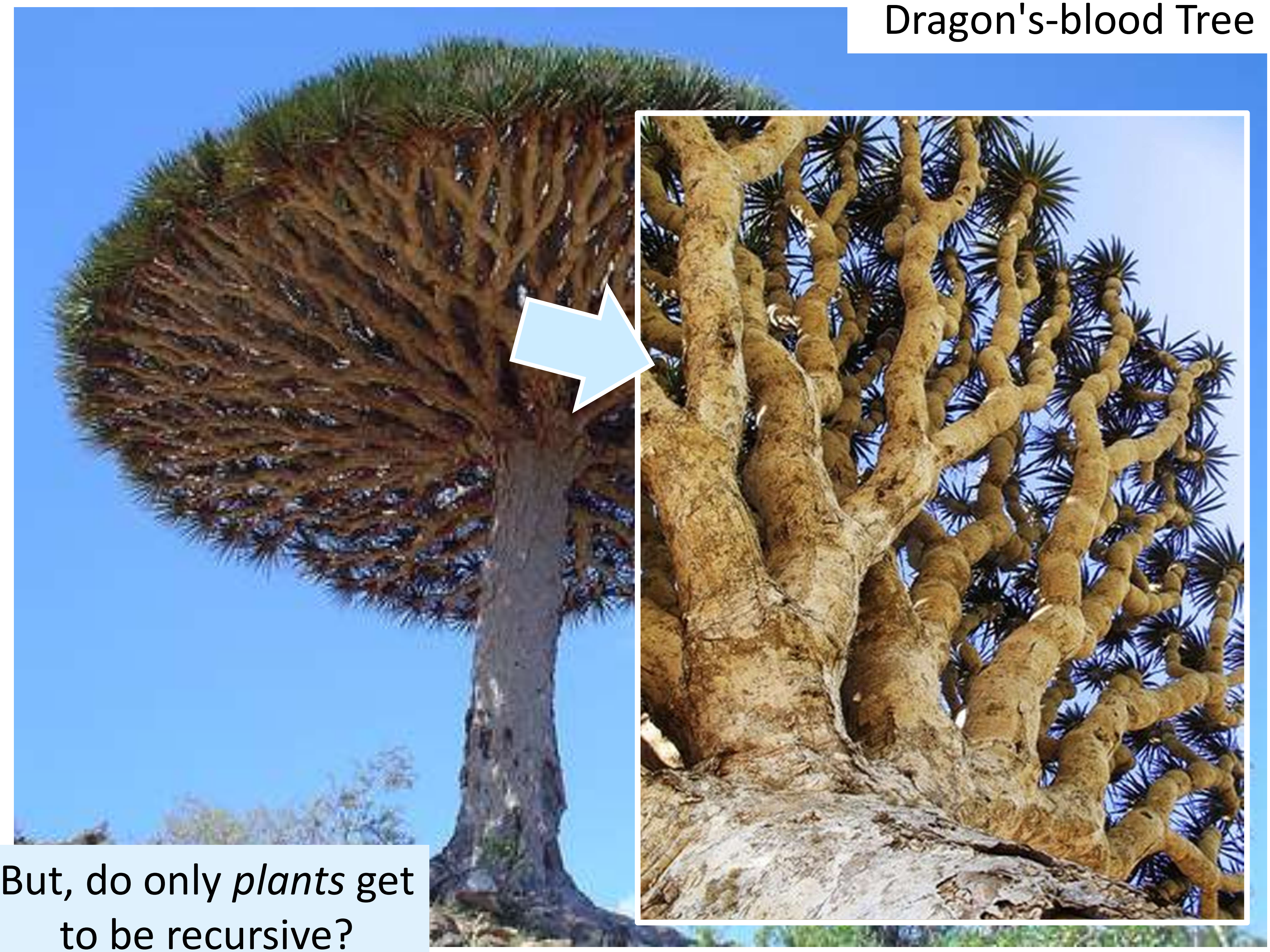
A brief word from our sponsor, Mother Nature...

Like broccoli, recursion is "Good for You"™

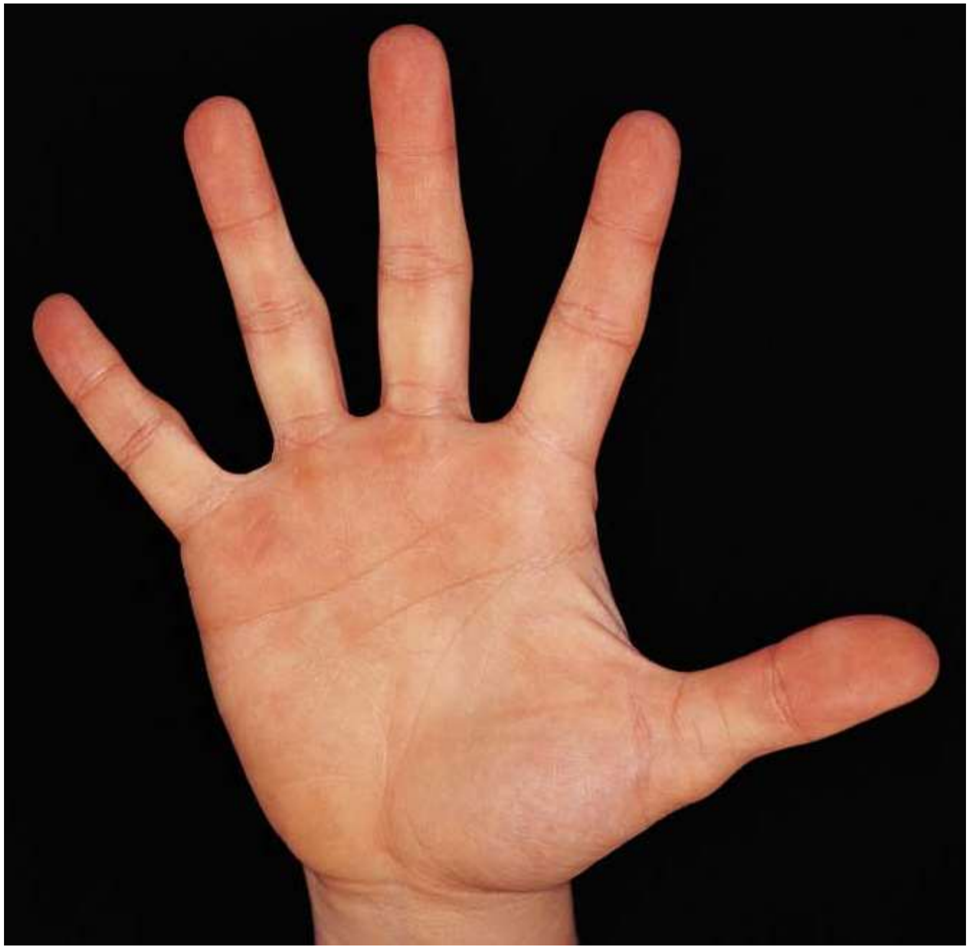
romanesco broccoli



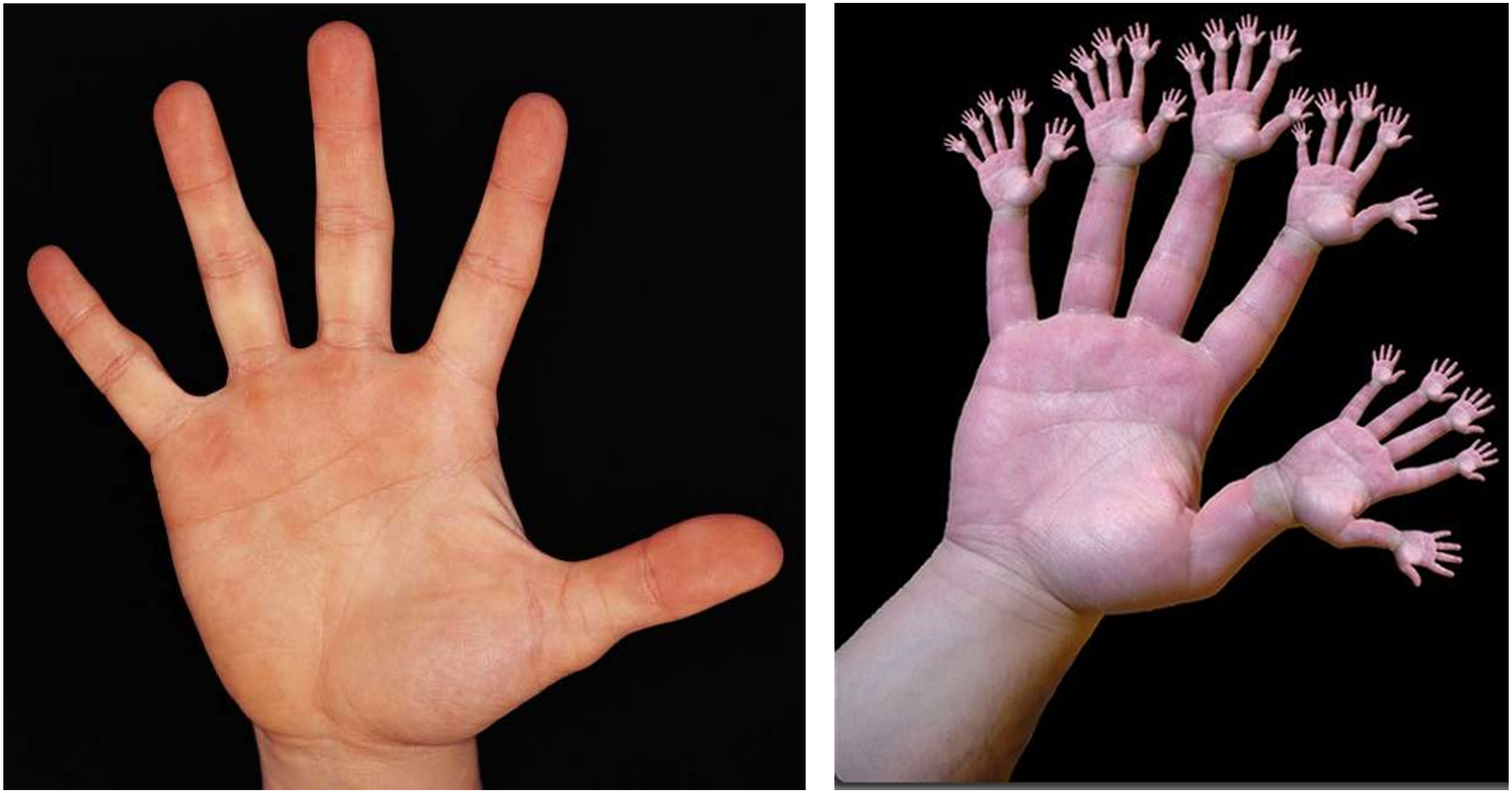
Dragon's-blood Tree



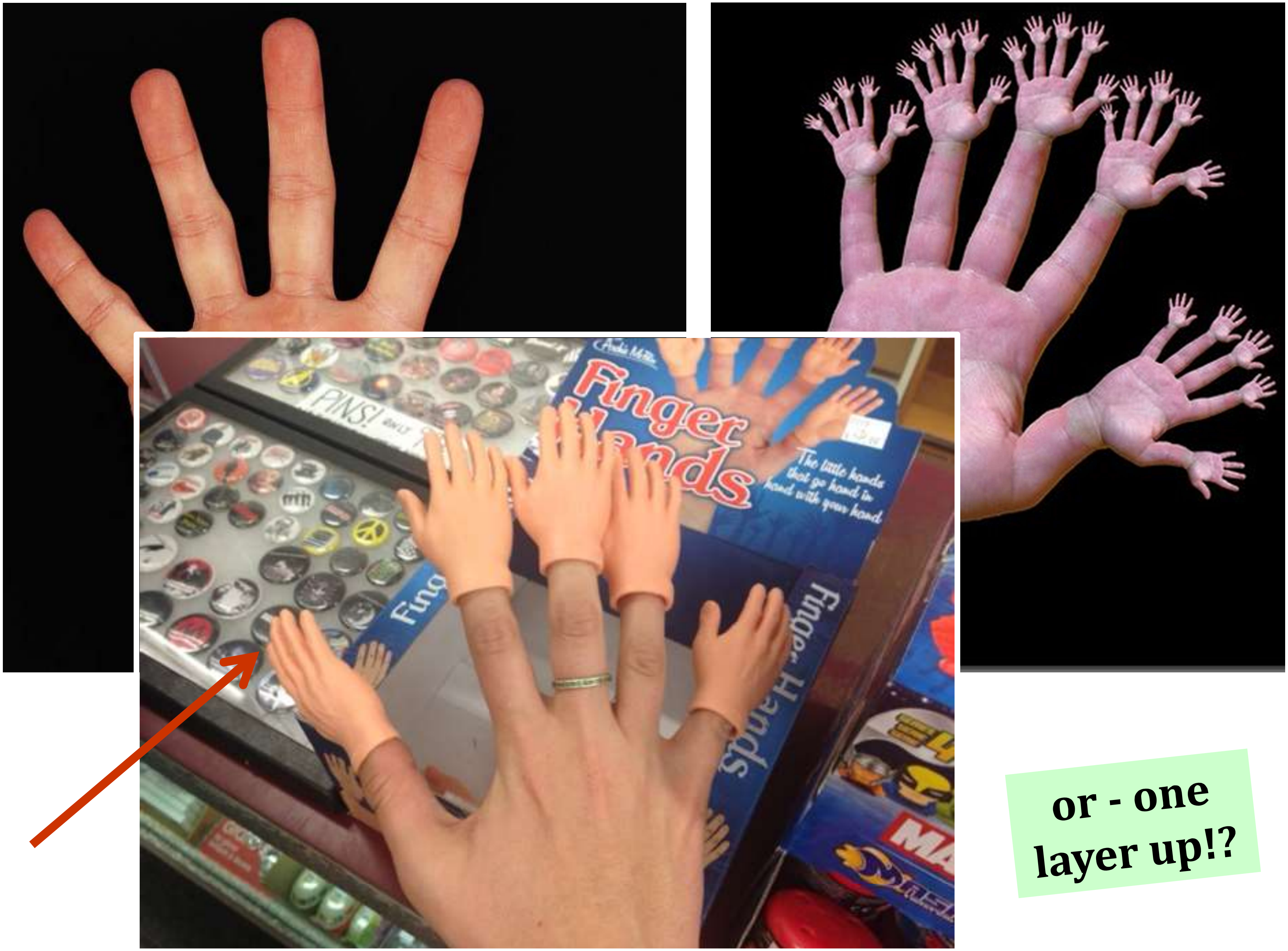
But, do only *plants* get to be recursive?



There still has to be a ***base case***…



or else!



or - one layer up!?

Leap before you look!

Try these four...

Python is... `in`

I guess Python's
the **in** thing

```
>>> '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
```

`vwl(s)`

# of vowels in s

`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 ______________________:
        return
               ____________________

    else:
        return
               ____________________
```

`keepvwl(s)`

keeps only the vowels from s

`keepvwl('pluto')`

keep vowels in `'pluto'`

is

keep vowels in `'p'`

+

keep vowels in `'luto'`

Base case:

keepvwl('') should return ___ ?

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

    elif                       :
        return _________________________

    else:
        return _________________________
```

`max(L)`

L's biggest element

`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 = 

    if            :
        return ______________
    else:
        return ______________
```

← The max of
the *REST* of L

`zeroest(L)`

L's closest-to-zero element

`zeroest([-7,5,9,2])`

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

is

either `-7`

or the zeroest of `[5,9,2]`

Base case:

if len(L) == 1, what should zeroest(L) return ?

```
def zeroest(L):
    """ returns L's element nearest 0
    """
    if len(L) == 1:
        return ____________

    Z =

    if                        :
        return ____________
    else:
        return ____________
```

The zeroest of
the *REST* of L

What seven-letter s
maximizes vwl(s)?

```
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?

```
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.