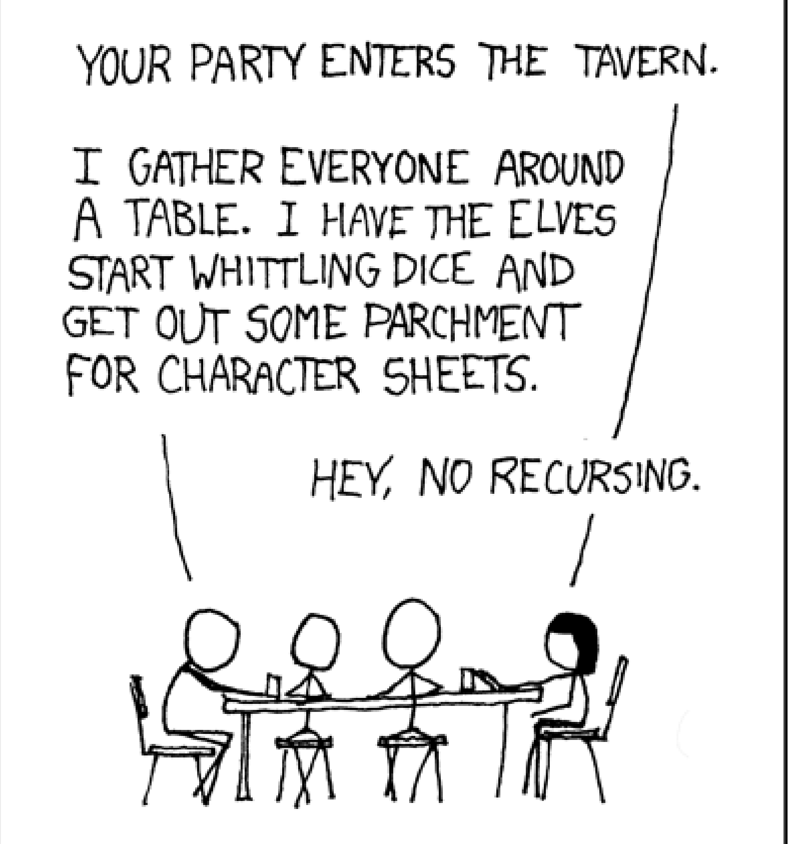
CS 5 Lecture 3

Functions and Recursion



INSTALLING THE XKCD DEVELOPMENT ENVIRONMENT

1. SPIN UP A VM
2. SPIN UP A VM INSIDE THAT VM
3. CONTINUE SPINNING UP NESTED VMs AND CONTAINERS UNTIL YOU GET FIRED

Last time: *Python slices...*

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

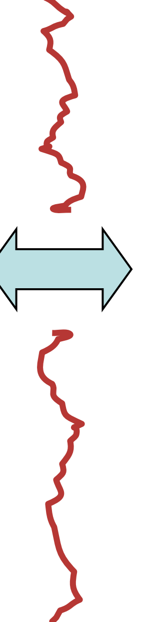
indexing `s[0] == 'a'`

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

and?

Computation's Dual Identity

Computation



Data Storage

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

variables ~ boxes

But what does the stuff on this side ***look like*** ?

Computation's Dual Identity

accessed through functions...

Computation

Data Storage

41
name: x
type: int
LOC: 300

42
name: y
type: int
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

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

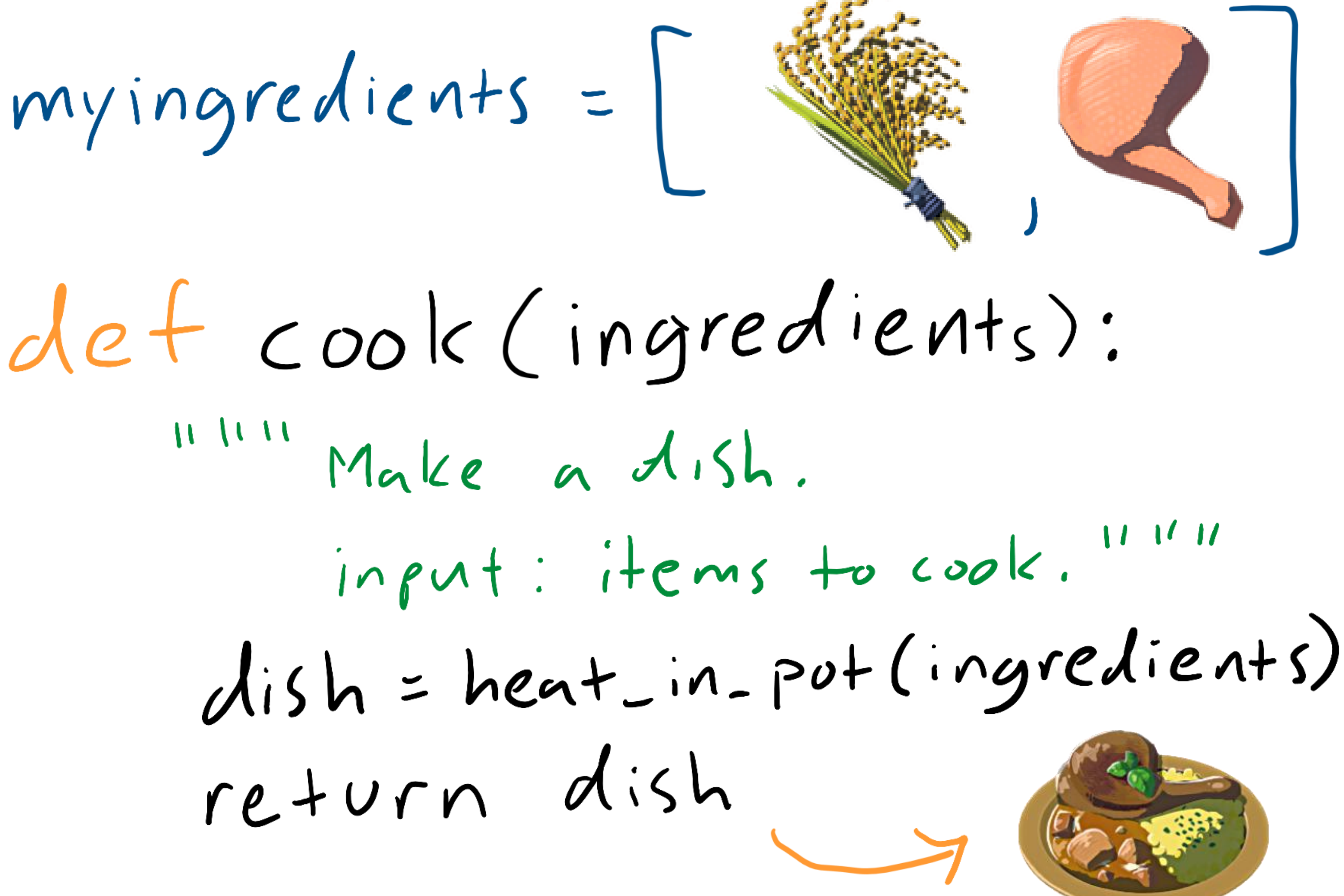
Math's googolizer

defined by ***what it relates***

+ what follows ***logically***

“Bunches
and
bunches
of horses
and
lunches”

THE LEGEND OF
ZELDA™
BREATH OF THE WILD



rice_dish = cook(myingredients)

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'
```

```
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.")
```

More Functions!

What's the difference?

```
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

```
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

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

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

this is a value for further use!

yes!

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

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

this turns lightbulbs on!

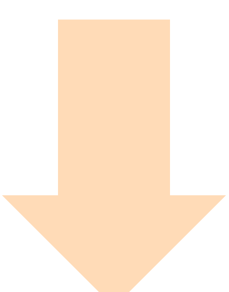
ouch!

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

... *which the terminal then prints!*

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

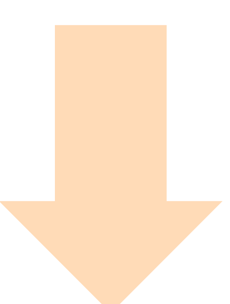
`return` >> **`print`**



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

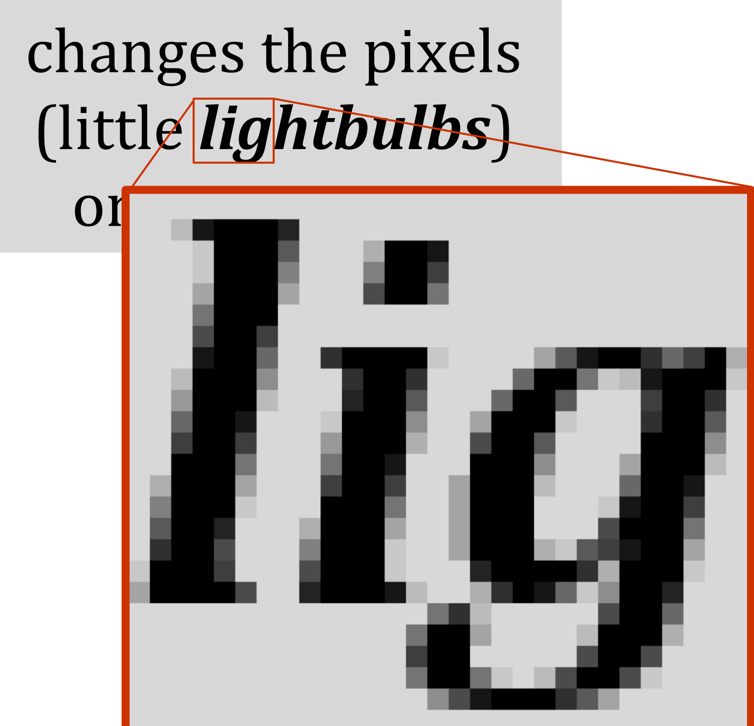
changes the pixels (little ***lightbulbs***) on your screen

`return` >> `print`



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

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



Terminology

function name

parameter

signature line

docstring

```
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]
```

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*

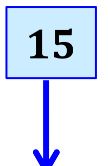
Big Ideas

- 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*

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
```

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?

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

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

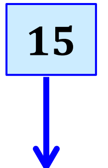
Functions!

Names: ______________________________

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

Quiz

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
```

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!

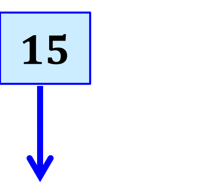
What is **`vwl("alien")`** here?

```
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?



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

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"

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"

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"

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)

```
1  def f(x):
2      if x == 0:
3          return 12
4      else:
5          return f(x-1) + 10*x
6
7  result = f(2)
8  print("f(2) is", result)
```

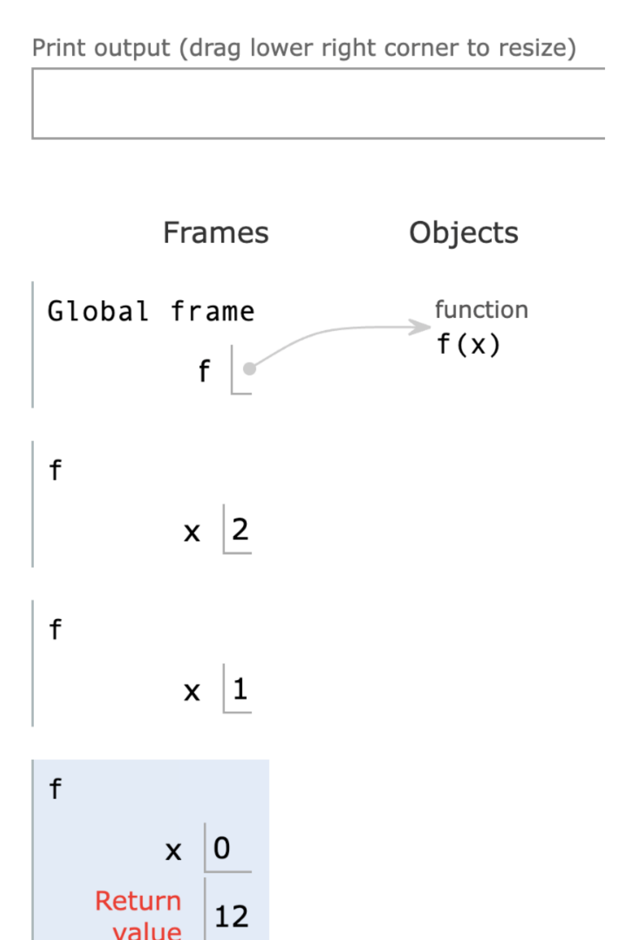
Edit this code

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"

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

call: f(1)
stack frame
local variables:
x = 1
need f(0)

call: f(0)
stack frame
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"

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"

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`

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`

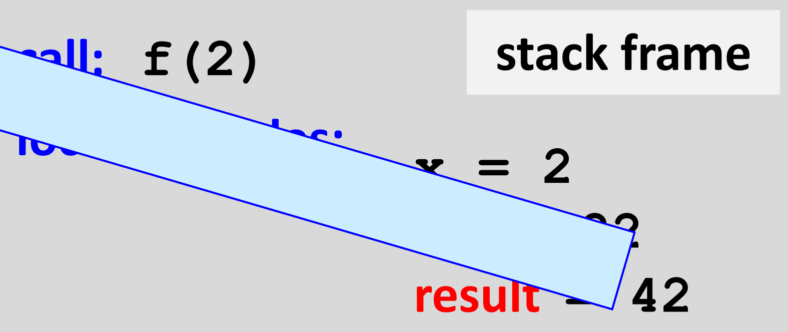
which then gets returned…

How functions work...

2

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

"the stack"



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"

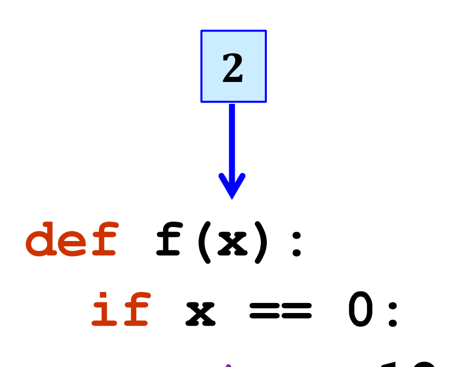
again, the stack is empty, but ready if another function is called...

Functions are software's **cells** ...

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

functions ***stack****.*

How functions work...




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

42

output

"the stack"

again, the stack is empty, but ready if another

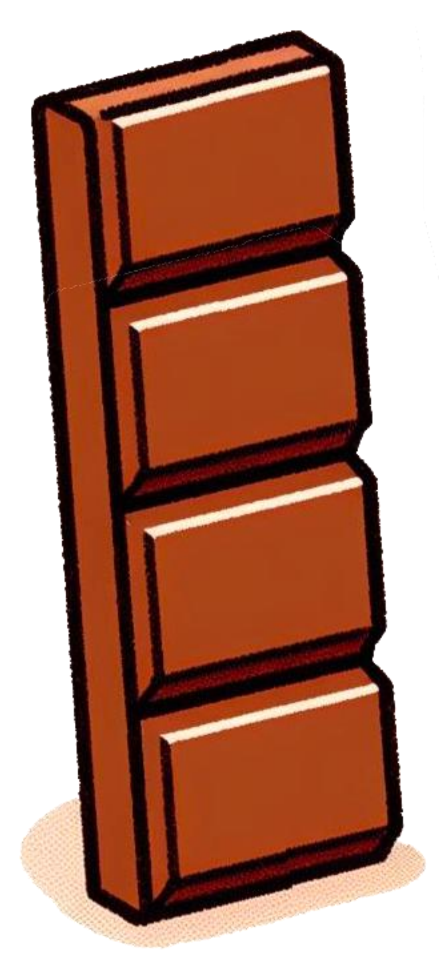
Pass those papers north!

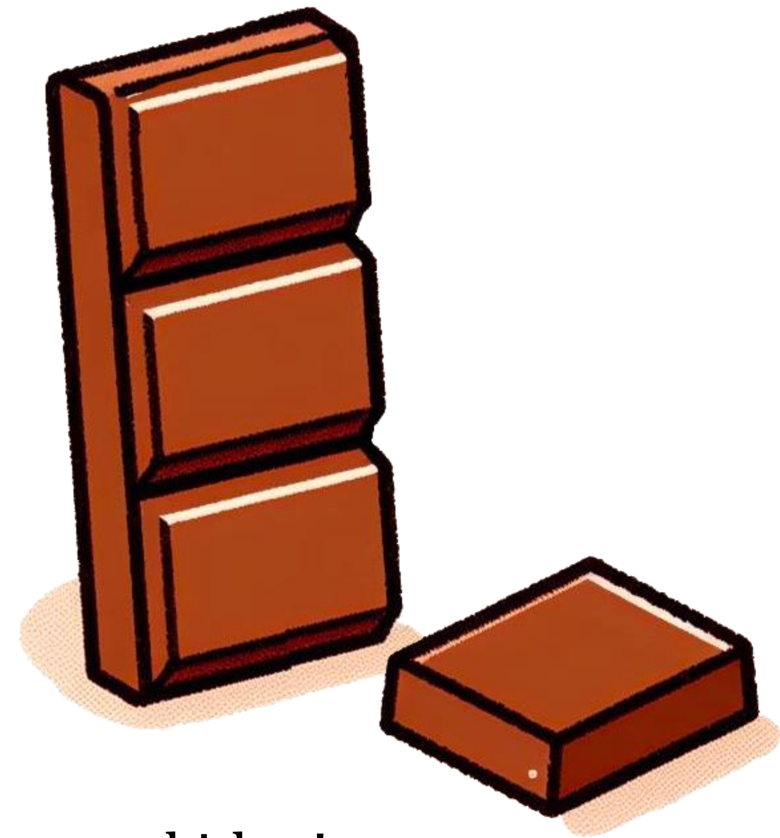
... each f'n is a ***self-contained*** *computational unit!*

functions ***stack****.*

How to Eat Chocolate Recursively

One piece at a time…





This is a fundamental idea!

“We have recursion at home…”

Q: Why is recursion important to learn?

- Teaches you to leverage **structure** to break large problems into **smaller sub-problems**.

- Can lead to more **concise** and **readable** code.

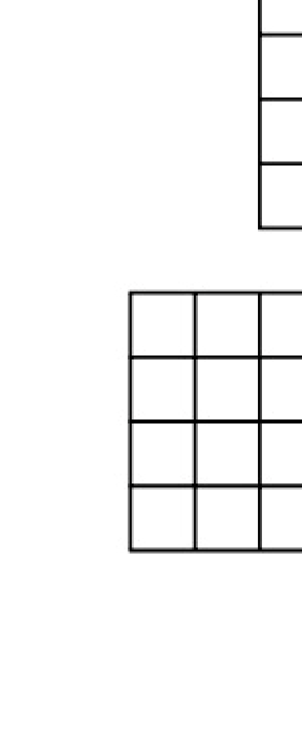
- **Technical interviewers at Meta, Microsoft, Google, Amazon, etc. will likely ask you to code a recursive function.**
  - The more you learn, the more you'll earn!*

** Not guaranteed in this economy!*

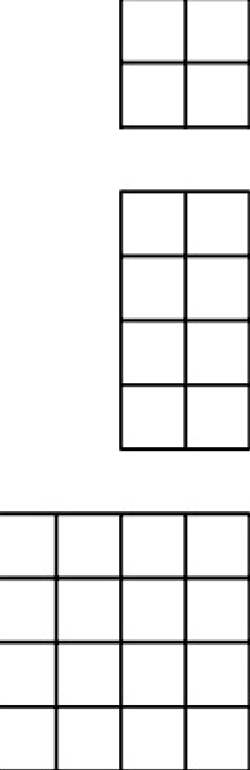
Chef Boyardee Inventory

LINKIN PARK
Recursion
REANIMATION

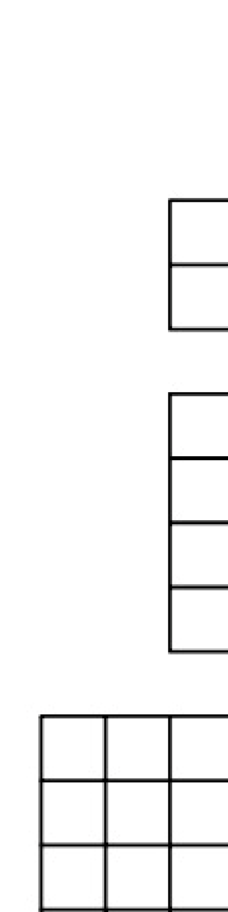
Stack



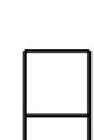
Stack

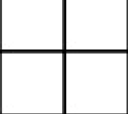


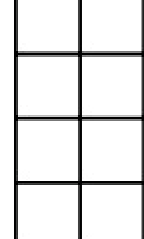
Stack

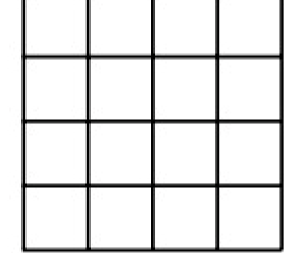


Stack

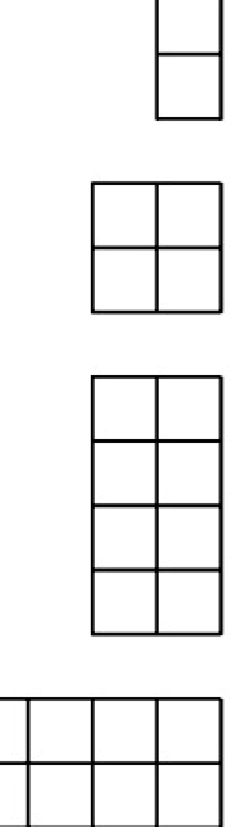




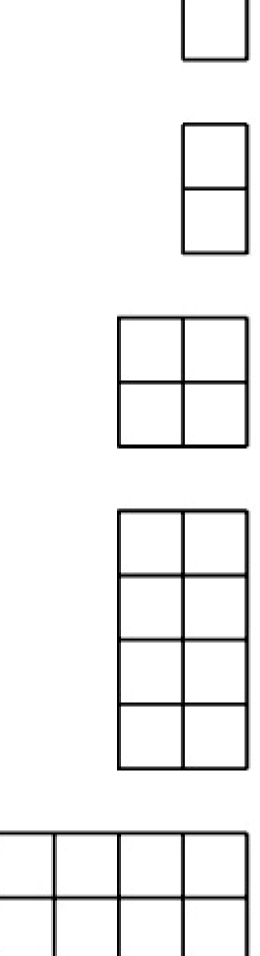




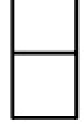
Stack

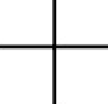


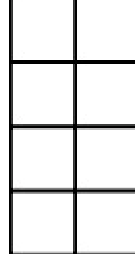
Stack

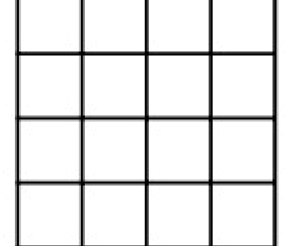


Stack

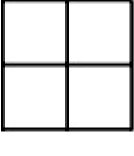
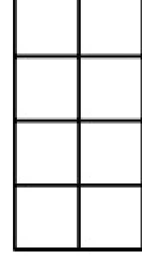
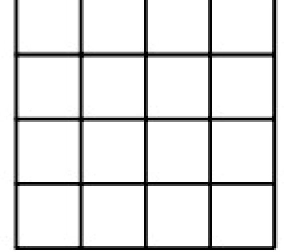




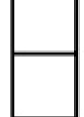


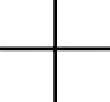


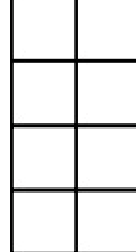
Stack

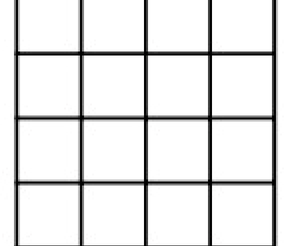




Stack

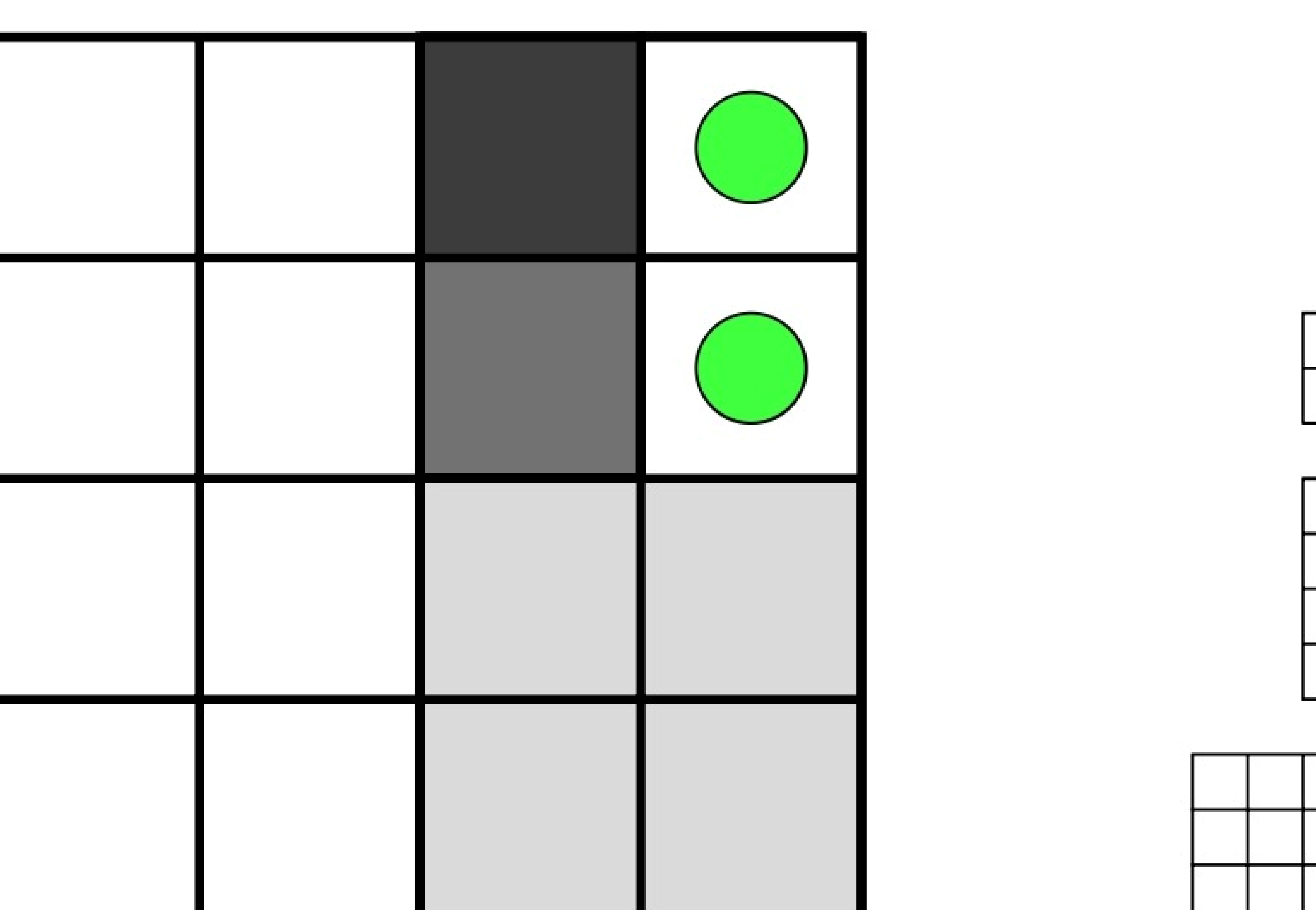


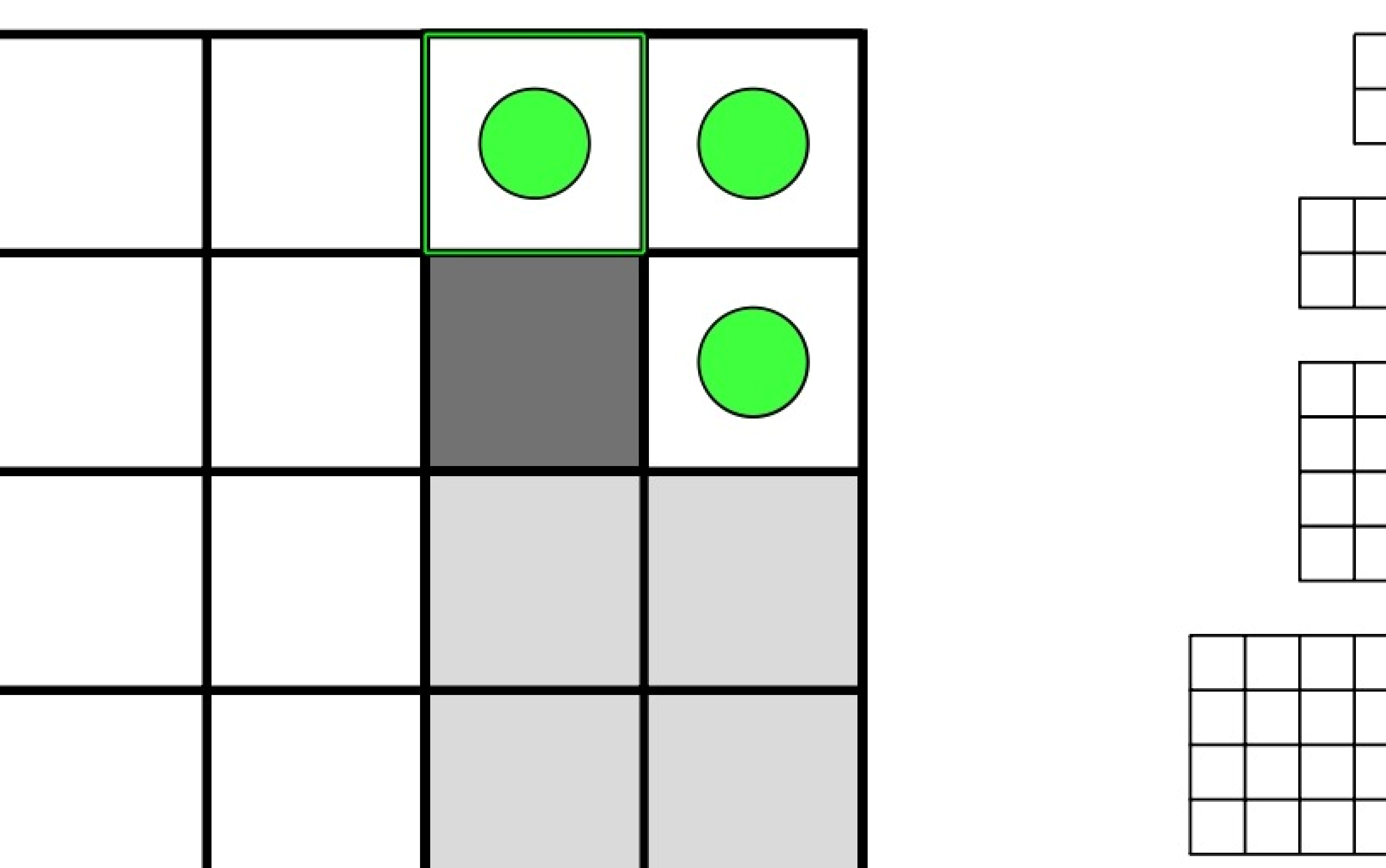




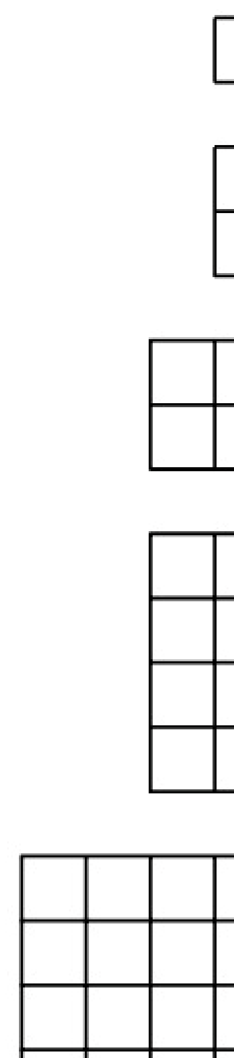


Stack

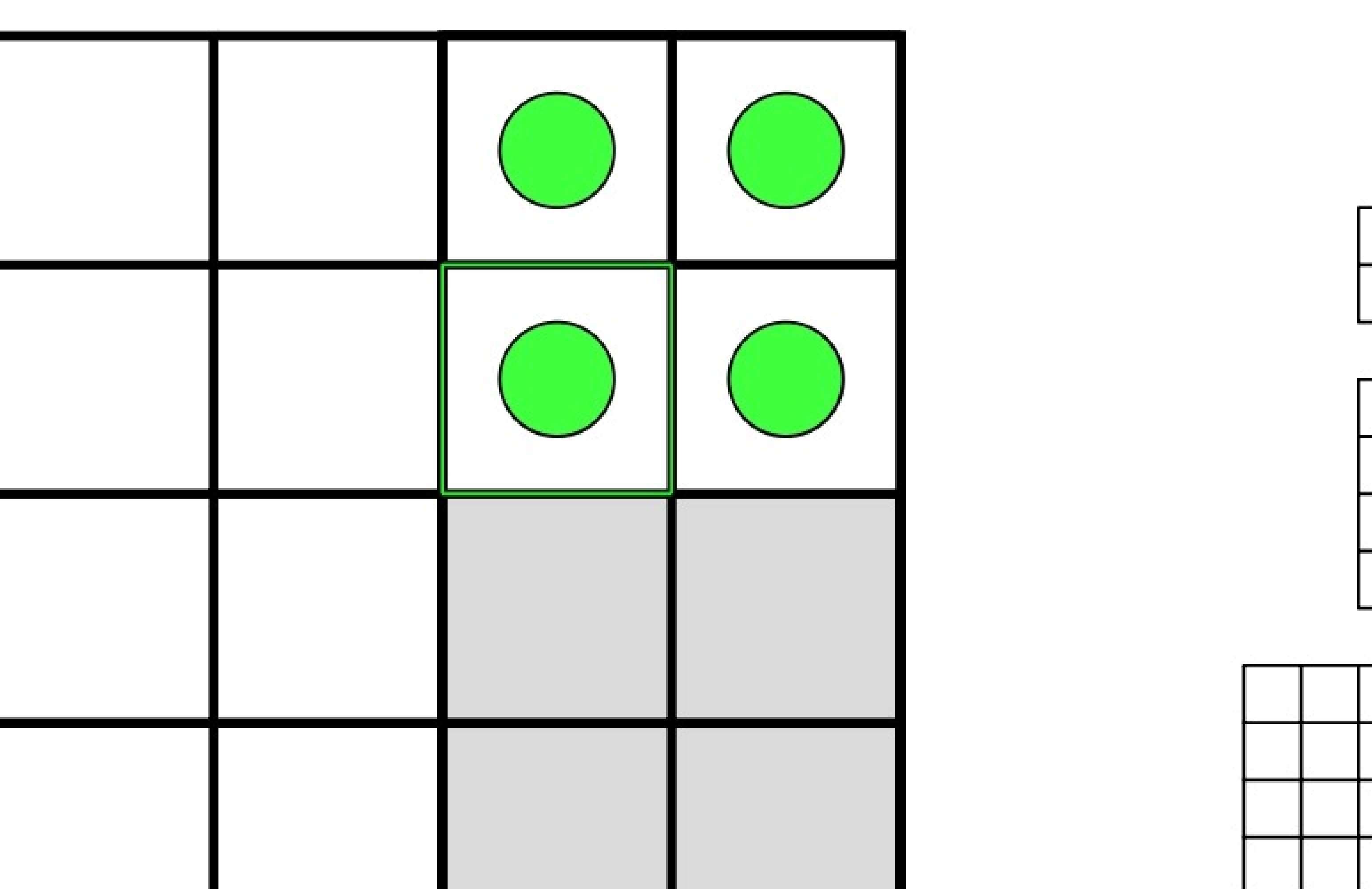



Stack

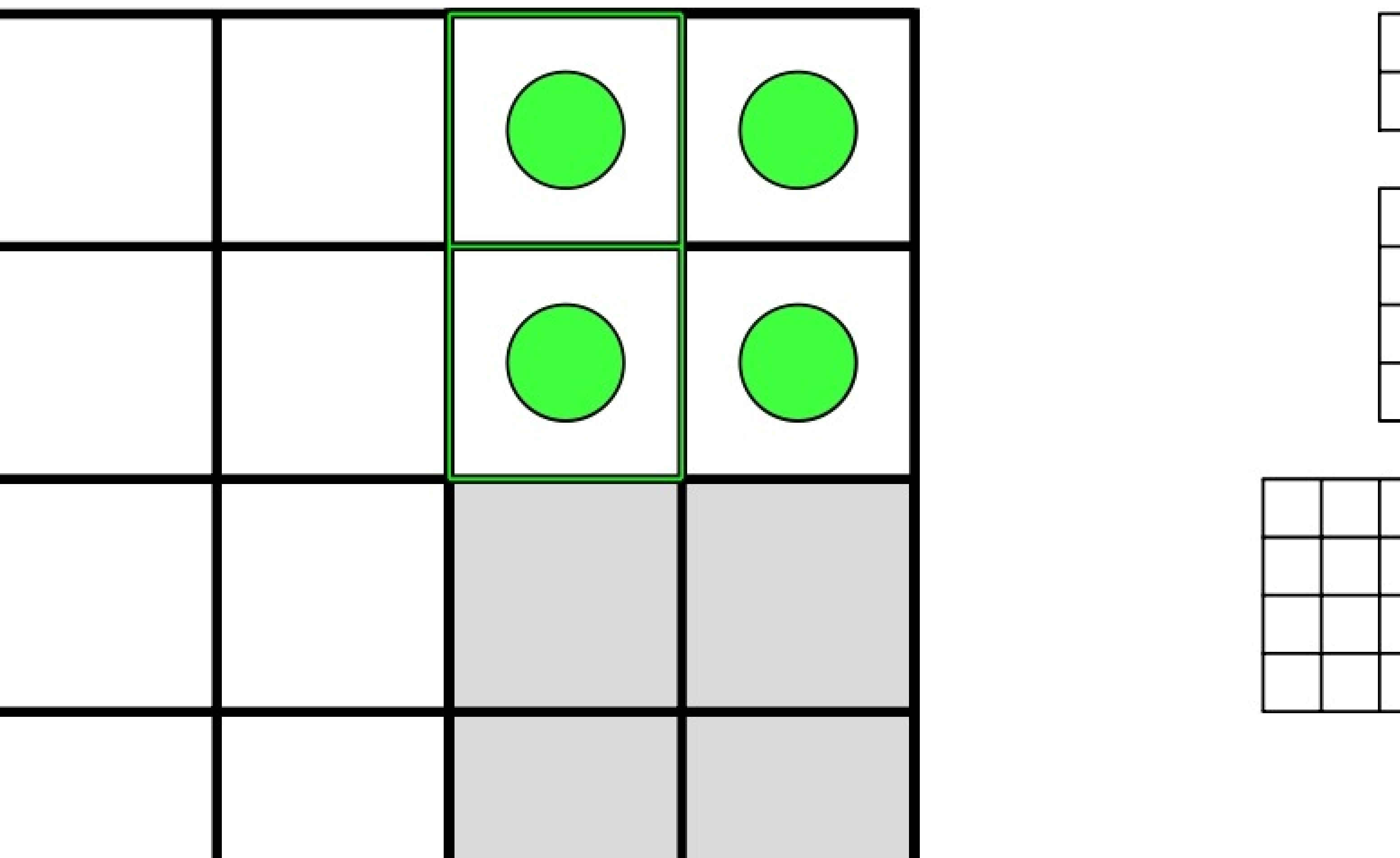
Stack



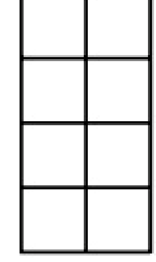
Stack

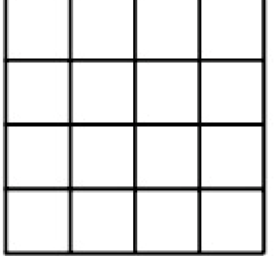


Stack

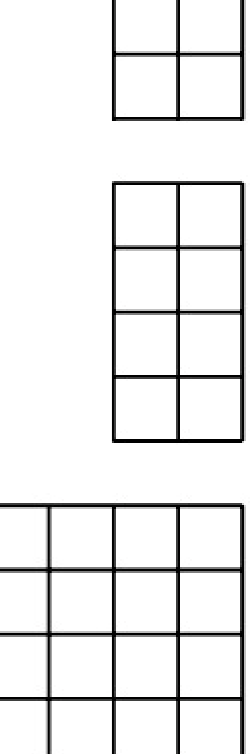


Stack

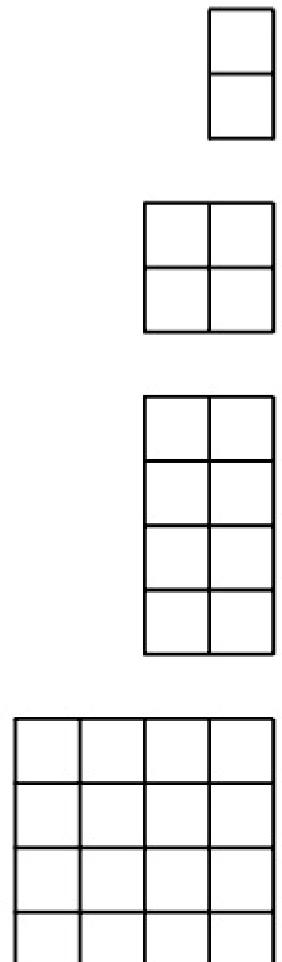




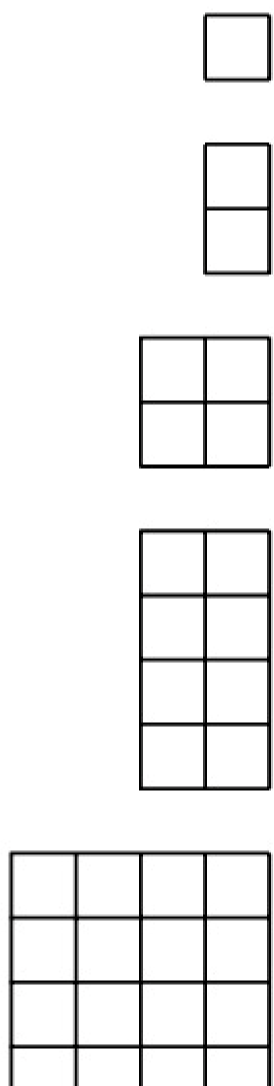
Stack



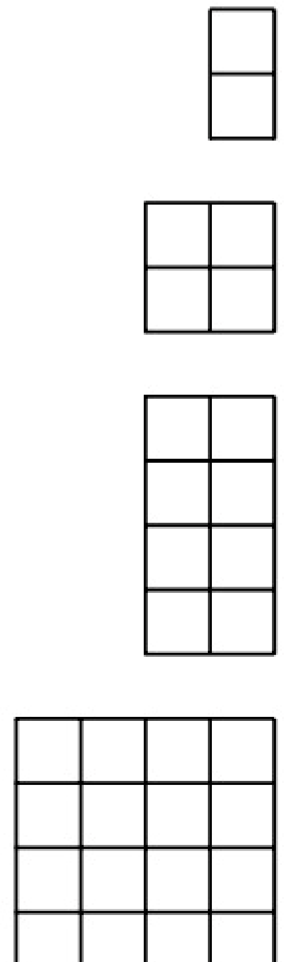
Stack



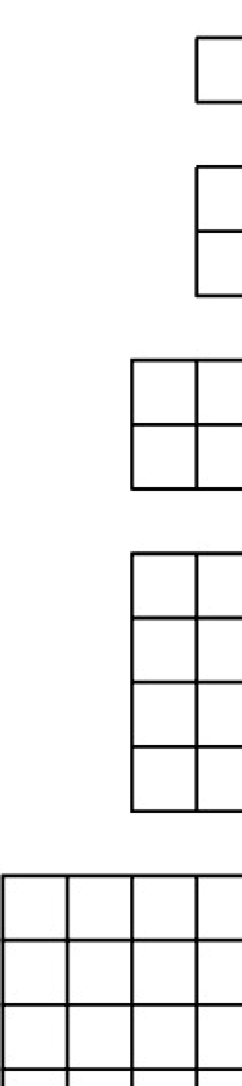
Stack



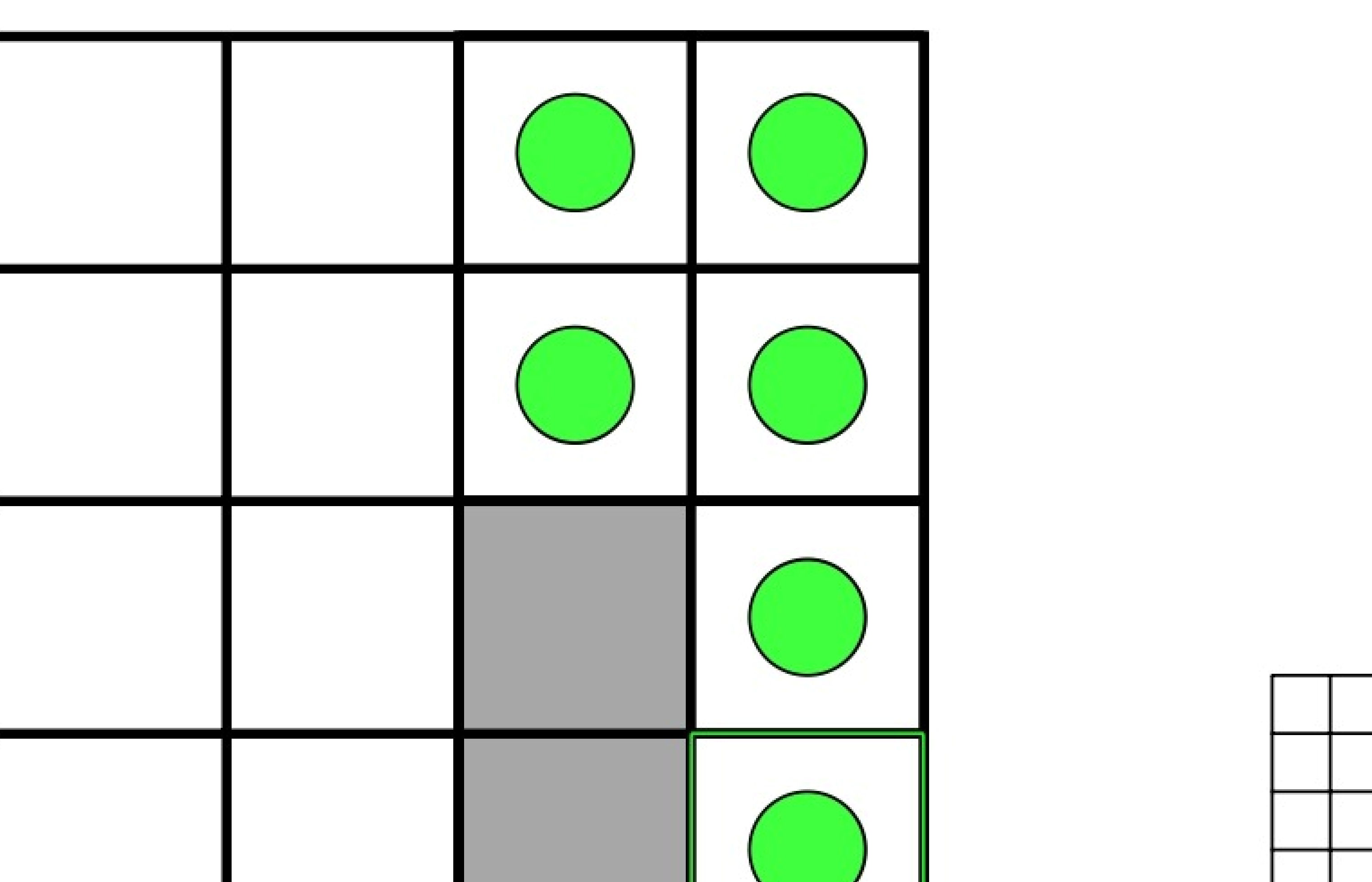
Stack



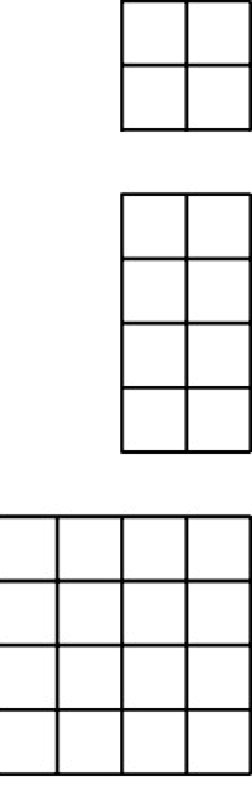
Stack



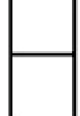
Stack

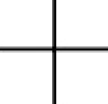


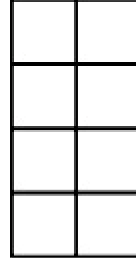
Stack

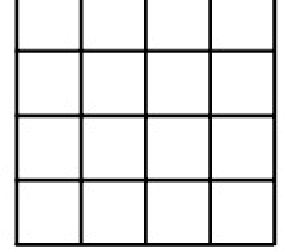


Stack

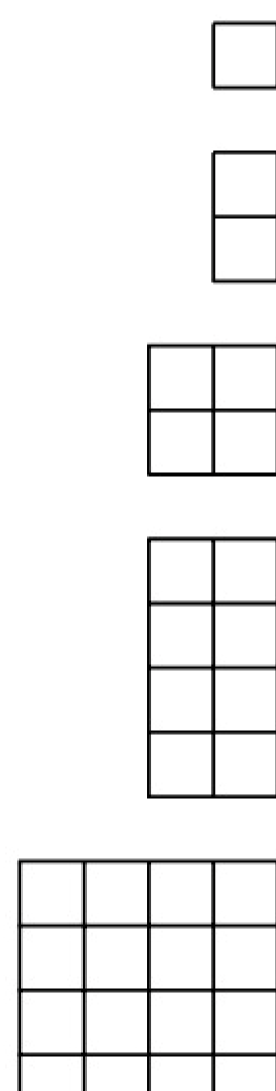




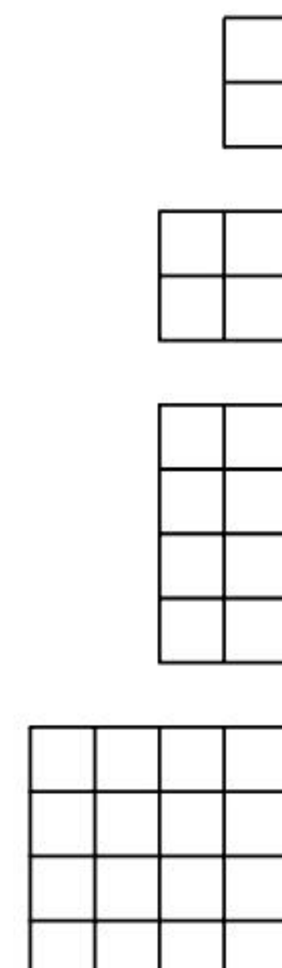




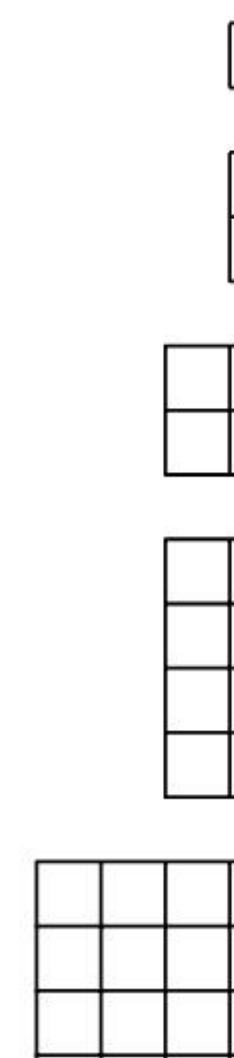
Stack



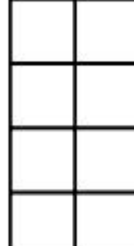
Stack

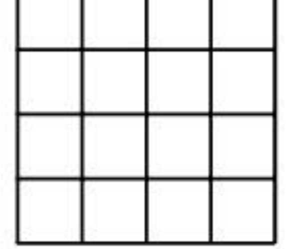


Stack

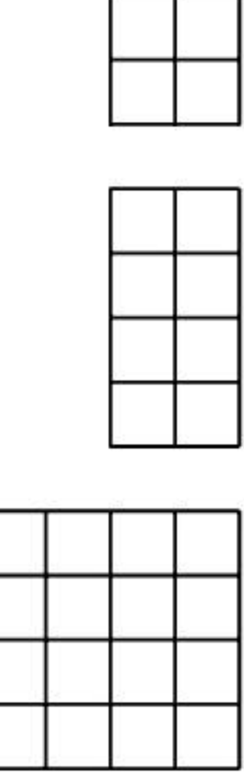


Stack

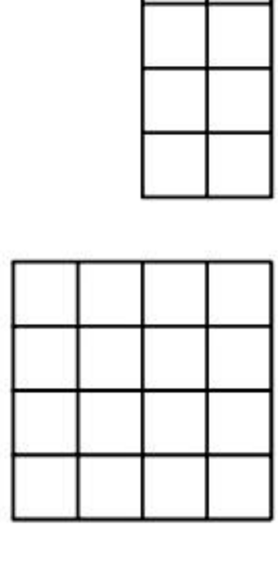




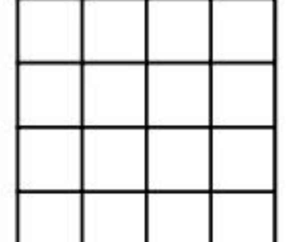
Stack



Stack

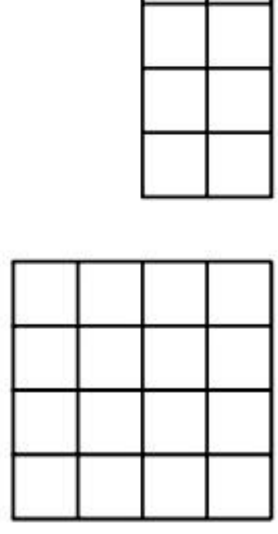


Stack



Stack

etc.



Let's write factorial!

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

or

First Rest

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

$$= 6 \times 5!$$

Recurse!

fac(3) N = 3

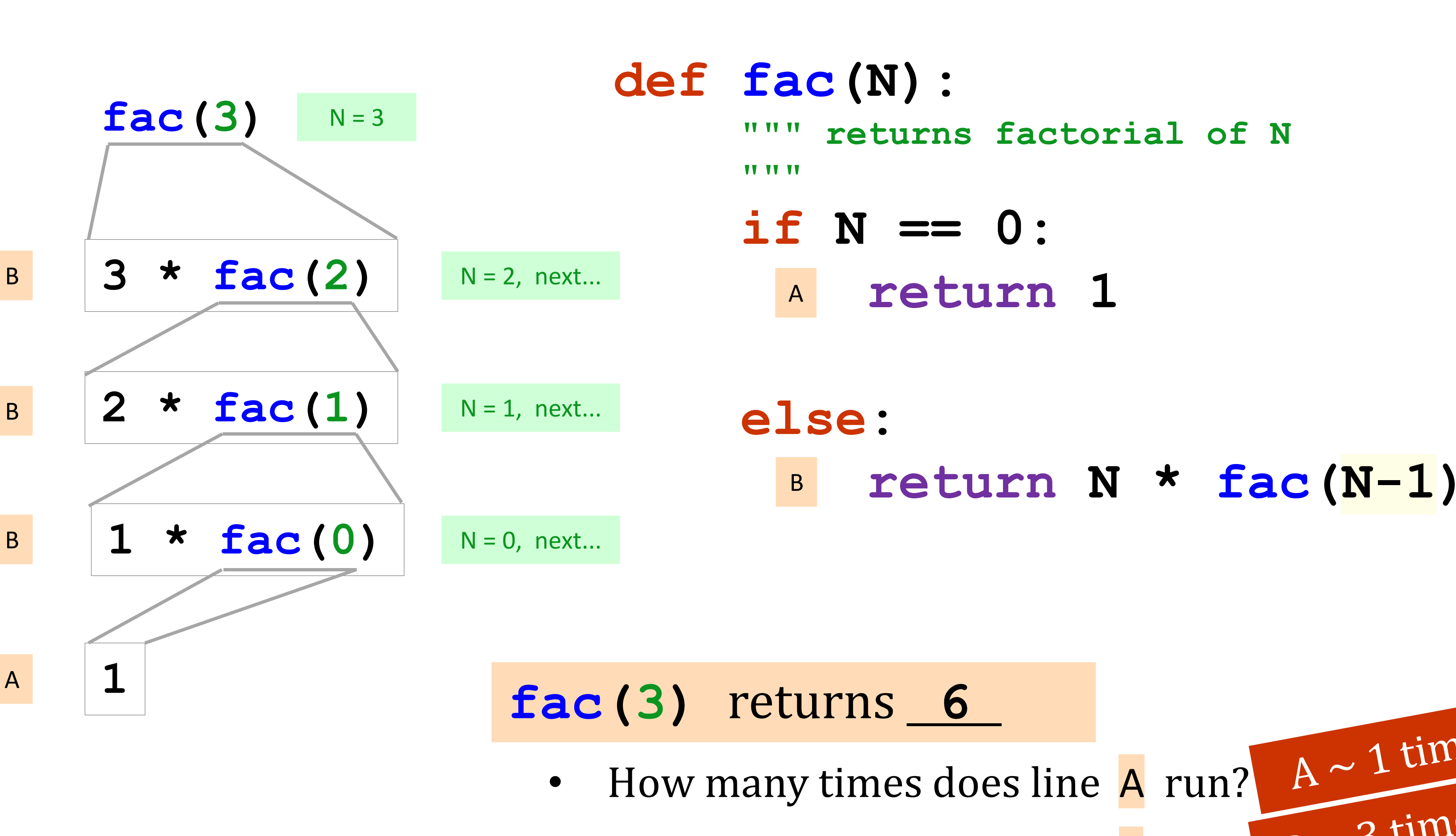
```
def fac(N):
    """ returns factorial of N
    """
    if N == 0:
  A     return 1

    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!



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

pythontutor.com

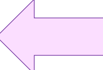
Print output (drag lower right corner to resize)

```
Factorial!
```

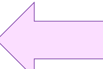
Frames | Objects

Global frame — fac → function fac(N)

fac — N 3

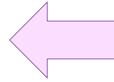


fac — N 2



fac — N 1

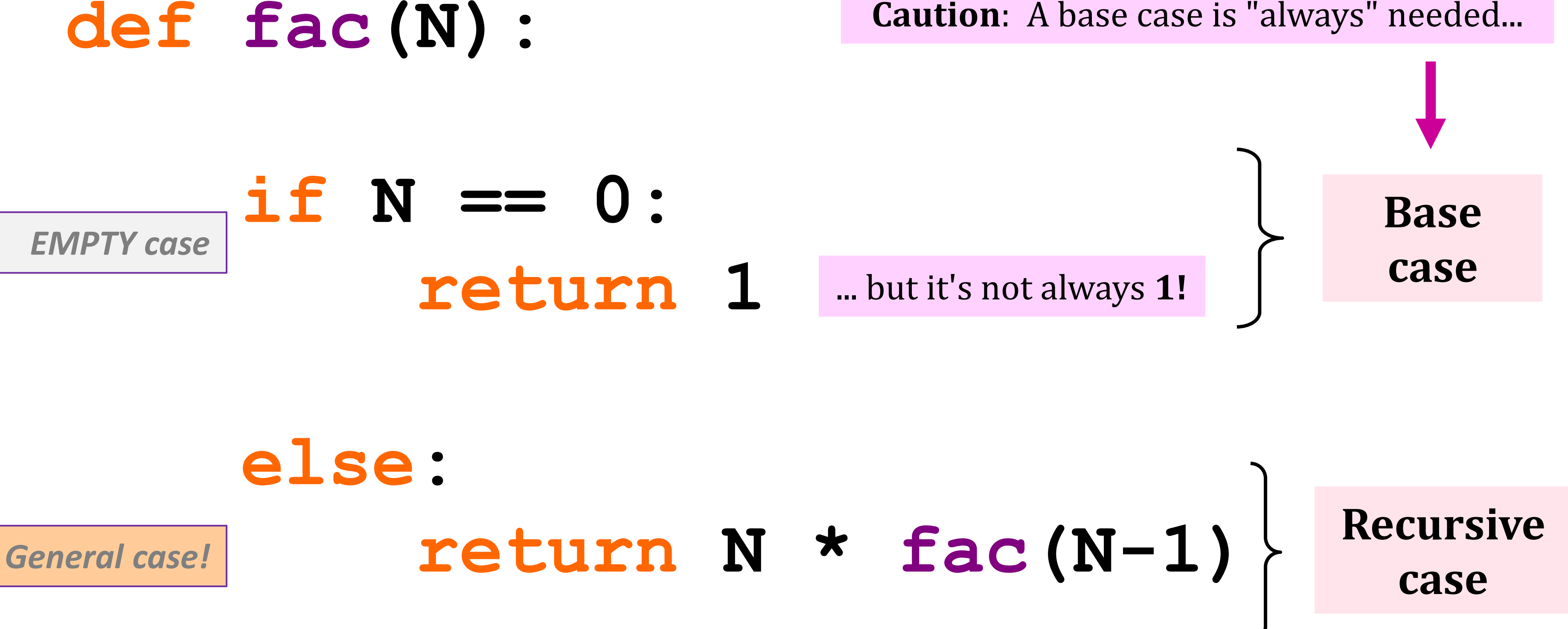
fac — N 0, Return value 1



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

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

Planning recursively...



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

***EMPTY** case*

***BASE** case*

*the empty **integer*** →

*the empty **float*** →

*the empty **string*** →

*the empty **list*** →

Thinking recursively...

```
def fac(N):

    if N == 0:
        return 1

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

EMPTY case

Base case

General 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!

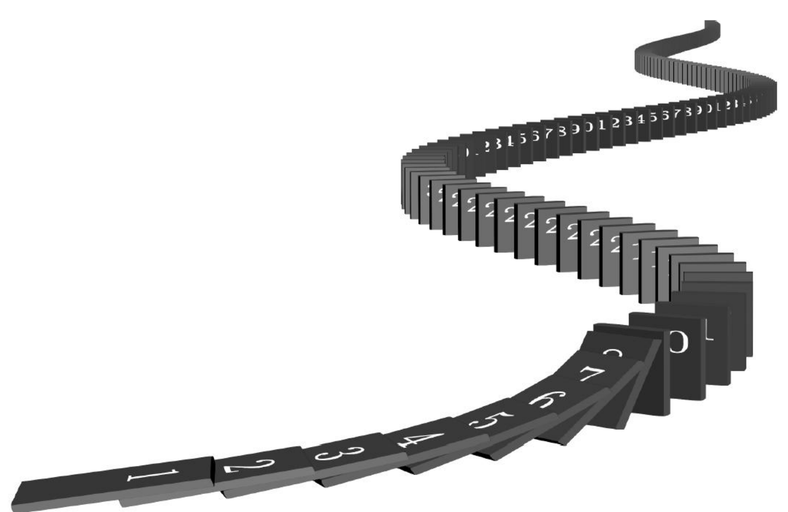
```
def fac(N):

  if N == 0:
    return 1

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

hooray for variables!

Conceptual

Actual

Mathematical Induction



- **Successive** structure;
- Begin with **one item**;
- **Start** at base case;
- Take **inductive steps** to process all items.

Recursion

- **Nested** structure;
- Begin with **all items**;
- **Terminate** at base cases;
- Take **recursive steps** to process all items.

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 = ''

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

is ... a 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

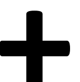
rest

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

the rest-of-S

is `s[0]` a vowel?

+

# 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

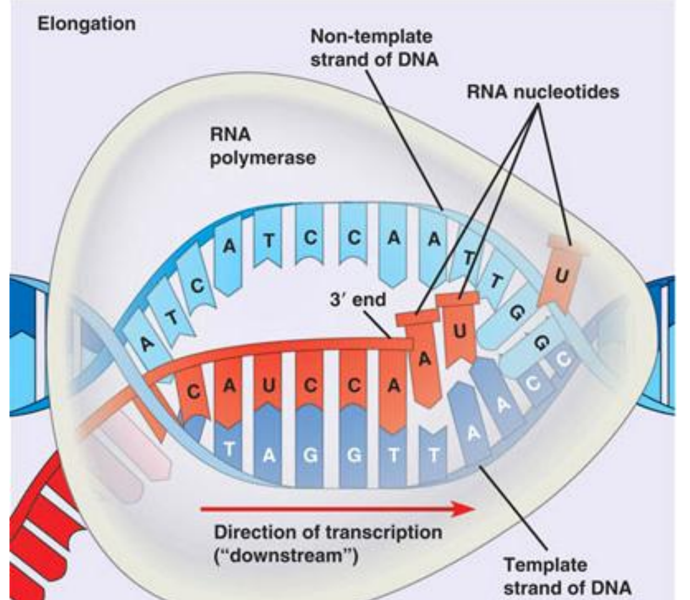
be sure to submit both pr1+pr2...

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

either way: submit pr1 + pr2

complete and submit hw1pr3 + start hw2pr4

Is this Python??



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

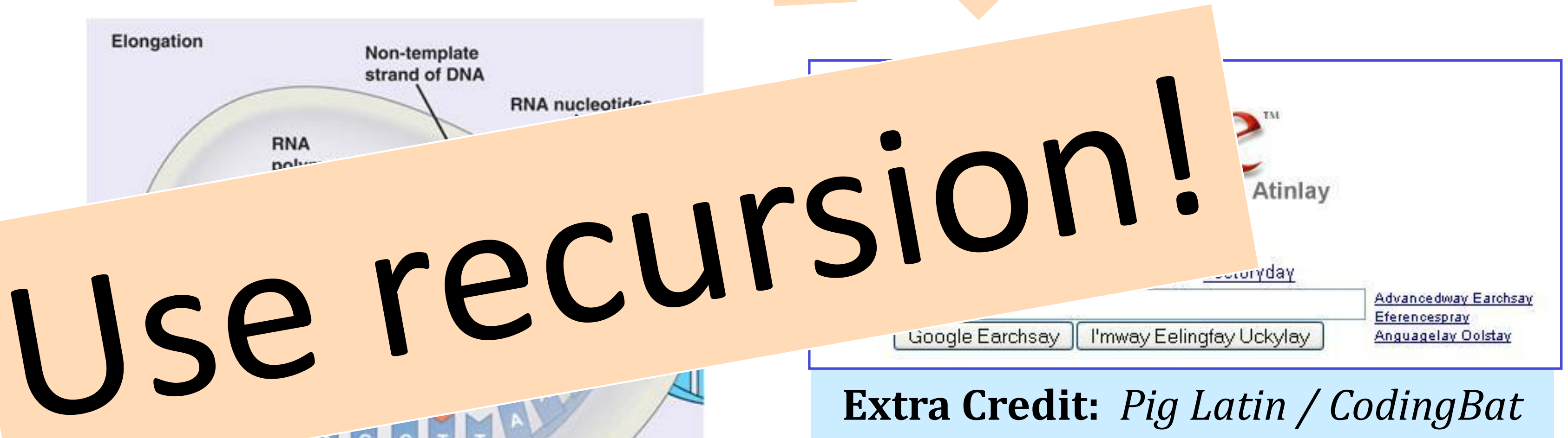
be sure to submit both pr1+pr2...

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

either way: submit pr1 + pr2

complete and submit hw1pr3 + start hw2pr4

Is this Python??



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!

Use PythonBat!

due for week 2

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

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') ?

[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!

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

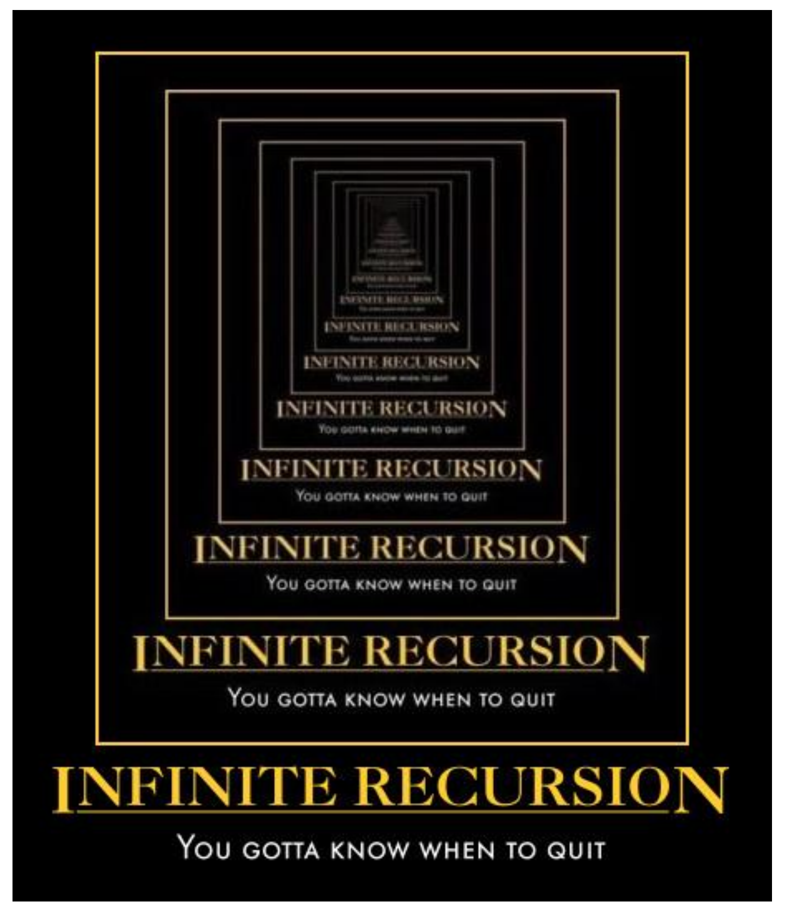
Warning: this code runs!

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

```
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**!

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

sequential

iteration

self-similar

recursion

problem-solving ***paradigms***

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

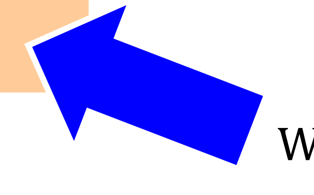
```
fac(5) = 5*4*3*2*1
```

```
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!?

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

- former CS 5 student

It's the eeri**<u>est</u>**!

but that's meant *facetiously*...