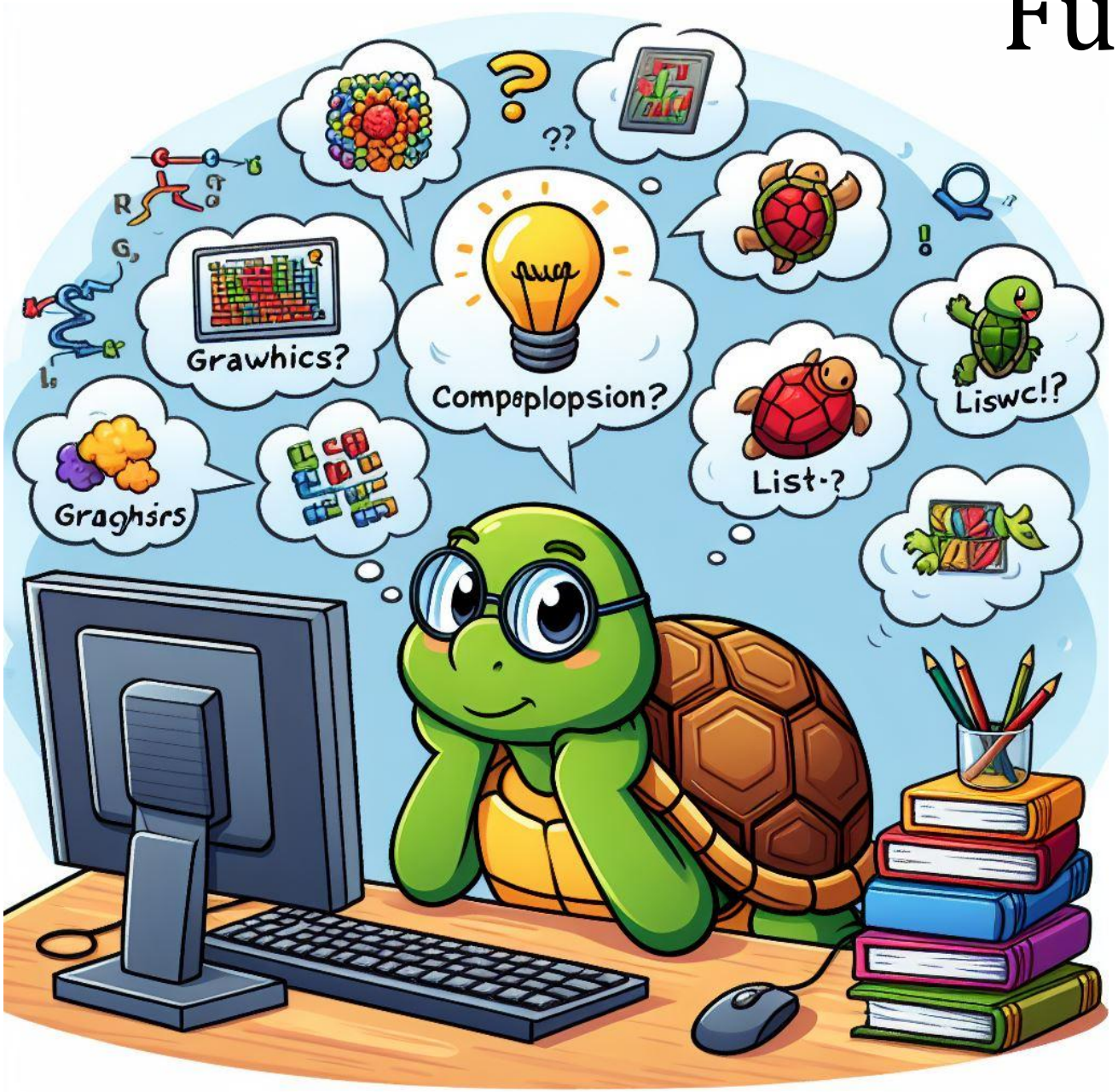
Functions!



Recursion?

Turtles?

Data!

List Comprehensions!

Bourton-on-the-water

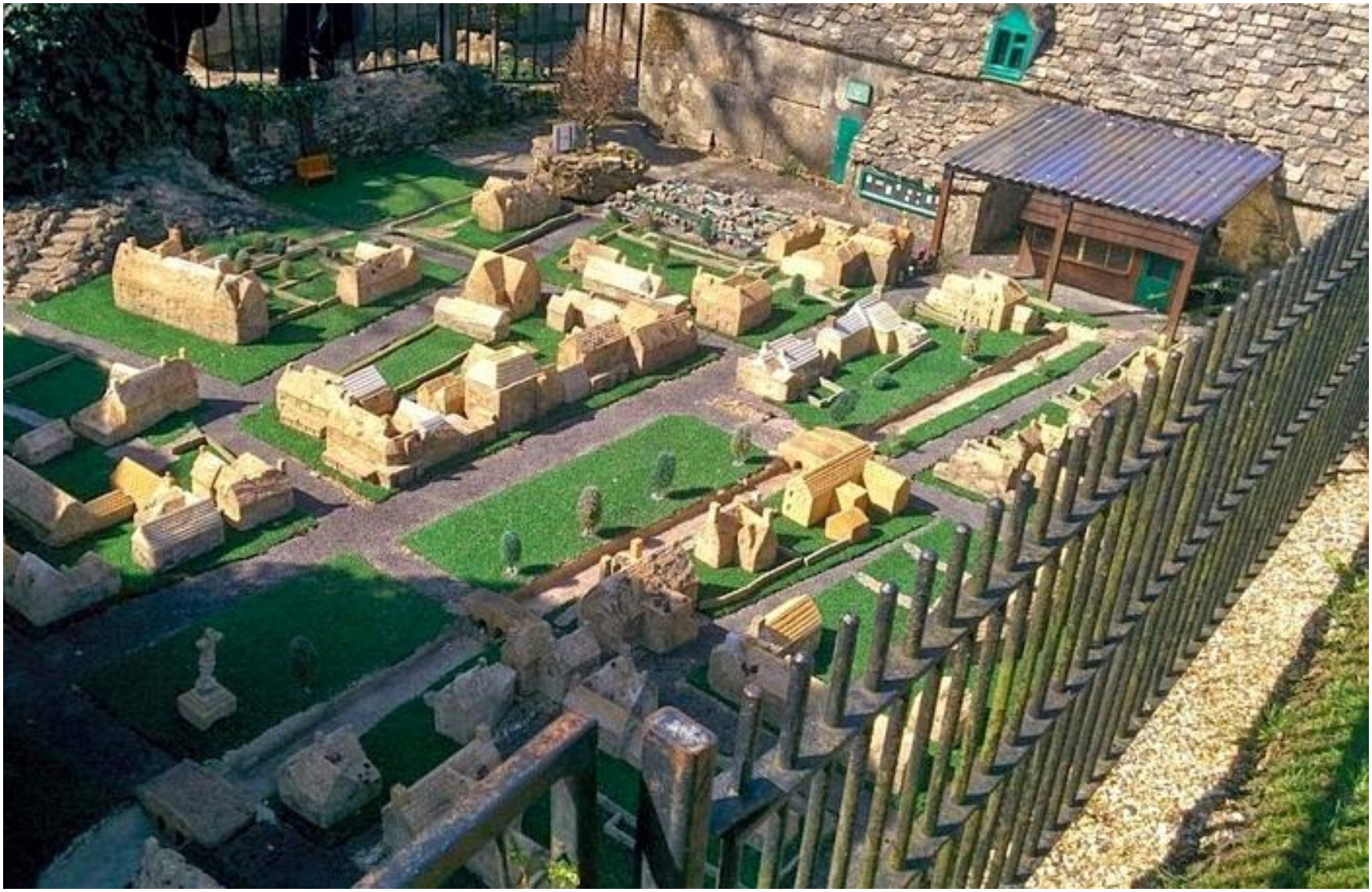
Bourton-on-the-water

Bourton-on-the-water

town of ~2000 people

Bourton-on-the-water's *1/9 model*

has a level-2 model...

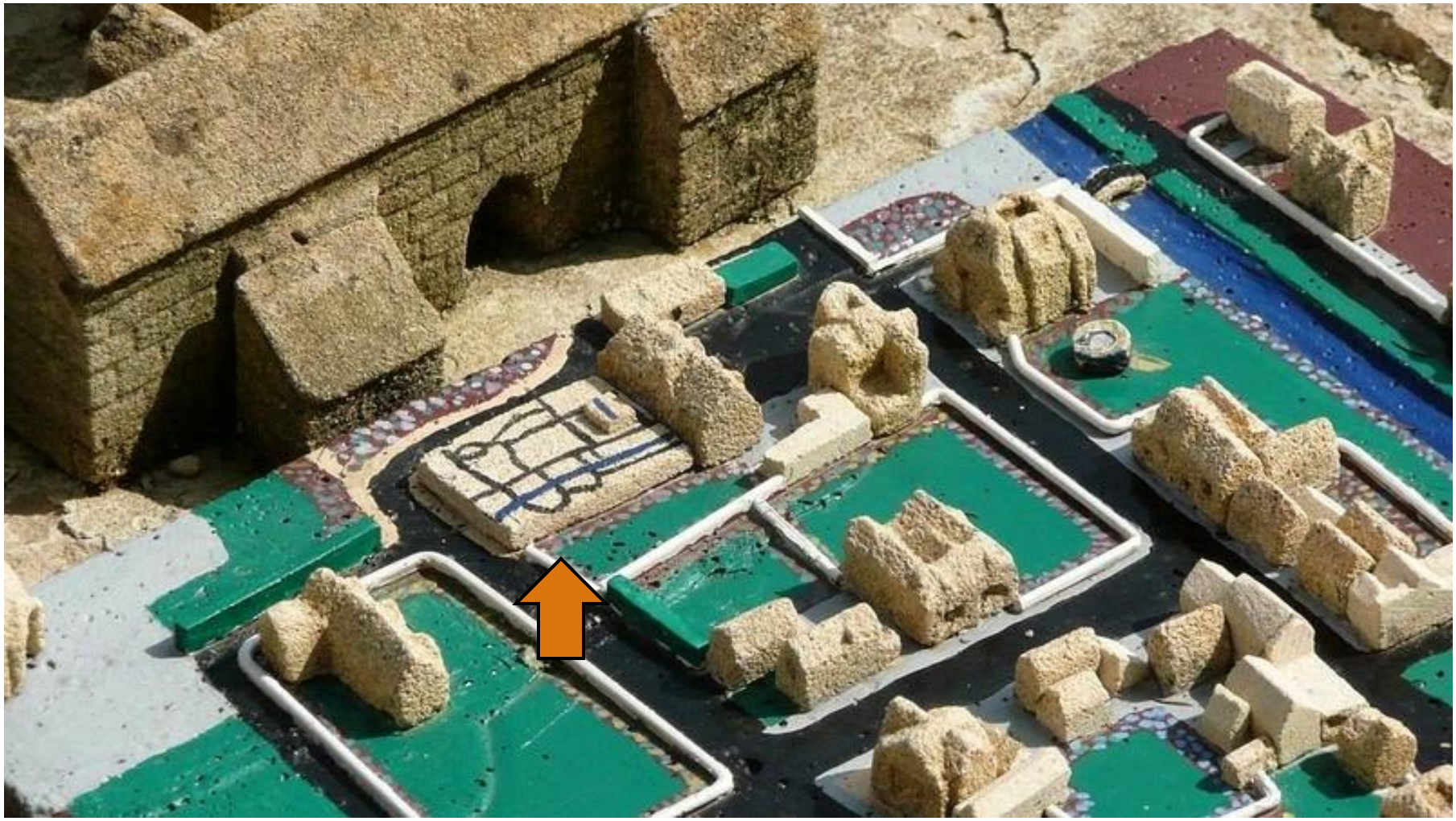


has a level-2 model...

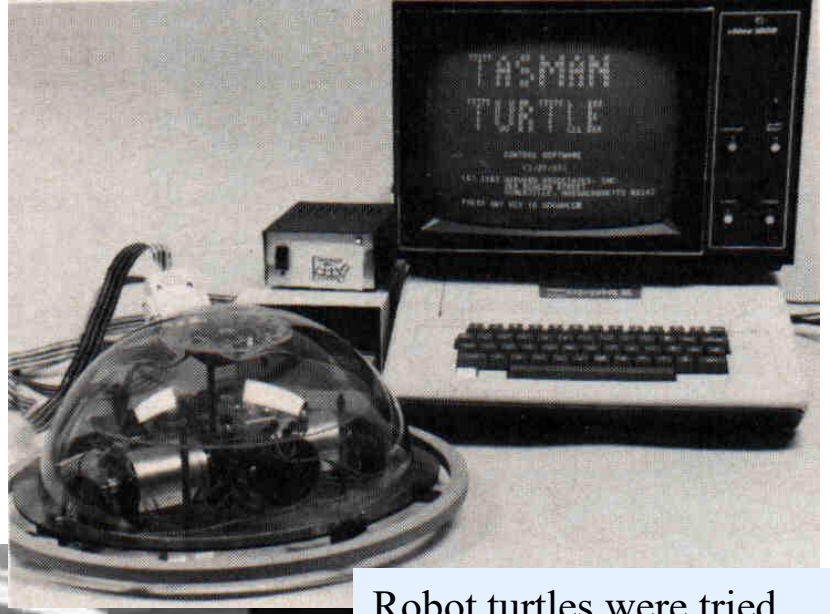
and a level-3 model…

and a level-3 model…

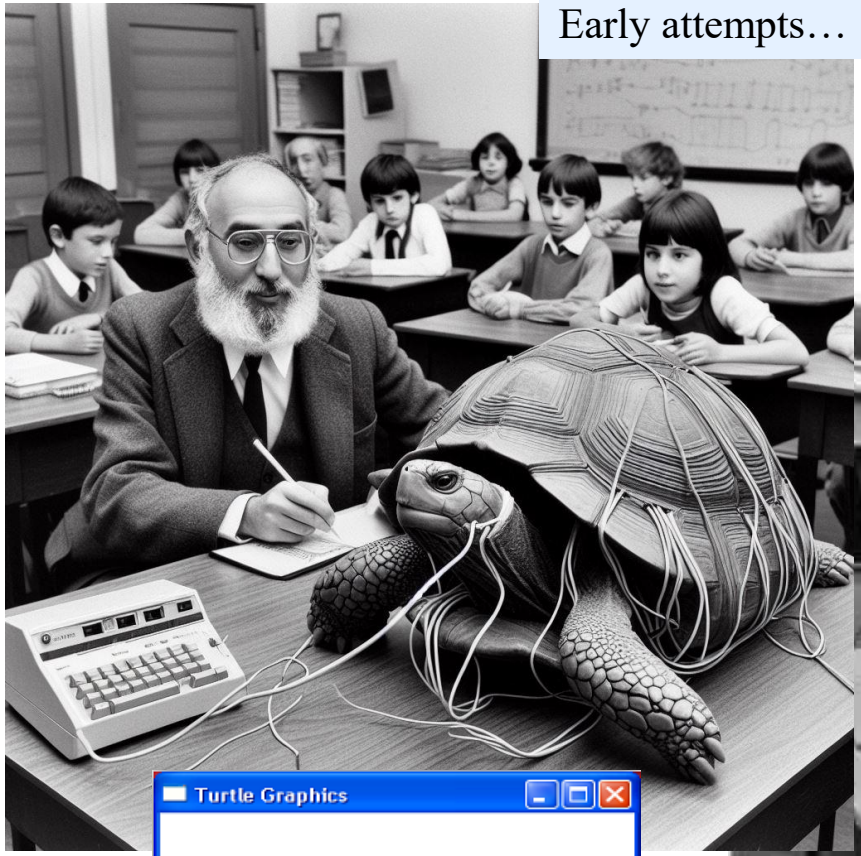
and even a (very small!) level-4 model



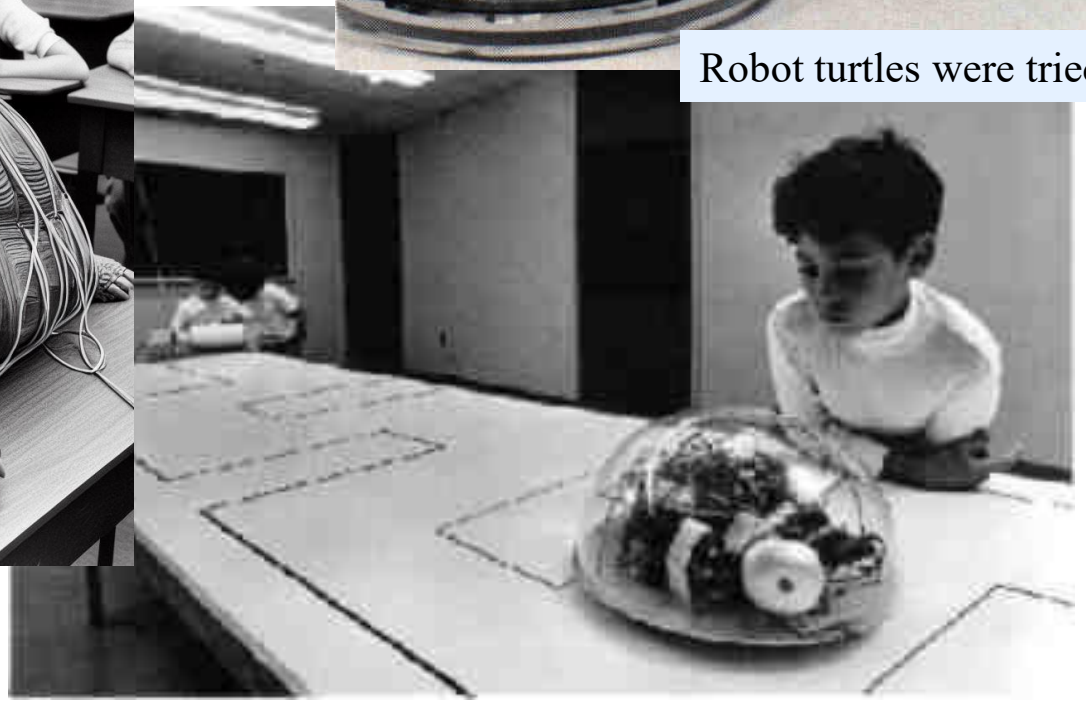
Turtle graphics…

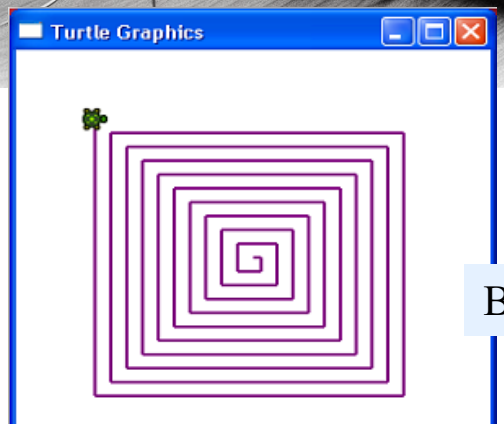


Robot turtles were tried…



Early attempts…





But a computer window was easier…

Something isn't right here...

functional programming

```
>>> 'fun' in 'functional'
True
```

oh my, `in` for strings finds substrings!

Functional programming

- ***functions*** are powerful!
- ***functions*** are "things" just like numbers or strings
- leverage self-similarity ***(recursive code and data)***

Composition & Decomposition

— our lever to solve/investigate problems.

functional programming

```
>>> print(print)
<built-in function print>
>>> exclaim = print
>>> exclaim("By jove!")
By jove!
```

oh my, `in` for strings finds substrings!

Functional programming

- ***functions*** are powerful!
- ***functions*** are "things" just like numbers or strings
- leverage self-similarity ***(recursive code and data)***

Composition & Decomposition

— our lever to solve/investigate problems.

Data

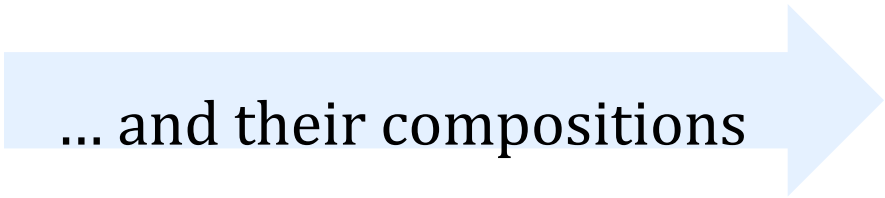
```
[13,14,15]
```

```
[3,4,5,6,7,8,9]
```

Functions

```
sum( )
```

```
range( )
```



sum(L)

list(range(low,hi,stride))

sum range

```
def mysum(L):
    """ input: L, a list of #s
        output: L's sum
    """
    if len(L) == 0:
        return 0.0
    else:
        return L[0] + sum(L[1:])
```

Empty Case

Base Case

Specicfic/General Case

Recursive Case

sum(L)

list(range(low,hi,stride))

sum range

what's cookin' here?

stride?

```
def myrange(low, hi     ):
    """ input:  ints low and hi
        output: list from low to hi
    """
    if low >= hi:
        return
    else:
        return
```

excluding **hi**

Recursion's **range**

myrange(3,7) → [3,4,5,6]

myrange(3,7,2) → [3,5]

We're on target!

```
def myrange(low, hi , stride ):
    """ input:  low and hi, integers
        output: a list from low upto hi
    """
    if low >= hi:
        return
    else:
        return
```

but excluding `hi`

Empty case: What if low is greater than or equal to hi?

Specific/General case: How could we use another call to range to help us?!

Extra! Take a positive third input in `stride`

***Extra* Extra** What if `stride` were negative?

Recursion's **range**

myrange(3,7) → [3,4,5,6]

myrange(3,7,2) → [3,5]

We're on target!

```
def myrange(low, hi , stride ):
    """ input:  low and hi, integers
        output: a list from low upto hi
    """
    if low >= hi:
        return []
    else:
        return low + range(low+1,hi)
```

but excluding `hi`

Empty case: What if low is greater than or equal to hi?

Alllllllllllmost...

Specific/General case: How could we use another call to range to help us?!

Extra! Take a positive third input in `stride`

***Extra* Extra** What if `stride` were negative?

Recursion's **range**

myrange(3,7) → [3,4,5,6]

myrange(3,7,2) → [3,5]

Solution! Try on the other page first!

We're on target!

```
def myrange(low, hi , stride ):
    """ input:  low and hi, integers
        output: a list from low upto hi
    """
    if low >= hi:
        return []
    else:
        return [low] + range(low+1,hi)
```

but excluding `hi`

Extra Extra!! What if `stride` were negative?

we'd use a different test!

Empty case: What if low is greater than or equal to hi?

Specific/General case: How could we use another call to range to help us?!

Extra! Take a positive third input in `stride`

```
[low] + range(low+stride, hi, stride)
```

Let's make some functions...

```
def double_all(L):
    """Takes a list and returns a new list
       with all the elements doubled."""
    if L == []:
        return []
    else:
        first_L = L[0]
        rest_L = L[1:]
        doubled_first = 2 * first_L
        doubled_rest = double_all(rest_L)
        return [doubled_first] + doubled_rest
```

Let's make some functions...

```
def double_all(L):
    """Takes a list and returns a new list
       with all the elements doubled."""
    if L == []:
        return []
    else:
        return [2 * L[0]] + double_all(L[1:])
```

Let's make some functions...

```
def twice(x):
    return 2 * x

def double_all(L):
    """Takes a list and returns a new list
       with all the elements doubled."""
    if L == []:
        return []
    else:
        return [twice(L[0])] + double_all(L[1:])
```

Let's make some functions...

```
def cube(x):
    return x * x * x

def cube_all(L):
    """Takes a list and returns a new list
       with all the elements cubed."""
    if L == []:
        return []
    else:
        return [cube(L[0])]  + cube_all(L[1:])
```

Let's generalize!

```
def apply_to_all(f, L):
    """Takes a function f and a list L and returns
       a new list with f applied to L's elements"""
    if L == []:
        return []
    else:
        return [        ] + apply_to_all(f, L[1:])
```

What goes here?

Let's generalize!

```
def apply_to_all(f, L):
    """Takes a function f and a list L and returns
       a new list with f applied to L's elements"""
    if L == []:
        return []
    else:
        return [ f(L[0]) ] + apply_to_all(f, L[1:])
```

```
def double_all(L):
    return apply_to_all(twice, L)

def cube_all(L):
    return apply_to_all(cube, L)
```

Python *already* has apply_to_all, it's called **map**

Let's make even more functions...

```
def is_even(n):
    return n % 2 == 0

def only_even(L):
    """Takes a list L and returns a new list
       with only the even numbers in L."""
    if L == []:
        return []
    else:
        if is_even(L[0]):
            return [L[0]] + only_even(L[1:])
        else:
            return only_even(L[1:])
```

Let's make even more functions...

```
def is_odd(n):
    return not is_even(n)

def only_odd(L):
    """Takes a list L and returns a new list
       with only the odd numbers in L."""
    if L == []:
        return []
    else:
        if is_odd(L[0]):
            return [L[0]] + only_odd(L[1:])
        else:
            return only_odd(L[1:])
```

Let's generalize!

```
def keep_if(f, L):
    """Takes a function f and a list L and returns
       a new list with only the elements of L
       for which f is true."""
    if L == []:
        return []
    else:
        if
            return [L[0]] + keep_if(f, L[1:])
        else:
            return keep_if(f, L[1:])
```

Let's generalize!

```
def only_even(L):
    return keep_if(is_even, L)

def only_odd(L):
    return keep_if(is_odd, L)
```

```
def keep_if(f, L):
    """Takes a function f and a list L and returns
       a new list with only the elements of L
       for which f is true."""
    if L == []:
        return []
    else:
        if f(L[0]):
            return [L[0]] + keep_if(f, L[1:])
        else:
            return keep_if(f, L[1:])
```

Python *already* has keep_if,
it's called **filter**

Powerful stuff

```
apply_to_all(cube, keep_if(is_odd, [1, 2, 3, 4, 5, 6]))
```

a.k.a.

```
map(cube, filter(is_odd, [1, 2, 3, 4, 5, 6]))
```

Math does it better!

$$S = \{2 \cdot x \mid x \in \mathbb{N},\ x^2 > 3\}$$

This notation is sometimes called a “set comprehension”.

But Python can do it, too…

```
def x2gt3(x):
    return x**2 > 3

S = map(twice, filter(x2gt3, N))
```

Python won't give in *that* easily!

Math does it better!

$$S = \{ \underbrace{2 \cdot x}_{\text{output expression}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{input set}} , \underbrace{x^2 > 3}_{\text{predicate}} \}$$

But Python can do it, too…

```
def x2gt3(x):
    return x**2 > 3

S = map(twice, filter(x2gt3, N))
```

Python won't give in *that* easily!

Math does it better!

$$S = \{ \underbrace{2 \cdot x}_{\text{output expression}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{input set}} , \underbrace{x^2 > 3}_{\text{predicate}} \}$$

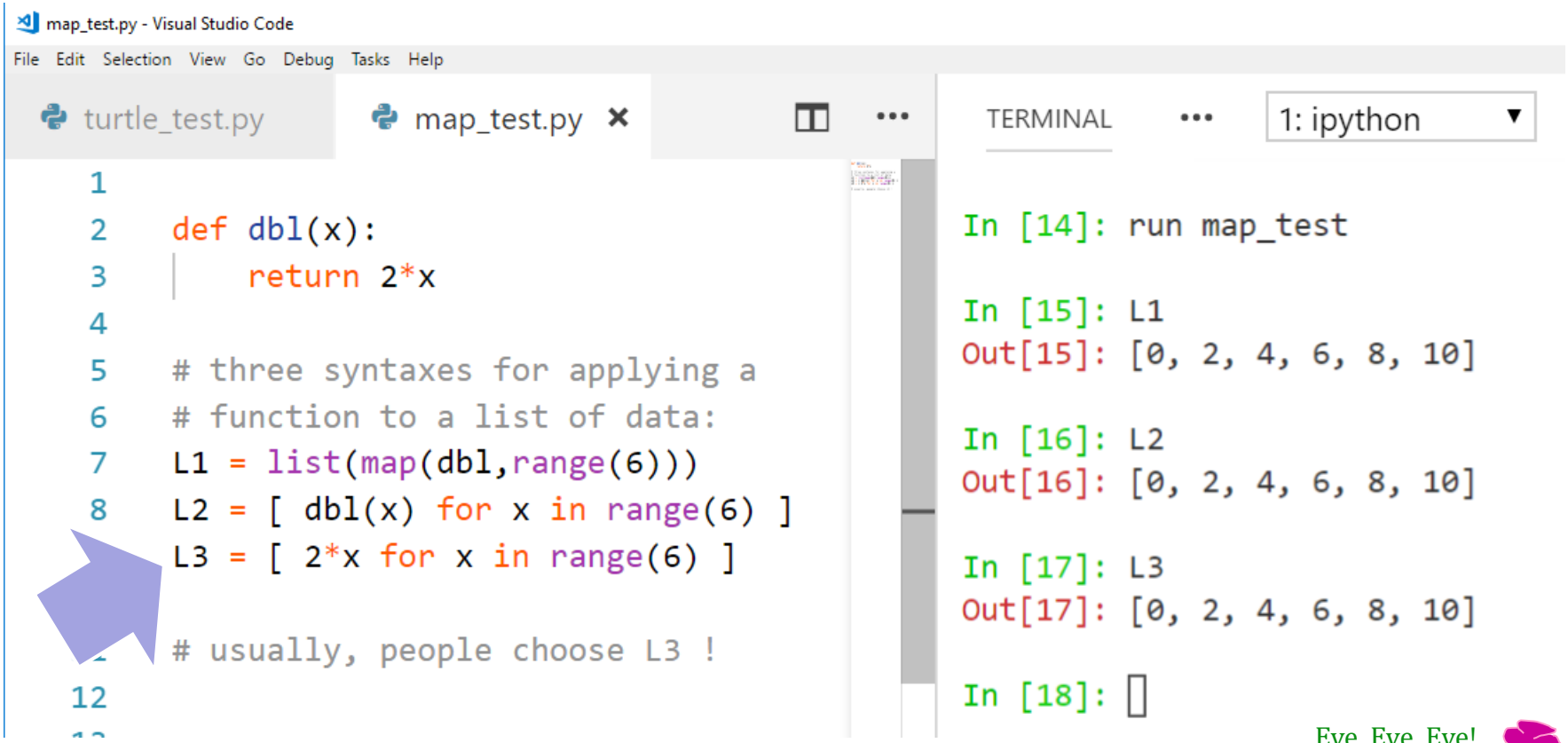
But Python can do it, too…

```
R = [twice(x) for x in N if x2gt3(x)]

# Or, more directly:

R = [2*x for x in N if x**2 > 3]
```

Python won't give in *that* easily!

Various approaches...

many options for mapping a function onto a list:



Eye, Eye, Eye!

List Comprehensions

```
In: [ 2*x for x in [0,1,2,3,4,5] ]
```

List Comprehension

```
[0, 2, 4, 6, 8, 10]
```

result

What's the syntax saying here?

List Comprehensions

Expression to evaluate for each list element

Name for each list element

The **list** - or **string** to *use*

```
In: [ 2*x for x in [0,1,2,3,4,5] ]
```

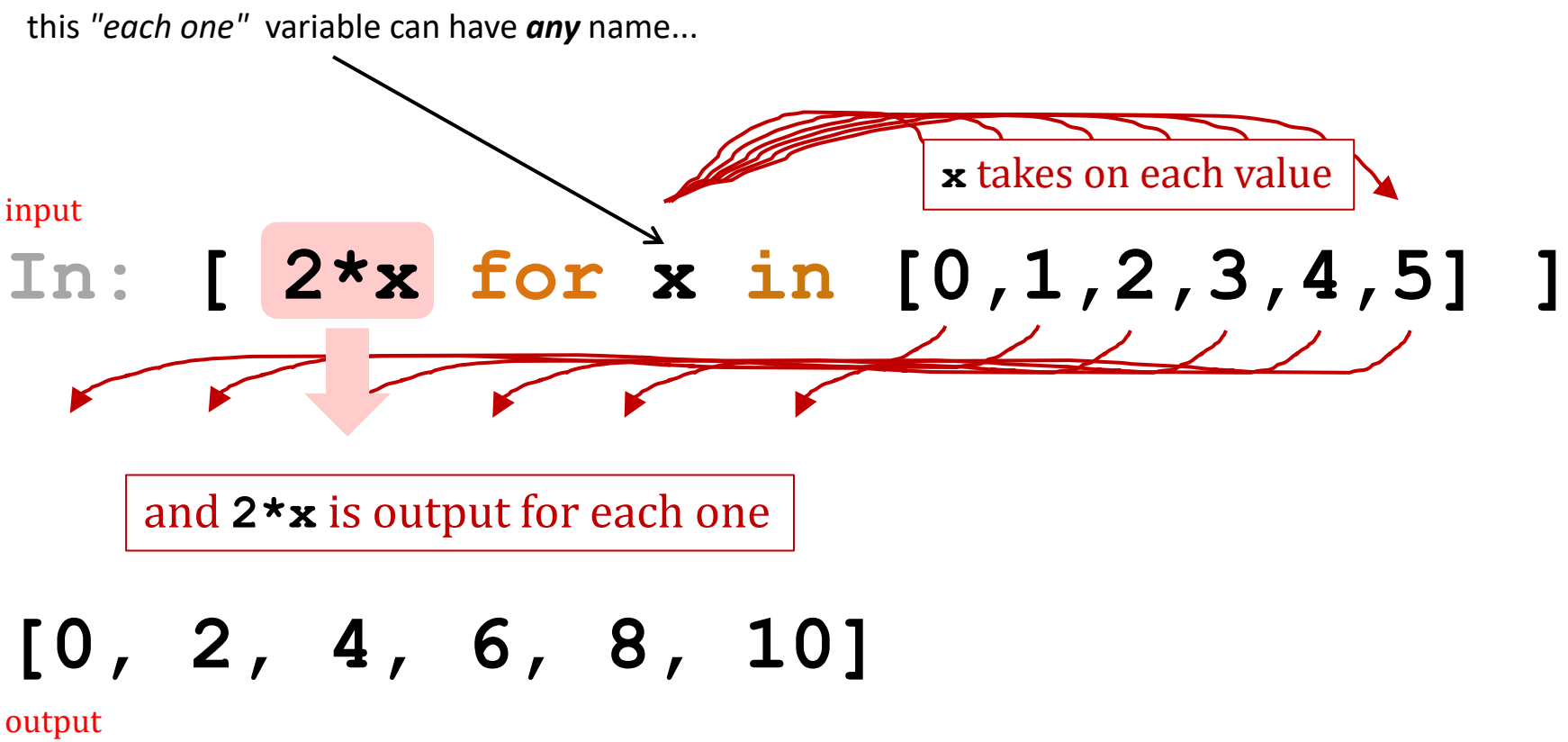
List Comprehenion

```
[0, 2, 4, 6, 8, 10]
```

result

What's the syntax saying here?

List Comprehensions



List Comprehensions

expression iteration condition

```
In: [ 10*x for x in [0,1,2,3,4,5] if x%2==0]
```

result

```
In: [ y*21 for y in range(0,3) ]
```

result

```
In: [ s[1] for s in ["hi", "5Cs!"] ]
```

result

Names: _____

Write Python's result for each LC:

```
[ n**2 for n in range(0,4) ]
```

[0,1,2,3]

list

A **range** of list comprehensions

Try them ~~out~~ **in**!

```
[ s[1::2] for s in ['aces','451!'] ]
```

```
[ -7*b for b in range(-6,6) if abs(b)>4 ]
```

```
[ a*(a-1) for a in range(8) if a%2==1 ]
```

Watch out !!!

```
[ z for z in [0,1,2] ]
[ 42 for z in [0,1,2] ]
[ 'z' for z in [0,1,2] ]
```

Got it!

But what about that *name*?

Names: _____

Write Python's result for each LC:

```
[ n**2 for n in range(0,4) ]
```

list [0,1,2,3]

[0,1,4,9]

A **range** of list comprehensions

Try them ~~out~~ in!

```
[ s[1::2] for s in ['aces','451!'] ]
```

```
[ -7*b for b in range(-6,6) if abs(b)>4 ]
```

```
[ a*(a-1) for a in range(8) if a%2==1 ]
```

Watch out !!!

```
[ z for z in [0,1,2] ]
[ 42 for z in [0,1,2] ]
[ 'z' for z in [0,1,2] ]
```

Got it!

But what about that *name*?

Write Python's result for each LC:

A **range** of list comprehensions

Try them ~~out~~ in!

```
[ n**2 for n in range(0,4) ]
```

[0,1,2,3]

list

[0,1,4,9]

```
[ s[1::2] for s in ['aces','451!'] ]
```

['cs','5!']

```
[ -7*b for b in range(-6,6) if abs(b)>4 ]
```

[-6, -5, 5]

[42,35,-35]

```
[ a*(a-1) for a in range(8) if a%2==1 ]
```

[1, 3, 5, 7]

[0,6,20,42]

Watch out !!!

```
[ z for z in [0,1,2] ]
```

[ ]

```
[ 42 for z in [0,1,2] ]
```

[ ]

```
[ 'z' for z in [0,1,2] ]
```

[ ]

Got it!

But what about that *name*?

Write Python's result for each LC:

A **range** of list comprehensions

Try them ~~out~~ in!

```
[ n**2 for n in range(0,4) ]
```

[0,1,2,3]

list

[0,1,4,9]

```
[ s[1::2] for
```

```
[ 0,1,2 ]
```

```
[ 42 for z in [0,1,2] ]
```

[42,42,42]

```
[ 'z' for z in [0,1,2] ]
```

['z','z','z']

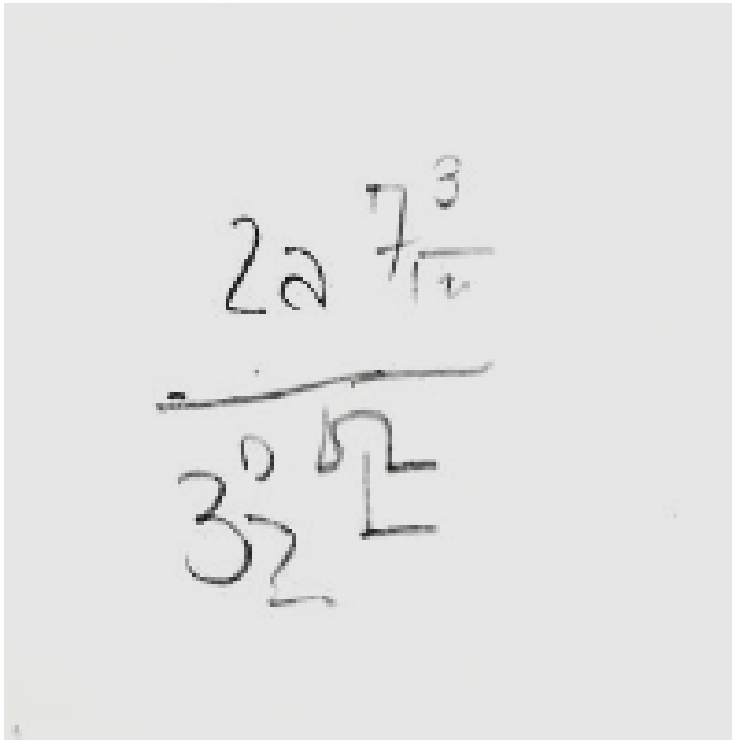
Watch out !!!

Hand these in, north-ward!

heliotropically!

Syntax ?!

```
>>> [ 2*x for x in [0,1,2,3,4,5] ]
[0, 2, 4, 6, 8, 10]
```

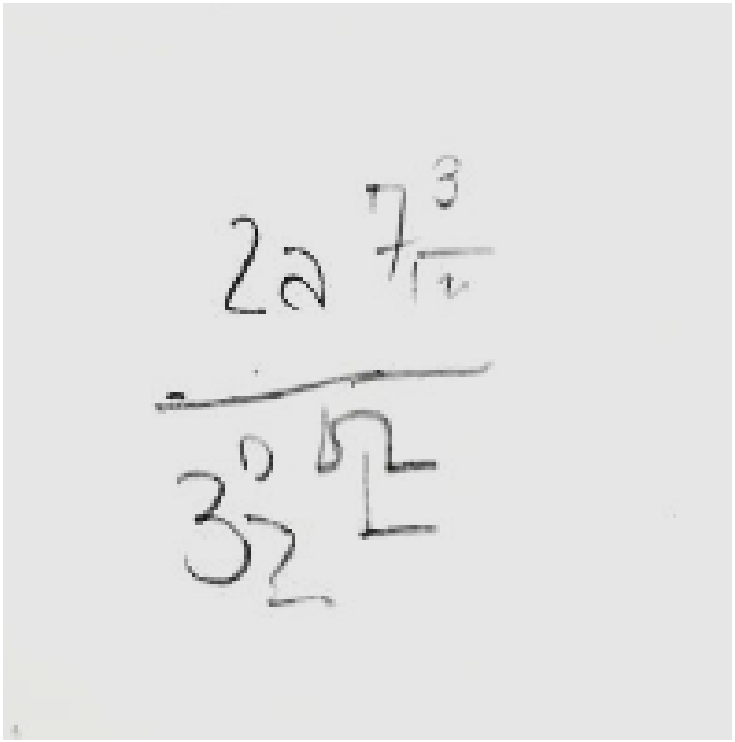


at first…

a jumble of characters
and random other stuff

a (frustrated!) rendering of
an unfamiliar math problem

Syntax ~ is CS's key resource!



a (frustrated!) rendering of
an unfamiliar math problem

(c) $\dfrac{12xy^4}{18x^3y^2}$

$$\frac{4\sqrt{x}\cdot\sqrt[3]{a}}{3\sqrt{x}\cdot\sqrt[4]{x^3}}$$

Where'd the change happen?

which was likely
similar to these…

Designing with LCs, sum, and range...

Key idea:

```
LC = [ 1 for c in 'i get it!' if c=='i' ]

answer = sum(LC)
```

What's `LC` here?

What number is `answer`?

What *question* is `answer` *answering*?!

Designing with LCs, sum, and range...

Key idea:

```
LC = [ 1 for c in 'i get it!' if c=='i' ]
```

```
[1,1]
```

What's `LC` here?

```
answer = sum(LC)
```

2

What number is `answer`?

How many `i`'s are in `'i get it'` ?

What *question* is `answer` *answering*?!

Two fun:

Short and sweet!

```
def fun1(L):
    LC = [1 for x in L]
    return sum( LC )
```

[7,8,9]

```
def letScore(c):
```

from hw1pr3

```
def fun2(S):
    LC = [letScore(c) for c in S]
    return sum( LC )
```

'twelve'

What fun are these?

Two fun:

len

[7,8,9]

```
def fun1(L):
    LC = [1 for x in L]
    return sum( LC )
```

But ***one-liners*** are my specialty...

```
def letScore(c):
```
from hw1pr3

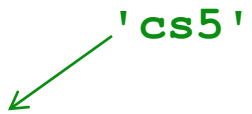
scrabbleScore

'twelve'

```
def fun2(S):
    LC = [letScore(c) for c in S]
    return sum( LC )
```

What fun are these?

"One-line" LCs




```
def len(L):
    LC = [1 for x in L]
    return sum( LC )
```

possible in 1 line, but ***not*** recommended!

I never get more than one line – who are the writers around here... ?

"One-line" LCs

'cs5'

```
def len(L):
    LC = [1 for x in L]
    return sum( LC )
```

possible in 1 line, but ***not*** recommended!

That's no one-liner!

```
def len(L):
    return sum([1 for x in L])
```

Maybe **too** short!

# of vowels

'sequoia'

if

```
def vwl(s):
    LC = [1 for c in s                ]
    return sum( LC )
```

# of times e is in L

42

[3,42,5,7,42]

```
def count(e,L):
    LC = [1 for x in L                ]
    return sum( LC )
```

if

# of vowels

'sequoia'

if

```
def vwl(s):
    LC = [1 for c in s if c in 'aeiou']
    return sum( LC )
```

# of times e is in L

42

[3,42,5,7,42]

```
def count(e,L):
    LC = [1 for x in L if x == e]
    return sum( LC )
```

if

Write each of these functions using list comprehensions...

input: **L**, any list of #s
output: the # of odd #s in **L**
example: nodds([3,4,5,7,42]) == 3

```
def nodds(L):
    LC = [ 1 for x in L if _____ ]
    return sum(LC)
```

Y are your #s **W** are the winning #s
inputs: **Y** and **W**, two lists of "lottery" numbers (ints)
output: the # of matches between **Y** & **W**
example: lotto([5,7,42,47] , [3,5,7,44,47]) == 3

```
def lotto(Y,W):
    LC = [ 1 for
    return sum(LC)
```

input: **x**, an int >= 2
output: the # of positive divisors of **x**
example: numdivs(12) == 6 (1,2,3,4,6,12)

```
def ndivs(x):
    LC = [ 1 for
    return sum(LC)
```

input: **P**, an int >= 2
output: the list of prime #s up to + incl. **P**
example: primesUpTo(12) == [2,3,5,7,11]

Whoa!

```
def primesUpTo(P):
    LC = [
    return LC
```

Write each of these functions using list comprehensions...

input: **L**, any list of #s
output: the # of odd #s in **L**
example: nodds([3,4,5,7,42]) == 3

```
def nodds(L):
    LC = [ 1 for x in L if x%2 == 1    ]
    return sum(LC)
```

Y are your #s **W** are the winning #s
inputs: **Y** and **W**, two lists of "lottery" numbers (ints)
output: the # of matches between **Y** & **W**
example: lotto([5,7,42,47] , [3,5,7,44,47]) == 3

```
def lotto(Y,W):
    LC = [ 1 for x in Y if x in W ]
    return sum(LC)
```

input: **N**, an int >= 2
output: the # of positive divisors of **N**
example: numdivs(12) == 6 (1,2,3,4,6,12)

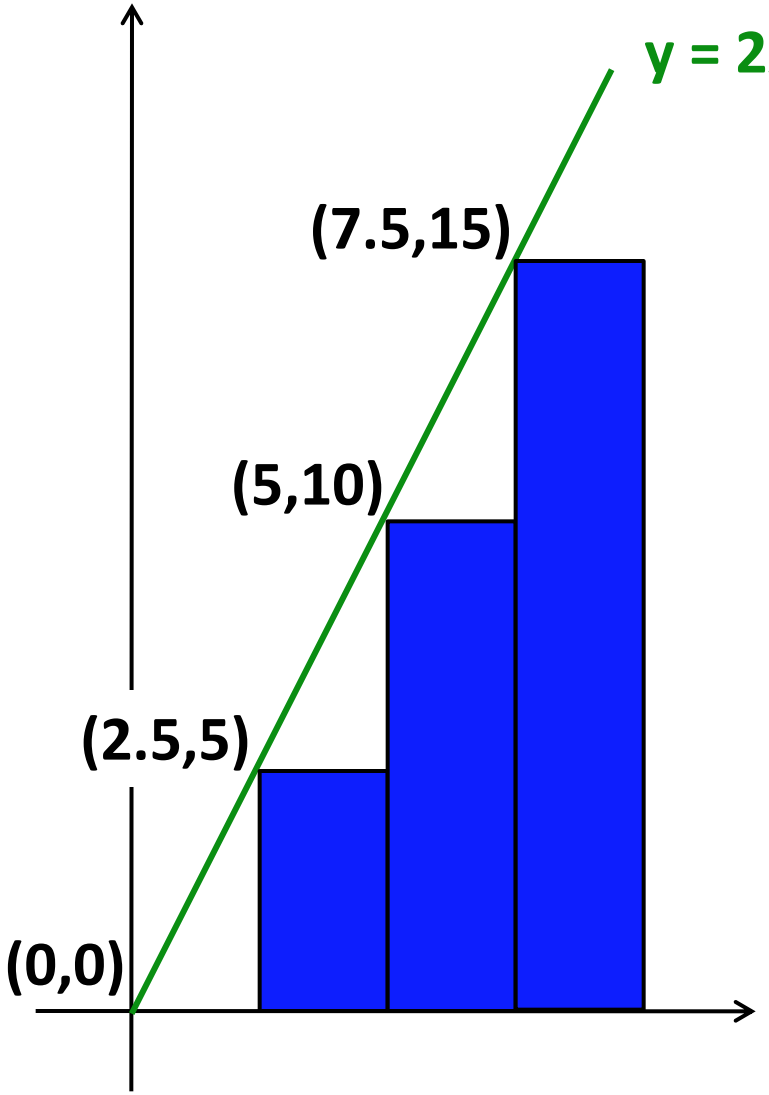
```
def ndivs(N):
    LC = [ 1 for x in range(1,N+1) if N%x == 0    ]
    return sum(LC)
```

input: **P**, an int >= 2
output: the list of prime #s up to + incl. **P**
example: primesUpTo(12) == [2,3,5,7,11]

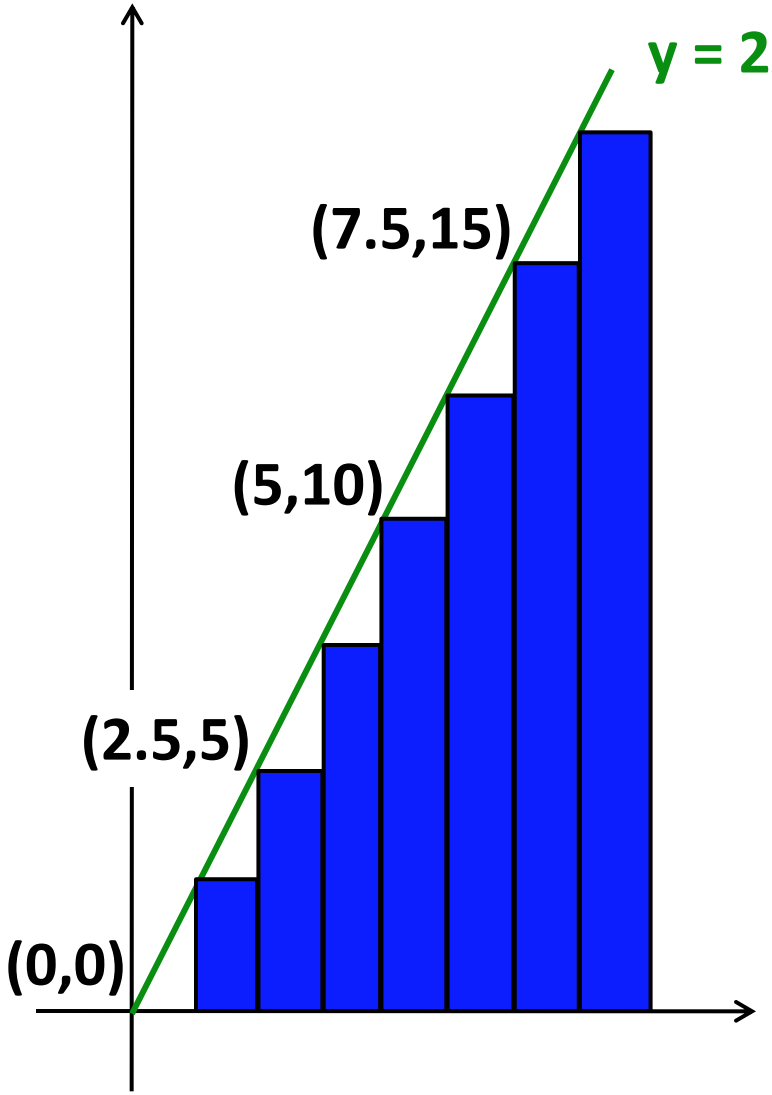
Whoa!

```
def primesUpTo(P):
    LC = [ x for x in range(2,P+1) if ndivs(x)==2 ]
    return LC
```

hw2pr3: *areas from rectangles*

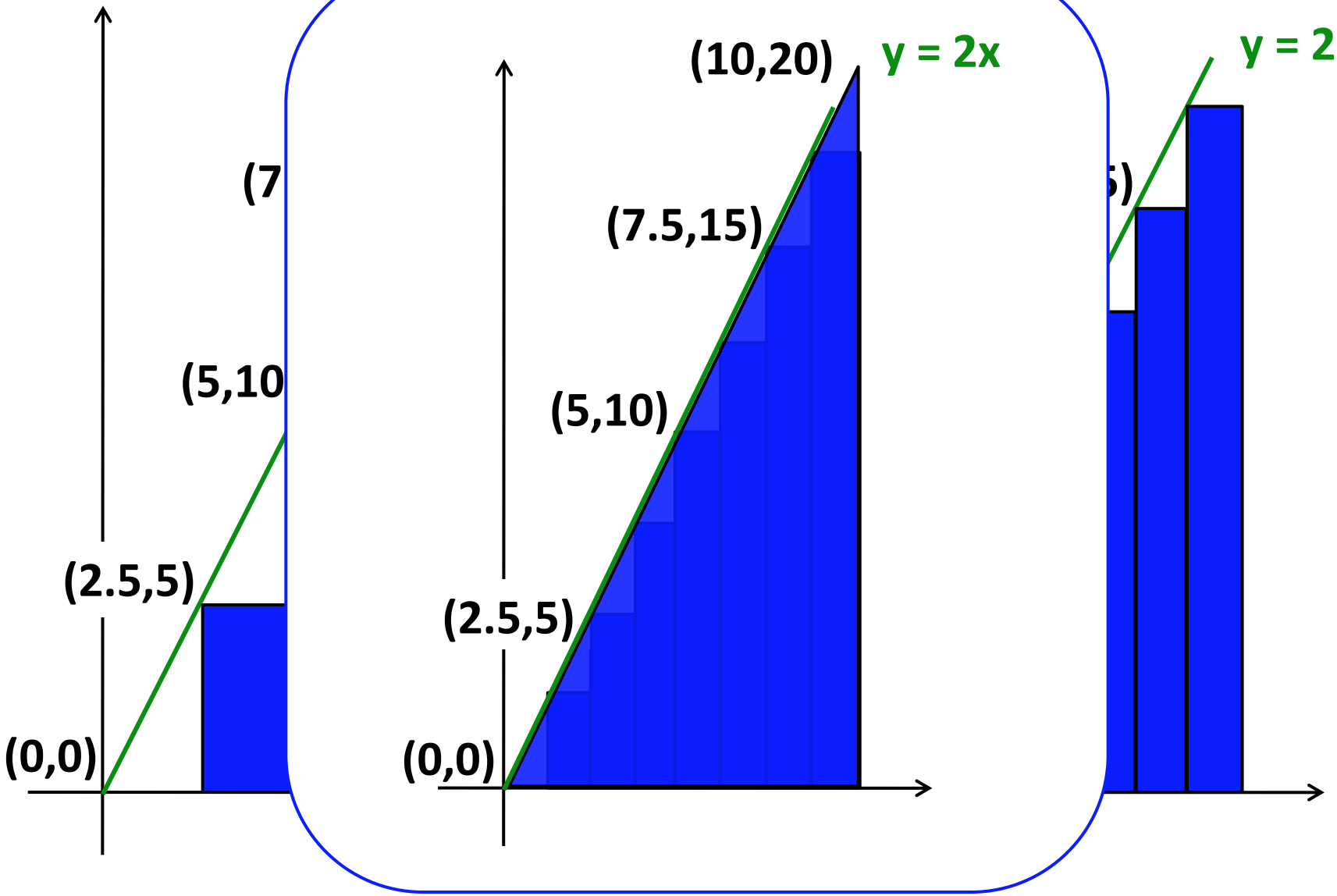


Areas of 4 rectangles



Areas of 8 rectangles

hw2pr3: *areas from rectangles*



Area of N rectangles *in the limit*

Maya Lin, *Artist and Computer Scientist…*

"two-by-four landscape"

hw2pr3: Maya Lin, *Architect…*

Maya Lin, *Artist and Computer Scientist…*

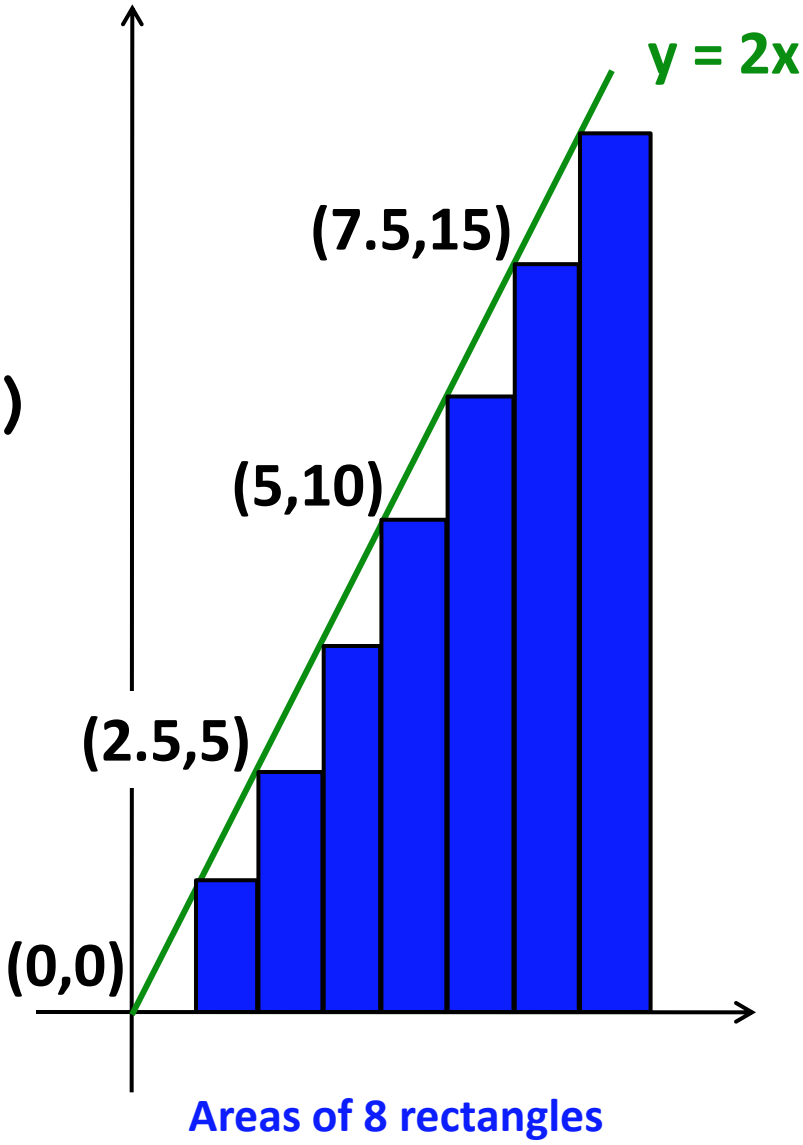
"two-by-four landscape"

CS ~ Building Blocks!

```
scaledfracs(low,hi,N)

f_of_fracs(f,low,hi,N)

integrate(f,low,hi,N)
```

only a few lines...

They're all LCs!

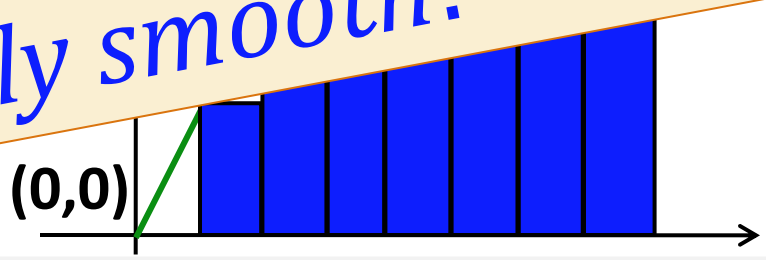


CS ~ Building Blocks!

Wander well

via hw#2...

... may this and all your weekends be *syntactically smooth*!



Next? Coffee! ;-)