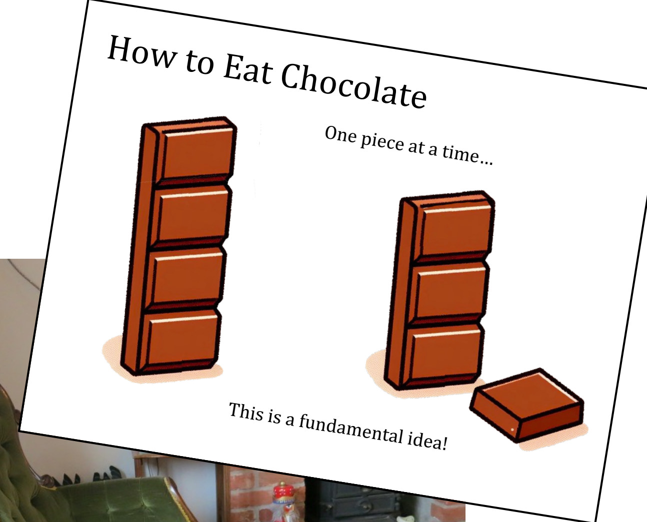
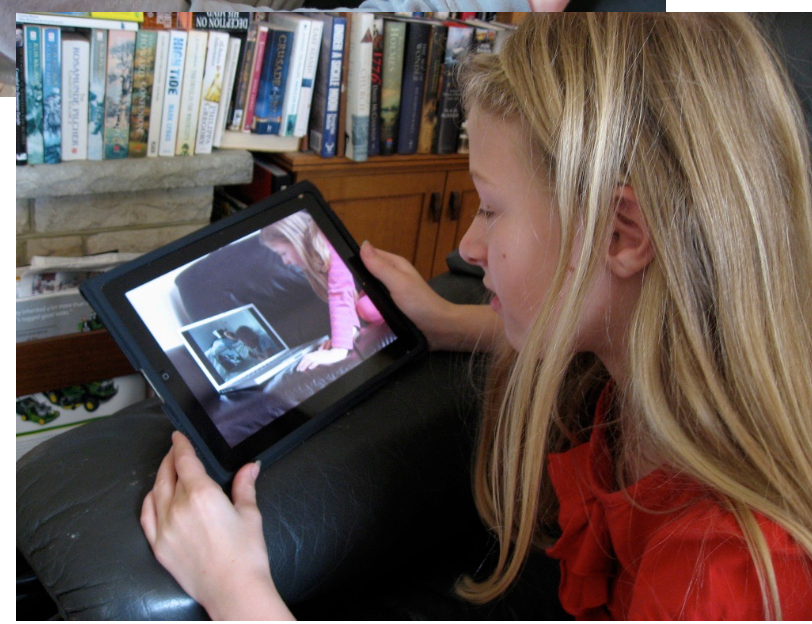
Fun with Functions!





We must go deeper

Last time: *Python slices...*

```
        0 1 2 3 4
s = 'alien'
```

indexing `s[0] ==` 'a' first

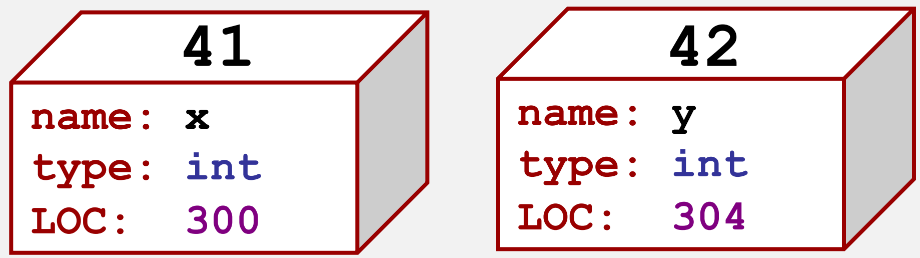
slicing `s[1:]==` 'lien' rest

and?

Computation's Dual Identity

Computation

Data Storage



memory location 300

memory location 304

variables ~ boxes

But what does the
stuff on this side
look like ?

Last time

Data!

This week's reading ***data vs theory***...

Illustration by PM Images/Getty Images.

Are we witnessing the dawn of post-theory science?

Does the advent of machine learning mean the classic methodology of hypothesise, predict and test has had its day?

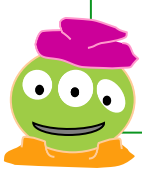
by Laura Spinney

Sun 9 Jan 2022 04.00 EST

927

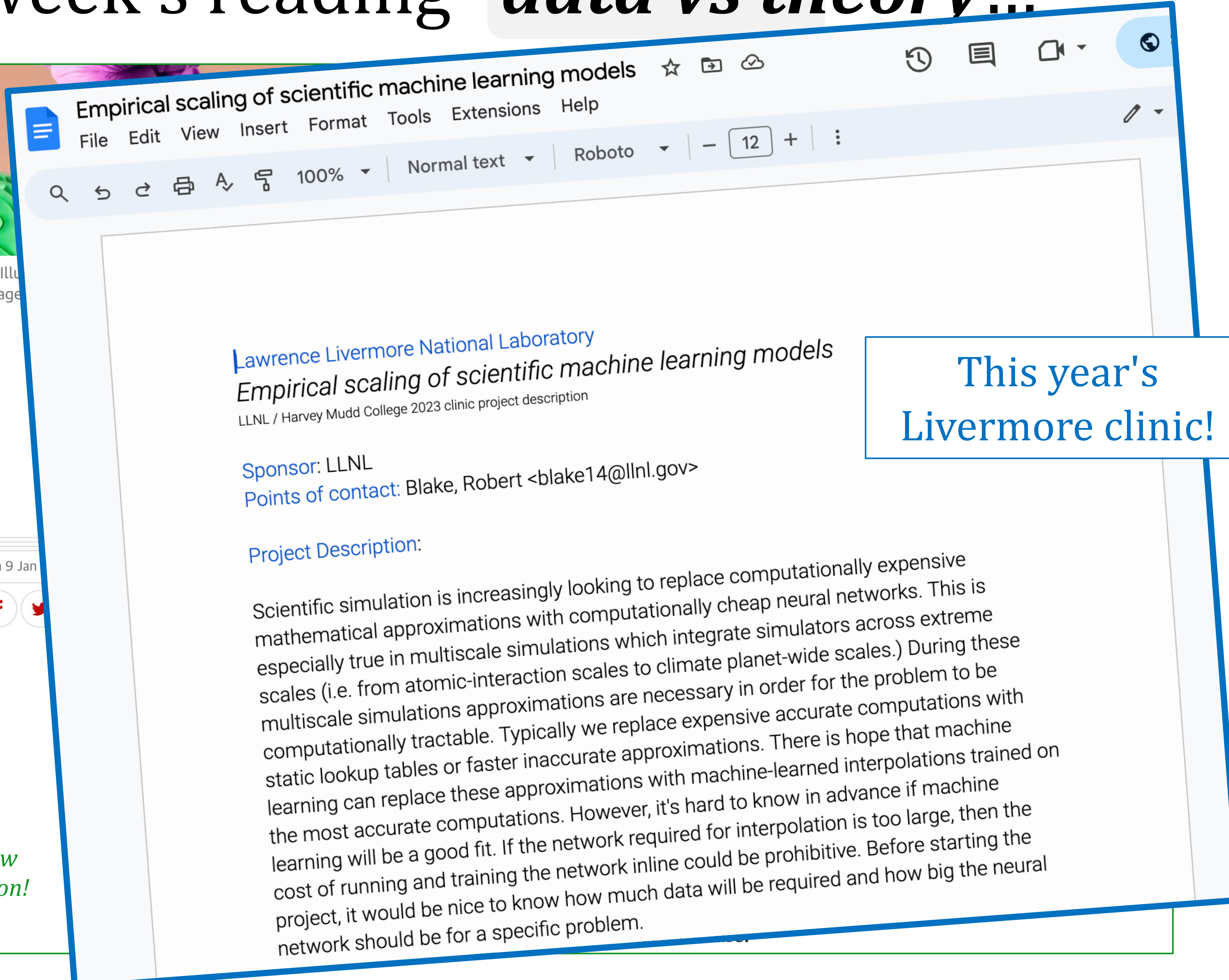
Isaac Newton apocryphally discovered his second law - the one about gravity - after an apple fell on his head. Much experimentation and data analysis later, he realised there was a fundamental relationship between force, mass and acceleration. He formulated a theory to describe that relationship - one that could be expressed as an equation, F=ma - and used it to predict the behaviour of objects other than apples. His predictions turned out to be right (if not always precise enough for those who came later).

Contrast how science is increasingly done today. Facebook's machine learning tools predict your preferences better than any psychologist. AlphaFold, a program built by DeepMind, has produced the most accurate predictions yet of protein structures based on the amino acids they contain. Both are completely silent on why they work: why you prefer this or that information; why this sequence generates that structure.



a New Newton!

This week's reading *data vs theory...*



Lawrence Livermore National Laboratory

Empirical scaling of scientific machine learning models

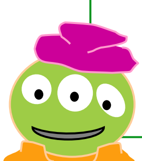
LLNL / Harvey Mudd College 2023 clinic project description

Sponsor: LLNL
Points of contact: Blake, Robert <blake14@llnl.gov>

Project Description:

Scientific simulation is increasingly looking to replace computationally expensive mathematical approximations with computationally cheap neural networks. This is especially true in multiscale simulations which integrate simulators across extreme scales (i.e. from atomic-interaction scales to climate planet-wide scales.) During these multiscale simulations approximations are necessary in order for the problem to be computationally tractable. Typically we replace expensive accurate computations with static lookup tables or faster inaccurate approximations. There is hope that machine learning can replace these approximations with machine-learned interpolations trained on the most accurate computations. However, it's hard to know in advance if machine learning will be a good fit. If the network required for interpolation is too large, then the cost of running and training the network inline could be prohibitive. Before starting the project, it would be nice to know how much data will be required and how big the neural network should be for a specific problem.

This year's Livermore clinic!



a New Newton!

Computation's Dual Identity

accessed through functions...

Computation

Data Storage

41	42
name: x	name: y
type: int	type: int
LOC: 300	LOC: 304

n 300

memory location 304

riables ~ boxes

Functions!

This time

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

Functioning across disciplines

procedure

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

CS's googolizer

defined by ***what it does***

+ what follows ***behaviorally***

structure

$$g(x) = x^{100}$$

Math's googolizer

defined by ***what it relates***

+ what follows ***logically***

Functions!

```
In [2]: verbify('random')
Out[2]: 'randomize'

In [3]: nounify('eat')
Out[3]: 'eater'
```

Functions!

```
In [2]: verbify('random')
Out[2]: 'randomize'

In [3]: nounify('eat')
Out[3]: 'eater'
```

```
def verbify(noun):
    return noun + 'ize'

def nounify(noun):
    return noun + 'er'
```

Functions!

```
In [2]: verbify('random')
Out[2]: 'randomize'

In [3]: nounify('eat')
Out[3]: 'eater'

In [4]: nounify('bake')
Out[4]: 'bakeer'
```

```
def verbify(noun):
    return noun + 'ize'

def nounify(noun):
    return noun + 'er'
```

Functions!

```
In [2]: verbify('random')
Out[2]: 'randomize'

In [3]: nounify('eat')
Out[3]: 'eater'

In [4]: nounify('bake')
Out[4]: 'baker'
```

```python
def verbify(noun):
    return noun + 'ize'

def nounify(verb):
    return stem(verb) + 'er'
```

More Functions!

```
In [2]: verbify('random')
Out[2]: 'randomize'

In [3]: nounify('eat')
Out[3]: 'eater'

In [4]: nounify('bake')
Out[4]: 'baker'
```

```
def stem(word):
    if word[-1] == 'e':
        return word[:-1]
    else:
        return word

def verbify(noun):
    return stem(noun) + 'ize'

def nounify(verb):
    return stem(verb) + 'er'
```

Use variables!

I'm happy
about this, too!

```
def insertOh(s):
    m = len(s)//2
    return s[m:] + 'OH' + s[:m]
```

these two functions
do the "same" thing...

Ok, we humans work better when naming things...

*...why might computers **"prefer"** the top version?!*

```
def insertOh(s):
    return s[len(s)//2:] + 'OH' + s[:len(s)//2]
```

Aargh!

More Functions!

```
def convLengthPrint(inches):
    """ convert inches to customary length units
        input: inches, an int
    """
    miles = inches // (8 * 10 * 22 * 3 * 12)  # 8 furlongs per mile
    inches = inches % (8 * 10 * 22 * 3 * 12)
    furlongs = inches // (10 * 22 * 3 * 12)   # 10 chains per furlong
    inches = inches % (10 * 22 * 3 * 12)
    chains = inches // (22 * 3 * 12)          # 22 yards per chain
    inches = inches % (22 * 3 * 12)
    yards = inches // (3 * 12)                # 3 feet per yard
    inches = inches % (3 * 12)
    feet = inches // 12                       # 12 inches per foot
    inches = inches % 12
    print(miles, "miles,", furlongs, "furlongs,", chains, "chains,",
          yards, "yards,", feet, "feet, and", inches, "inches.")
```

What's the difference?

More Functions!

```
def convLength(inches):
    """ convert inches to customary length units
        input: inches, an int
    """
    miles = inches // (8 * 10 * 22 * 3 * 12)  # 8 furlongs per mile
    inches = inches % (8 * 10 * 22 * 3 * 12)
    furlongs = inches // (10 * 22 * 3 * 12)   # 10 chains per furlong
    inches = inches % (10 * 22 * 3 * 12)
    chains = inches // (22 * 3 * 12)          # 22 yards per chain
    inches = inches % (22 * 3 * 12)
    yards = inches // (3 * 12)                # 3 feet per yard
    inches = inches % (3 * 12)
    feet = inches // 12                       # 12 inches per foot
    inches = inches % 12

    return [miles, furlongs, chains, yards, feet, inches]
```

return vs. print

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


ans = dbl(20)
```

```python
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!

`return` conveys the function's ***value***

... which the terminal then prints!

`print` changes only pixels-on-the-screen `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

Terminology

function name

parameter

signature line

```
def convLength(inches):
    """ convert inches to customary length
        input: inches, an int
    """
    miles = inches // (8 * 10 * 22 * 3 * 12)  # 8 furlongs per mile
    inches = inches % (8 * 10 * 22 * 3 * 12)
    furlongs = inches // (10 * 22 * 3 * 12)   # 10 chains per furlong
    inches = inches % (10 * 22 * 3 * 12)
          hes // (22 * 3 * 12)                # 22
         = inches % (22 * 3 * 12)
    yards = inches // (3 * 12)                # 3 feet pe
    inches = inches % (3 * 12)
    feet = inches // 12                       # 12 inches per foot
    inches = inches % 12

    return [miles, furlongs, chains, yards, feet, inches]
```

docstring

code block

in-line comments —optional in CS 5

return statement

follow the data!

```
def undo(s):
    """ this "undoes" its input, s """
    return 'de' + s
```

```
>>> undo('caf')
```

follow the data!

```
def undo(s):
    """ this "undoes" its input, s """
    return 'de' + s
```

```
>>> undo('caf')
'decaf'

>>> undo(undo('caf'))
```

strings, lists, numbers ...
*all **data** are fair game*

follow the data!

```
def undo(s):
    """ this "undoes" its input, s """
    return 'de' + s
```

```
>>> undo('caf')
'decaf'

>>> undo(undo('caf'))
'dedecaf'
```

strings, lists, numbers ...
*all **data** are fair game*

Big Ideas

- We can write functions
  - Those functions can make decisions
- We can call functions
- We can write functions that call functions we've written and use their results
- Variables in functions belong to the function and vanish when it's done!

Names: ____________________

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

Quiz

1

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

15

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

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

2

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

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

I might have a guess...

Extra!

en") here?

```
ou':
    return 1 + vwl(s[1:])

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

Functions!

Names: ______________________________

How f'ns ***work***… *Quiz*

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

15

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

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

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

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

I might have a guess…

Extra!

What is **`vwl("alien")`** here?

```
def vwl(s):

    if s == '':
        return 0

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

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

Names: ______________________________

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

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

15

42

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

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

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

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

42

I might have a guess...

Extra!

What is **`vwl("alien")`** here?

```
def vwl(s):

    if s == '':
        return 0

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

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

3

Python Tutor: Visualize code in Python

Write code in Python 3.6

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

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

result = demo(15)
print("demo(15) is", result)
```

One snapshot...

Python 3.6
([known limitations])

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

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

result = demo(15)
print("demo(15) is", result)
```

[Edit this code]

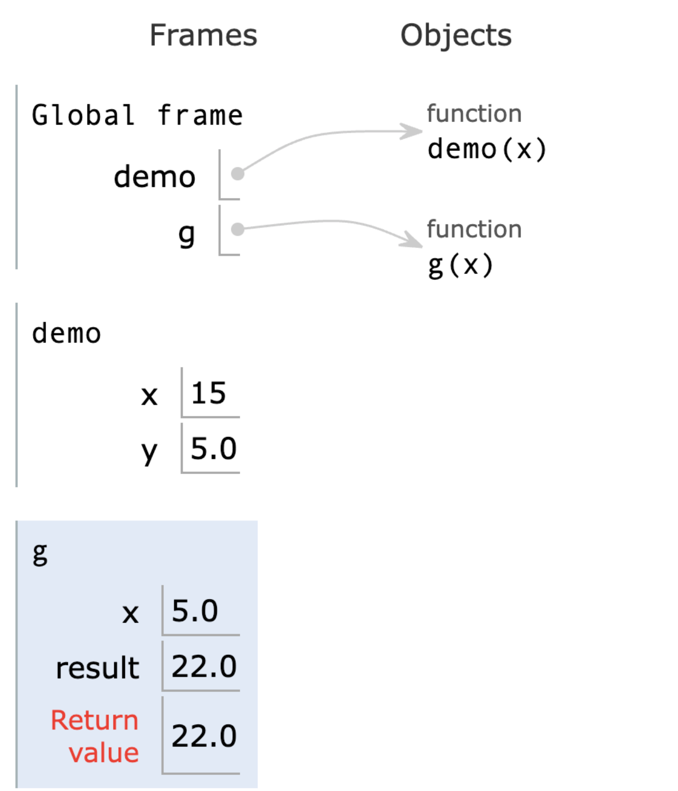
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 10 of 13

[Customize visualization]

Print output (drag lower right corner to resize)



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

```python
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"

stack frame

call: `demo(15)`

local variables:

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

stack frame

call: `g(5)`

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 = ?????
```

stack frame

call: `g(5)`

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"

So many **x**'*es... !*

Python 3.6
(known limitations)

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

result = f(2)
print("f(2) is", result)
```

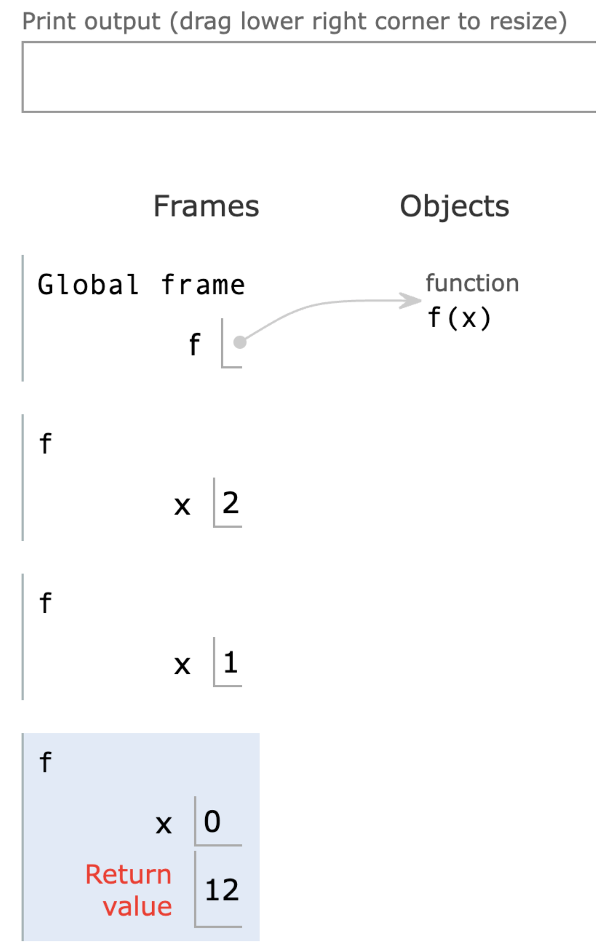
Edit this code

line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 12 of 15

Customize visualization



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:

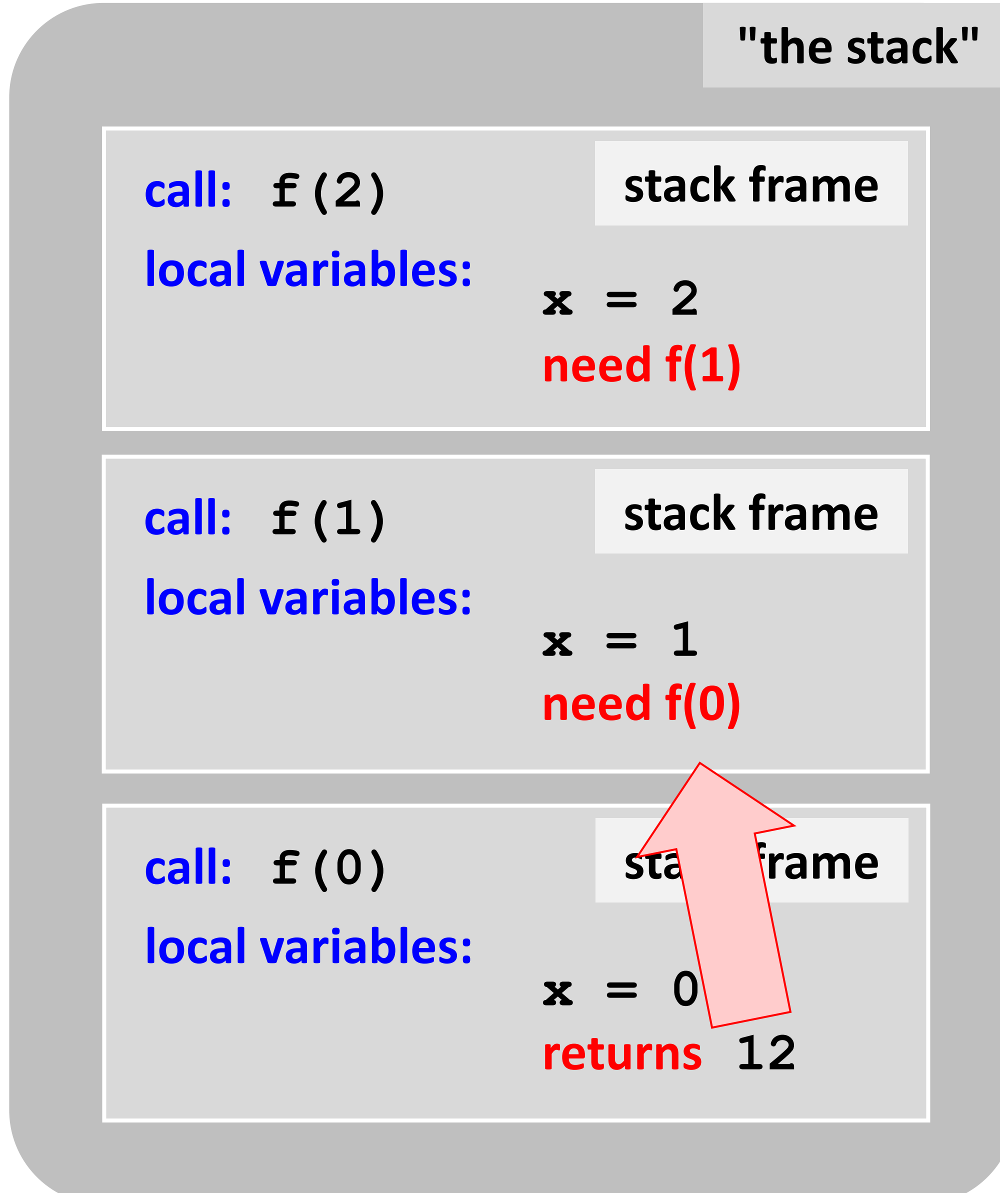
`x = 0`

returns `12`

How functions work...

0

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



How functions work...

1

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

"the stack"

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

call: `f(1)`	stack frame
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"

call: `f(2)`

stack frame

local variables:

`x = 2`

need f(1)

call: `f(1)`

stack frame

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"

call: `f(2)`

stack frame

local variables:

`x = 2`

need f(1)

call: `f(1)`

stac ame

local variables:

`x = 1`

f(0) = `1`

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)

x = 2

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 f'n is a ***self-contained*** *computational unit!*

functions ***stack****.*

How functions work...

2

```
def f(x):
  if x == 0:
    return 12
  else:
```

42

output

"the stack"

again, the stack is empty, but ready if another

Pass those papers north!

... each f'n is a ***self-contained*** *computational unit!*

functions ***stack****.*

Functions' *conceptual* challenge?

You need to see BOTH the ***internal details*** AND the ***world-facing interface*** simultaneously!

cells!!!

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!

Like broccoli, recursion is
"Good for You"

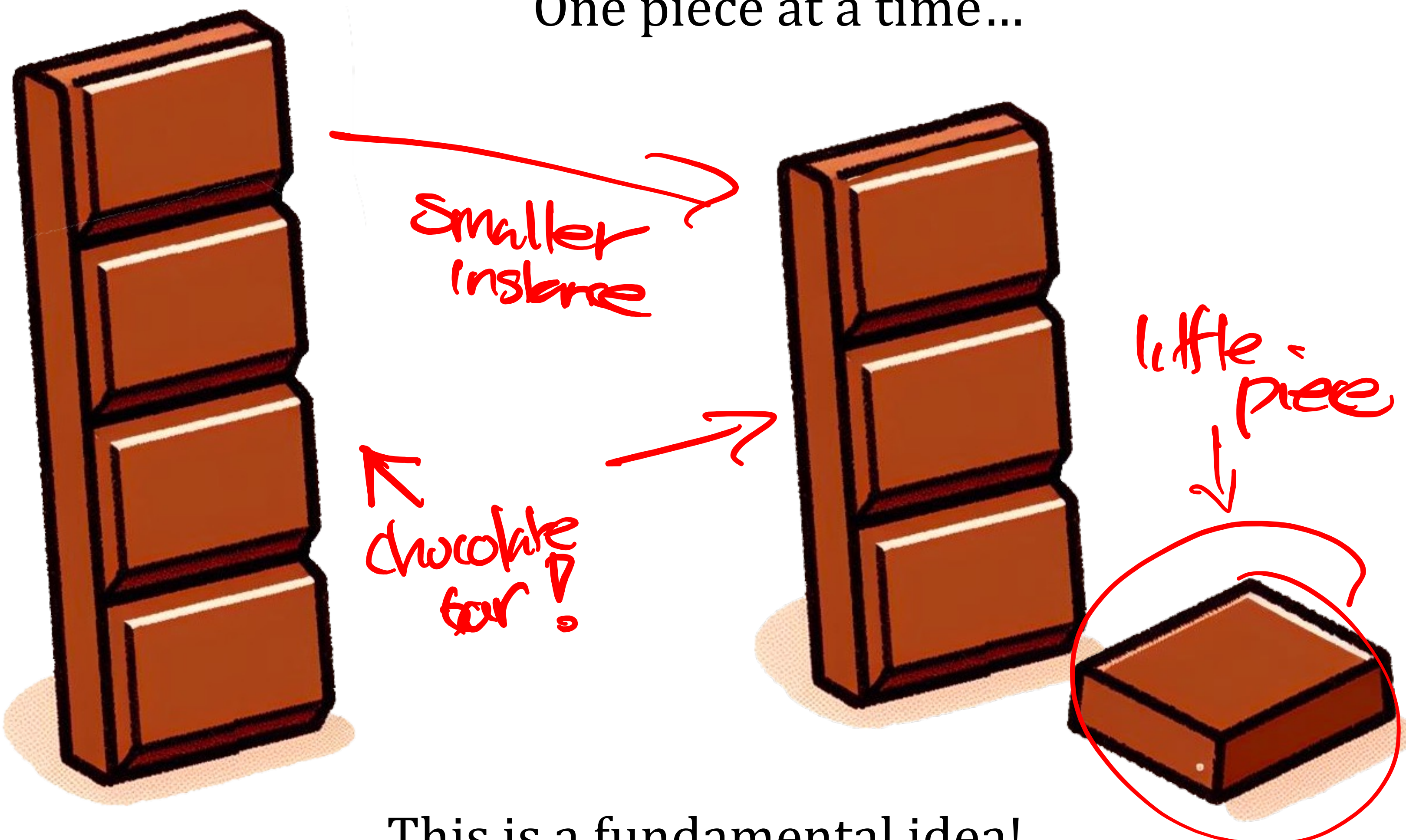
romanesco broccoli


Yikes!


Let's Recurse!
Yikes!

How to Eat Chocolate

One piece at a time…



This is a fundamental idea!

Let's write factorial!

def fac(n):

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$$

or

$$6! = 6 \times \underbrace{(5 \times 4 \times 3 \times 2 \times 1)}_{?}$$

Recurse!

fac(3)   N = 3

```
def fac(N):
    """ returns factorial of N
    """
    if N == 0:
  A     return 1

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

What does fac(3) return? ___

When working,

- How many times does line A run?
- How many times does line B run?
- How many N's are alive at once?!

Recurse!

fac(3)　N = 3

B　3 * fac(2)　N = 2, next...

B　2 * fac(1)　N = 1, next...

B　1 * fac(0)　N = 0, next...

A　1

```
def fac(N):
    """ returns factorial of N
    """
    if N == 0:
A       return 1

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

fac(3) returns _6_

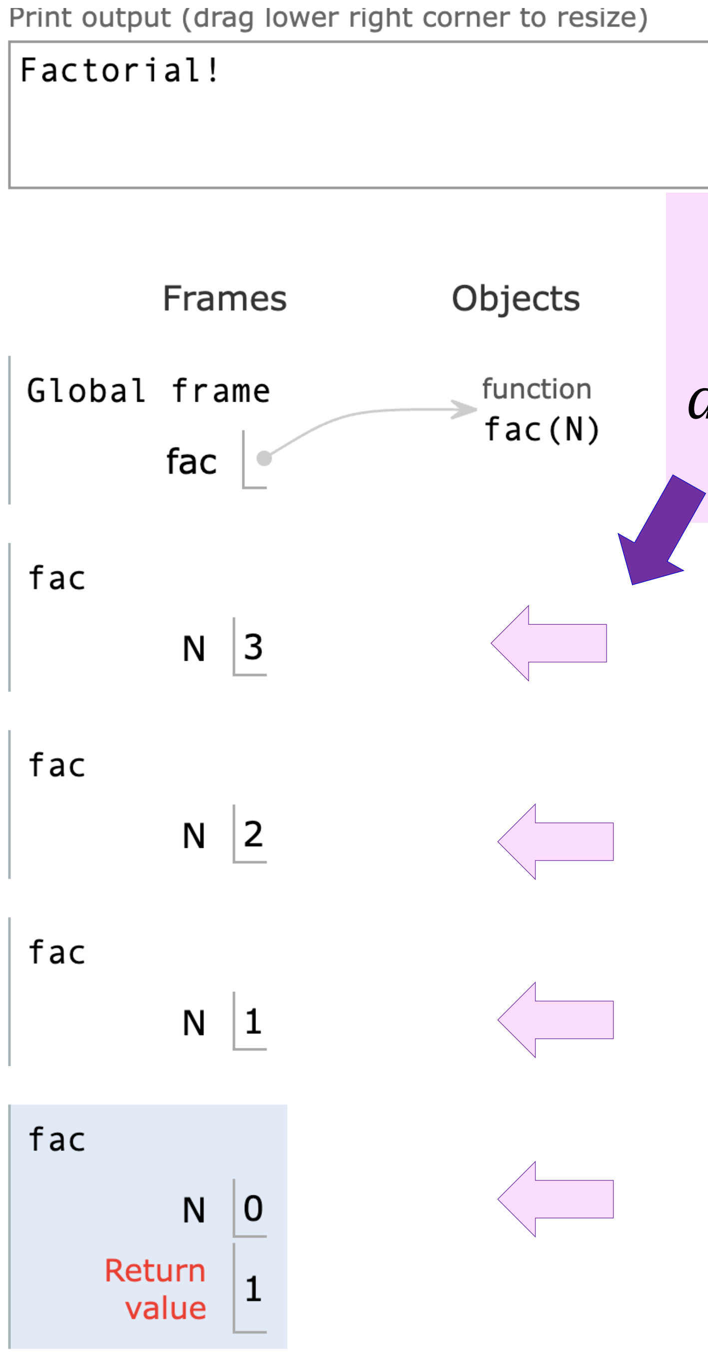
- How many times does line A run?
- How many times does line B run?
- How many N's are alive at once?!

A ~ 1 time

B ~ 3 times

4 N's total!

pythontutor.com



There are many different values of N – *all alive simultaneously*, in the **stack**

how would you <u>design</u> this?

Planning recursively...

Caution: A base case is "always" needed...

```
def fac(N):

    if N == 0:
        return 1

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

EMPTY case

... but it's not always **1!**

Base case

General case!

Recursive case

Empty case! *So many ways ... !?*

a.k.a "base case"

EMPTY case | BASE case

(or maybe)

the empty **integer**	→	0	1
the empty **float**	→	0.0	1.0
the empty **string**	→	""	
the empty **list**	→	[]	

Thinking recursively...

```
def fac(N):

    if N == 0:
        return 1

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

EMPTY case

General case!

Base case

Recursive case

Crazy! How can we multiply **N** times something that hasn't happened yet?!

Acting recursively

```
def fac(N):

  if N == 0:
    return 1

  else:

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

this recursion happens first!

```
def fac(N):

  if N == 0:
    return 1

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

hooray for variables!

Conceptual

Actual

Recursion example: *vwl(S)*

```
#
# vwl example
#
def vwl(S):
    """vwl returns the number of vowels in S
       input:  S, which will be a string
    """
    if S == '':        # if S is the empty string
        return 0       # it has no vowels
    elif S[0] in 'aeiou':       # if first-of-S is a vowel
        return 1 + vwl(S[1:])   # add 1 to # of vwls in rest-of-S
    else:
        return 0 + vwl(S[1:])   # otherwise, don't add 1
                                # the  0 +  is nice, but not needed
```

*human explanation – of what's **wanted**!*

*human explanations – of what's **happening***

syntactic stuff!

syntactic definition

today: **bridging** these!

The idea...

vwl(S), the total # of vowels in
S = `'alien'`

is `'a'` a vowel?

+

# of vowels in
`'lien'`

first

rest

The idea...

vwl(S), the total # of vowels in

`S = 'alien'`

```
elif S[0] in 'aeiou':
```

is `'a'` a vowel?

+

```
vwl(S[1:])
```

# of vowels in `'lien'`

first

rest

The idea...

vwl(S), the total # of vowels in

`S = 'lien'`

`elif S[0] in 'aeiou':`

is `'l'` a vowel?

+

`vwl(S[1:])`

# of vowels in `'ien'`

first

rest

The idea...

vwl(S), the total # of vowels in

S = **'ien'**

`elif S[0] in 'aeiou':`

is **'i'** a vowel?

+

`vwl(S[1:])`

# of vowels in **'en'**

first

rest

The idea...

vwl(S), the total # of vowels in

S = 'en'

```
elif S[0] in 'aeiou':
```

is 'e' a vowel?

+

```
vwl(S[1:])
```

# of vowels in 'n'

first

rest

The idea...

vwl(S), the total # of vowels in

S = 'n'

`elif S[0] in 'aeiou':`

is 'n' a vowel?

+

`vwl(S[1:])`

# of vowels in

''

first

rest

The idea...

vwl(S), the total # of vowels in

S = ''

```python
if S == '':          # if S is the empty string
    return 0         # it has no vowels
```

vowel?

+

# of vowels in

''

first

rest

The idea, in one slide:

vwl(S), the total # of vowels in *S*

```
elif S[0] in 'aeiou':
```

is **S[0]** a vowel?

+

```
vwl(S[1:])
```

# of vowels in **S[1:]**

first

rest

Recursion example: *vwl(S)*

total # of vowels in
S

Analysis...

is **s[0]** a
vowel?

+

# of vowels in
s[1:]

... via *self-similarity!*

first

rest

Indexing + slicing!

the first-of-S

is `S[0]` a vowel?

+

the rest-of-S

# of vowels in `S[1:]`

first

rest

hw1

if you worked on lab and submit pr1+pr2 :
you'll get full credit for pr1 + pr2

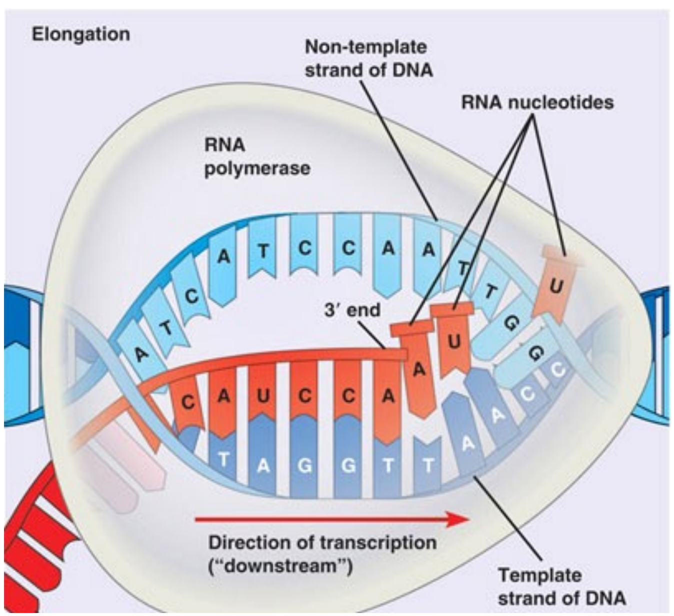
be sure to submit both pr1+pr2...

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

either way: submit pr1 + pr2

complete and submit hw1pr3 + start hw2pr4

Is this Python??



DNA transcription

Extra Credit: *Pig Latin / CodingBat*

hw1

if you worked on lab and submit pr1+pr2 :
you'll get full credit for pr1 + pr2

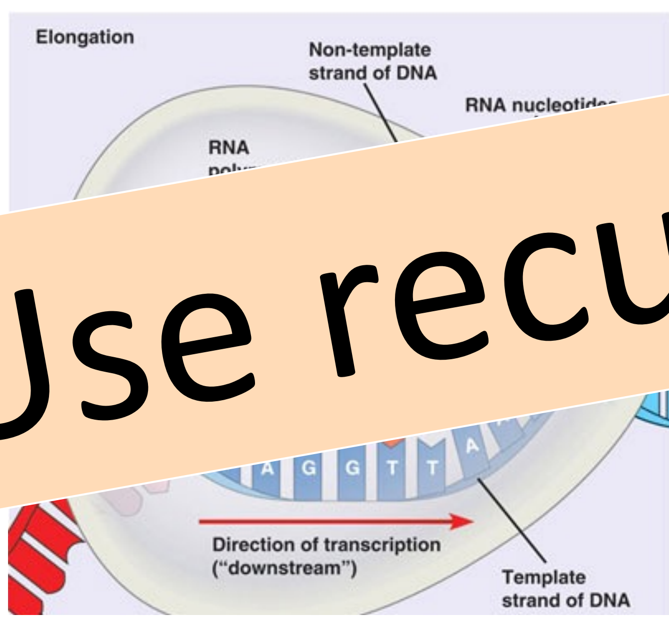
be sure to submit both pr1+pr2...

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

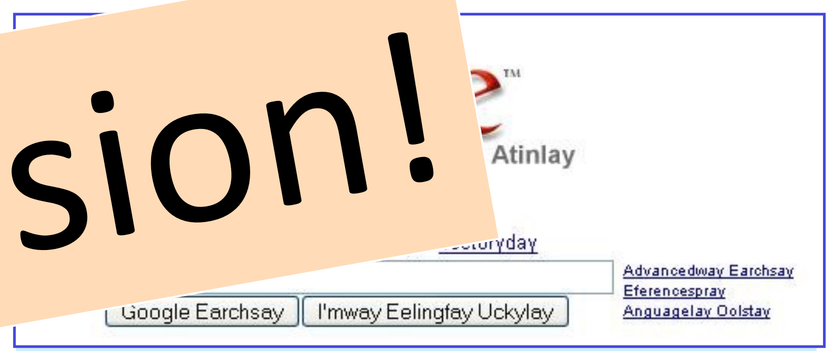
either way: submit pr1 + pr2

complete and submit hw1pr3 + start hw2pr4

Is this Python??



Use recursion!



Extra Credit: *Pig Latin / CodingBat*

DNA transcription

hw1

if you worked on lab and submit pr1+pr2 :
you'll get full credit for pr1 + pr2

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

either way: submit pr1 + pr2

complete and submit hw1pr3 + start hw2pr4

Recursion-free!

Is this Python??

Use PythonBat!

due for week 2

pythontutor.com

There are six different values of S – *all alive simultaneously*, in the **stack**

Python 3.6
(known limitations)

```
print("vwl!")

def vwl( S ):
    """ vwl counts vowels
            input: a string s
            output: # of vowels
    """
    if S == '':
        return 0
    elif S[0] in 'aeiou':
        return 1 + vwl( S[1:] )
    else:
        return vwl( S[1:] )

result = vwl( 'alien' )
print("result is", result)
```

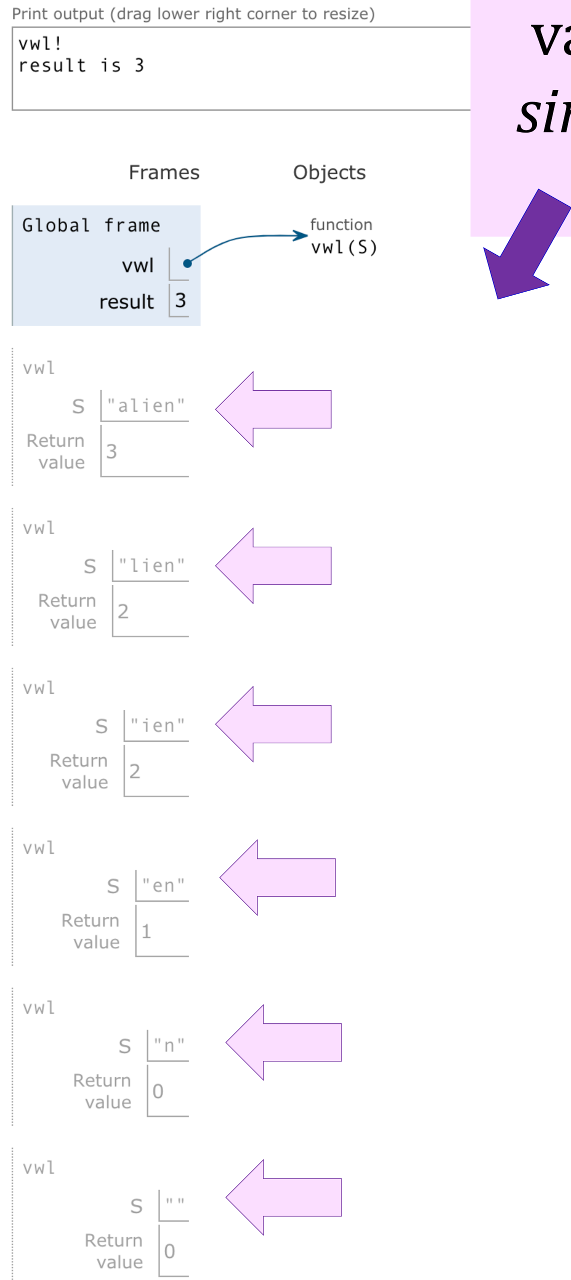
Edit this code

st executed
execute

<< First | < Prev | Next > | Last >>

Done running (33 steps)

ization (NEW!)



Variations!

How could we CHANGE this function to "keep" all of the vowels? That is, it should return **'aie'** instead of 3

```
def vwl(s):
    """ returns # of vowels in s
    """
    if s == '':
        return 0

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

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

EMPTY case

BASE case

Specific case

General case!

here's **keepvwl**

Writing keepvwl, to return `'aie'` instead of 3

```
def keepvwl( S ):
    if len(S) == 0:
        return ''

    elif S[0] in 'aeiou':
        return S[0] + keepvwl(S[1:])

    else:
        return '' + keepvwl(S[1:])
```

EMPTY case

EMPTY output

Specific case

Specific output

General case

General output!

dropvwl ?

v_w_l ?

cVcVc ?

others?!

here's **keepvwl**

[A] What is `keepvwl('recursion')` ?

[A]

```
def keepvwl( S ):
    if len(S) == 0:
        return ''

    elif S[0] in 'aeiou':
        return S[0] + keepvwl(S[1:])

    else:
        return '' + keepvwl(S[1:])
```

[B] When running **[A]**, how many times does this *base-case* line `return`?

[B]

[C] When running **[A]**, how many times does *this elif-case* line `return`?

[C]

[D] When running **[A]**, how many times does *this else-case* line `return`?

[D]

Extra! For what word w does `keepvwl(w)` return `'aeiou'` ?

create **drpvwl**

Fill in the code at left in order to...

```
def dropvwl( S ):
    if len(S) == 0:
        return ___

    elif S[0] in 'aeiou':
        return ___ + dropvwl(S[1:])

    else:
        return ___ + dropvwl(S[1:])
```

... first, finish **`drpvwl`**

then...

... change to **`v_w_l`**

then...

... change to **`cVcVc`**

here's keepvwl

[A] What is keepvwl('recursion') ?

'euio' [A]

```
def keepvwl( S ):
    if len(S) == 0:
        return ''

    elif S[0] in 'aeiou':
        return S[0] + keepvwl(S[1:])

    else:
        return '' + keepvwl(S[1:])
```

[B] When running **[A]**, how many times does this *base-case* line `return`?

[B]

[C] When running **[A]**, how many times does *this elif-case* line `return`?

[C]

[D] When running **[A]**, how many times does *this else-case* line `return`?

[D]

Extra! For what word w does `keepvwl(w)` return `'aeiou'` ?

create drpvwl

```
def dropvwl( S ):
    if len(S) == 0:
        return ___

    elif S[0] in 'aeiou':
        return ___ + dropvwl(S[1:])

    else:
        return ___ + dropvwl(S[1:])
```

Fill in the code at left in order to...

... first, finish **drpvwl**

then...

... change to **v_w_l**

then...

... change to **cVcVc**

here's **keepvwl**

[A] What is keepvwl('recursion') ?

[A] **'euio'**

```
def keepvwl( S ):
    if len(S) == 0:
        return ''

    elif S[0] in 'aeiou':
        return S[0] + keepvwl(S[1:])

    else:
        return '' + keepvwl(S[1:])
```

[B] When running **[A]**, how many times does this *base-case* line `return`? [B] **1**

[C] When running **[A]**, how many times does *this elif-case* line `return`? [C] **4**

[D] When running **[A]**, how many times does *this else-case* line `return`? [D] **5**

Extra! For what word w does keepvwl(w) return 'aeiou' ?

create **drpvwl**

Fill in the code at left in order to...

```
def dropvwl( S ):
    if len(S) == 0:
        return ''

    elif S[0] in 'aeiou':
        return '' + dropvwl(S[1:])
               '_'   'v'
    else:
        return s[0] + dropvwl(S[1:])
               s[0]    'c'
```

... first, finish **drpvwl**

then...

... change to **v_w_l**

then...

... change to **cVcVc**

```python
def dropvwl(s):
    """ returns only non-vowels in s!
    """
    if s == '':
      return ''

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

    else:
      return s[0] + dropvwl(s[1:])
```

base case! return the empty string

if vowel, leave it out!

if not a vowel, keep it!

```python
def v_w_l(s):
    """ replaces vowels with _
    """
    if s == '':
      return ''

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

    else:
      return s[0] + v_w_l(s[1:])
```

base case! return the "zero" of strings...

if a vowel, replace with a '_'

if not a vowel, keep it!

```python
def cVcVc(s):
    """ vowels -> V, consonants -> c
    """
    if s == '':
      return ''

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

    else:
      return  'c' + v_w_l(s[1:])
```

base case! return the "zero" of strings...

if a vowel, replace with a 'V'

if not a vowel, replace with a 'c'

```python
def VoWeL(s):
    """ SPoNGeBoBBiFy s
    """
    if s == '':
      return ''

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

    else:
      return s[0].upper() + VoWeL(s[1:])
```

base case! return the "zero" of strings...

if it's a vowel, keep s[0], the vowel itself!

if it's not a vowel, make it an UPPERCASE s[0]!

Variations!

Warning: this code runs!

but it it doesn't work!

```
def vwl(s):
    return vwl(s)
```

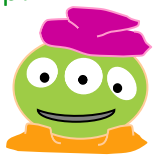
stackoverflow

Warning: this code runs!

but it has problems!

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

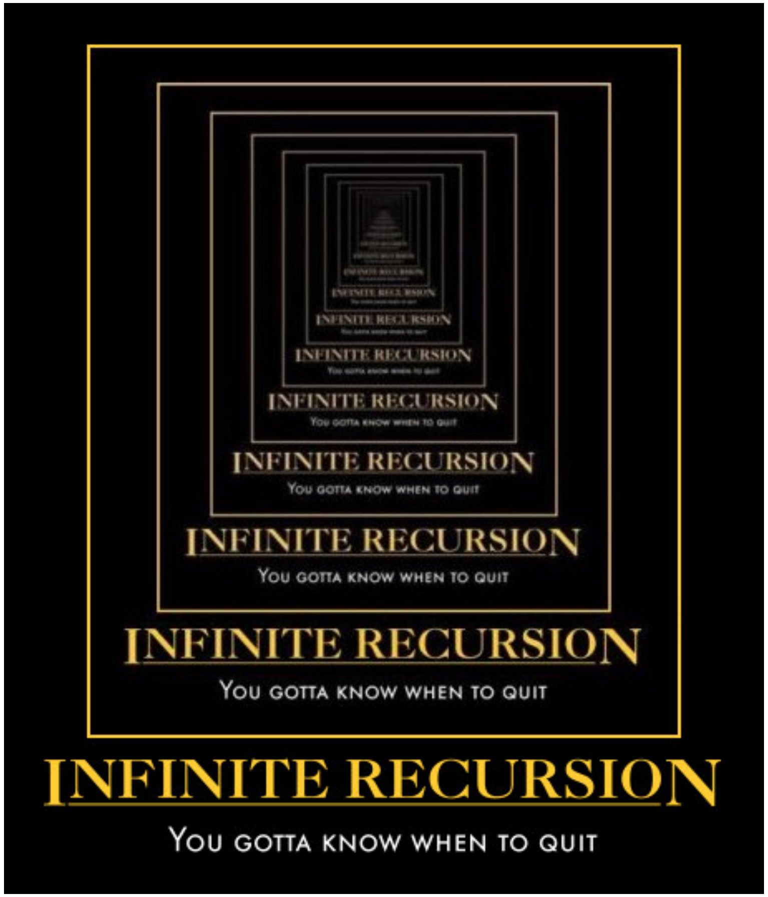
I wonder how this code
will **STACK** up?



```
def facBAD(N):
    print("N is", N)
    return N * facBAD(N-1)
```

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

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



Recursion

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

Google

recursion

All · Books · Images · Videos · News · More · Settings · Tools

About 37,000,000 results (0.50 seconds)

Did you mean: ***recursion***

Dictionary

Search for a word

re·cur·sion

/rəˈkərZHən/

noun MATHEMATICS • LINGUISTICS

the repeated application of a recursive procedure or definition.

- a recursive definition.
  plural noun: **recursions**

Translations, word origin, and more definitions

Definitions from Oxford Languages *Feedback*

en.wikipedia.org › wiki › Recursion_(computer_science)

Recursion (computer science) - Wikipedia

In computer science, **recursion** is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem. Such problems can generally be solved by iteration, but this needs to identify and index the smaller instances at programming time.

Types of recursion · Recursive programs · Recursion versus iteration

Google, 2021

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) =
```

Thinking ... *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!?

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

    elif p < 0:
        return

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

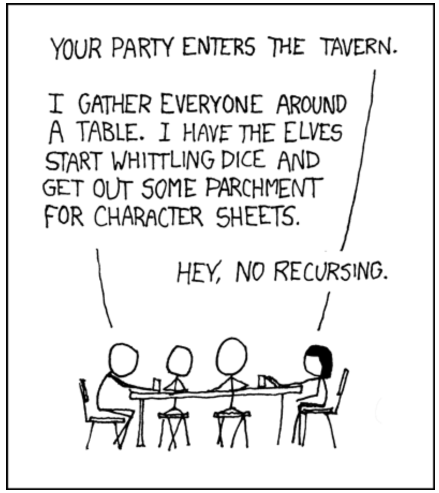
Extra! Can we also handle ***negative*** powers... ?

```
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... ?

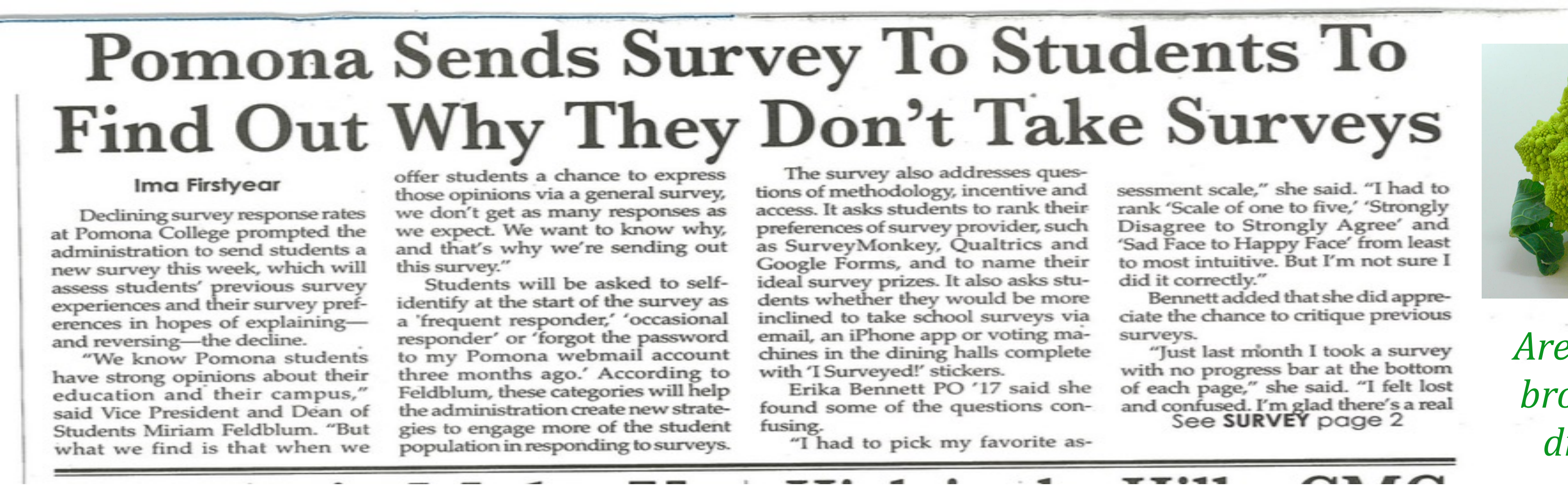
Recursion's advantage:

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



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

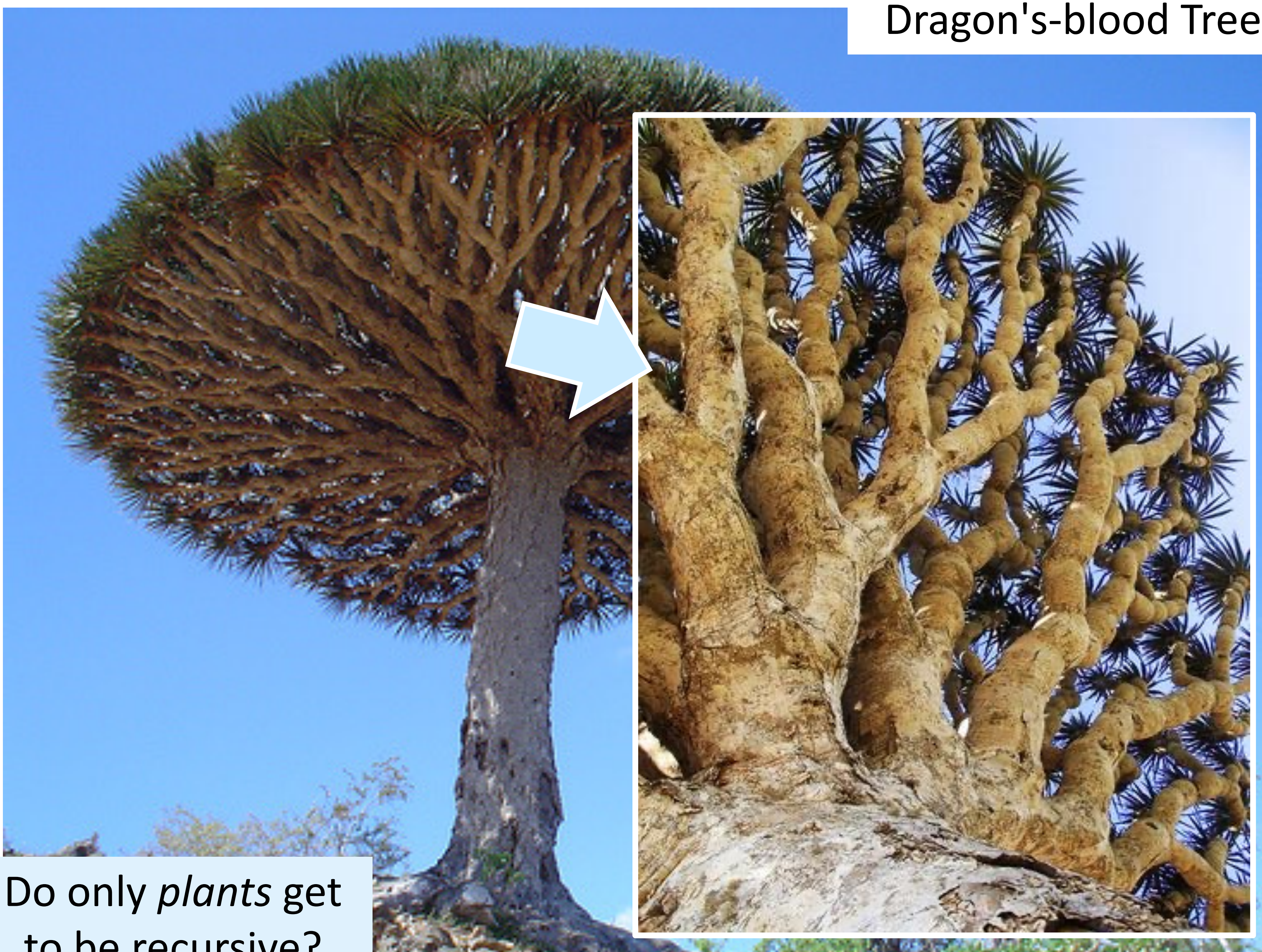
See SURVEY page 2

Are surveys the broccoli of our digital age?

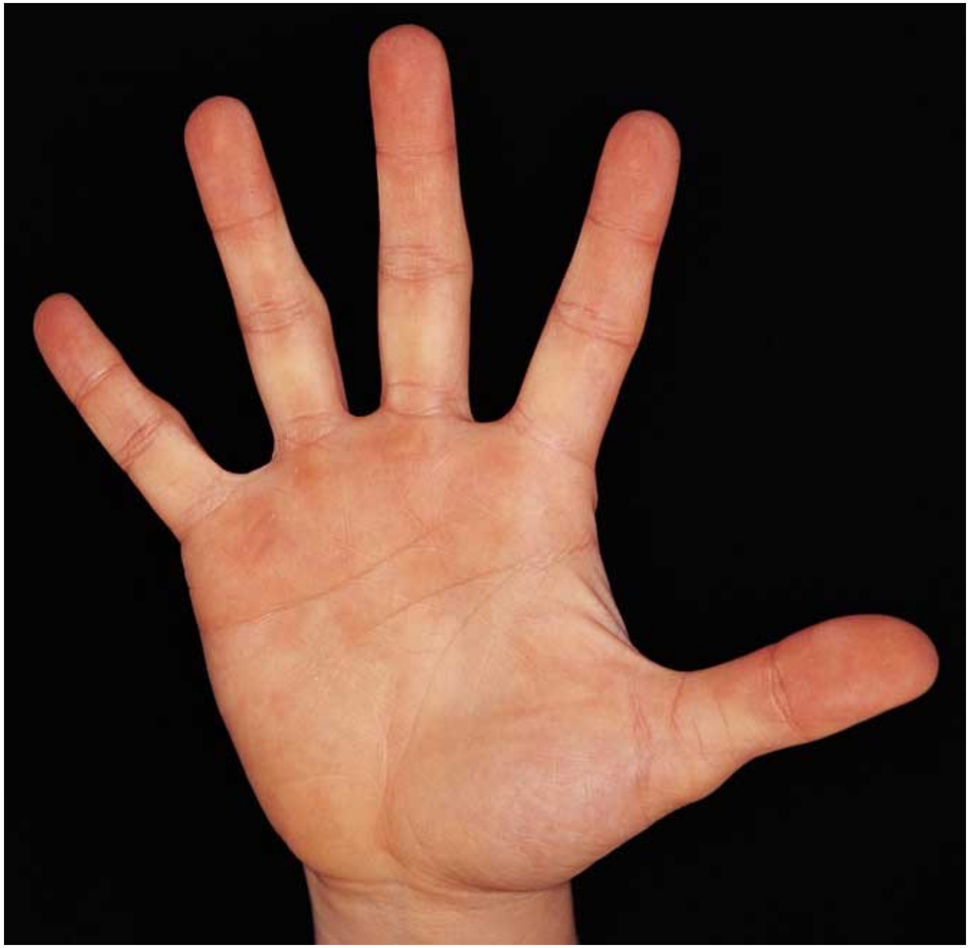
Recursion's advantage:

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

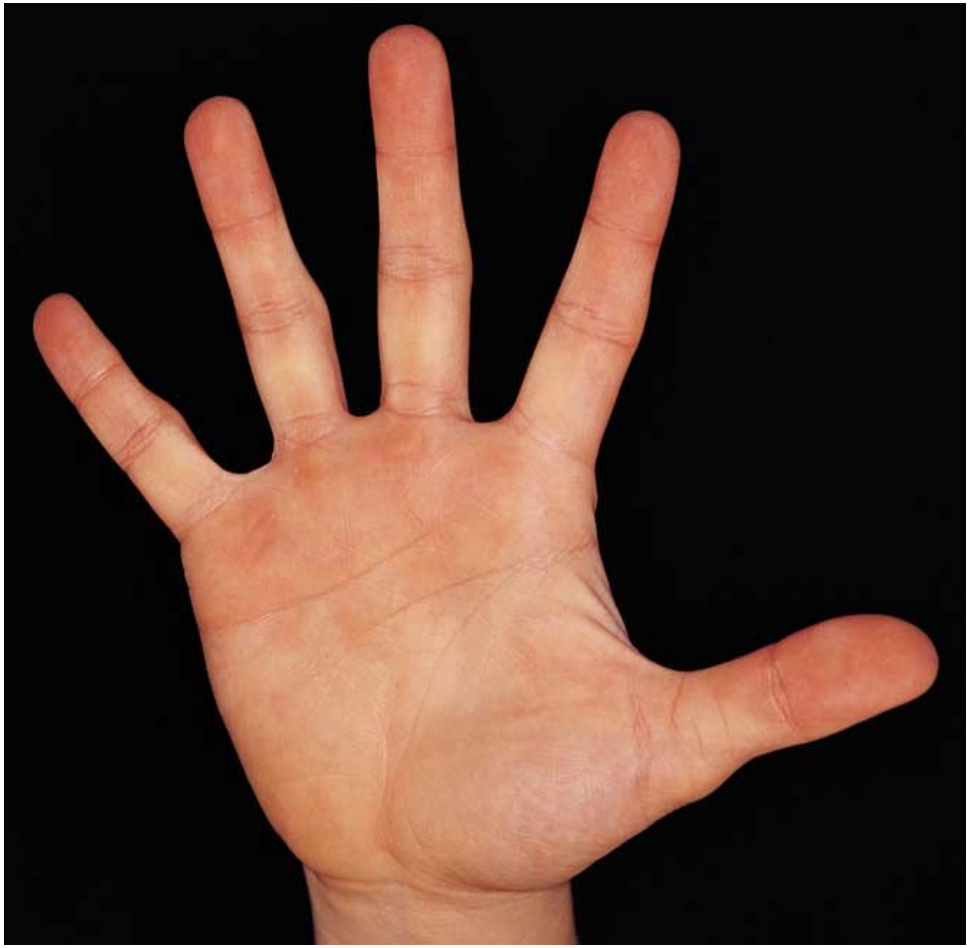
Dragon's-blood Tree



Do only *plants* get to be recursive?

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



or else!

or - one layer out !?

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

- former CS 5 student

It's the eeri**est**!

but that's meant *facetiously*...

Good luck with Homework #1

tutors @ McGregor: Th/F/Sa/Su/Mon.

More examples...