Ex. Cr. #1: *Read-it-and-weep*

```
       1
      11
      21
    1211
  111221
  312211
13112221

     ...
```

In the limit, the length of the Nth term of the read-it-and-weep sequence is

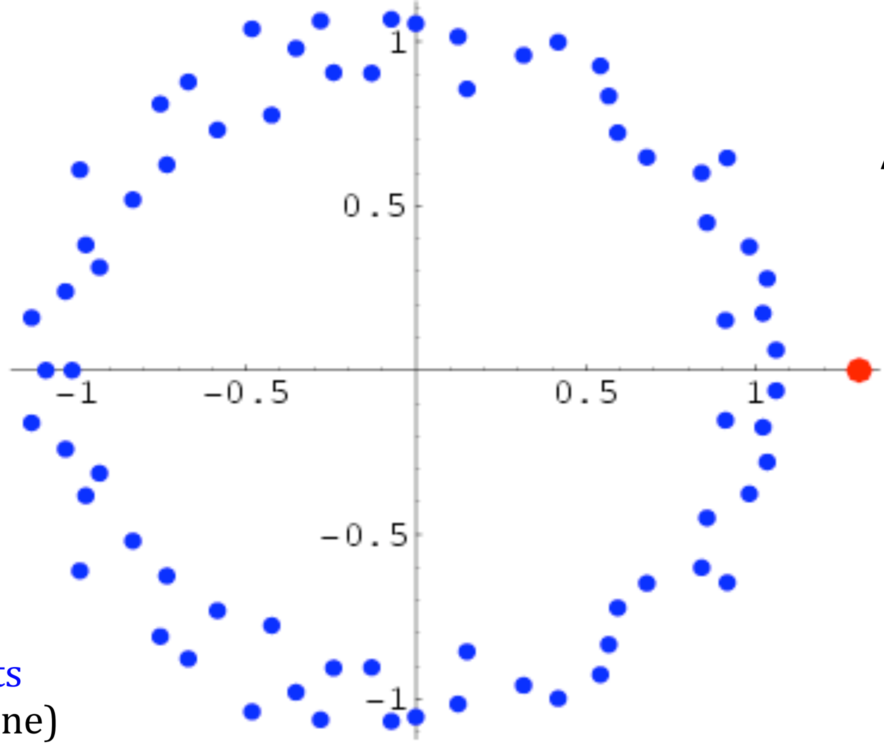
$$(1.303577...)^N$$

exponential growth

this base was found computationally by taking repeated ratios of term lengths...

experimentally found growth...

Growth determined analytically...



the 71 roots
(complex plane)

$\lambda = 1.30357726034296\ldots$

"Conway's Constant" has an ***analytic*** definition!

It is the largest real root of this ***71st-degree*** polynomial !!

$$x^{18}(x+1)(x-1)^2\left(x^{71} - x^{69} - 2x^{68} - x^{67} + 2x^{66} + 2x^{65} + x^{64} - x^{63} - x^{62} - x^{61} - x^{60} - x^{59} + 2x^{58} + 5x^{57} + 3x^{56} - 2x^{55} - 10x^{54} - 3x^{53} - 2x^{52} + 6x^{51} + 6x^{50} + x^{49} + 9x^{48} - 3x^{47} - 7x^{46} - 8x^{45} - 8x^{44} + 10x^{43} + 6x^{42} + 8x^{41} - 5x^{40} - 12x^{39} + 7x^{38} - 7x^{37} + 7x^{36} + x^{35} - 3x^{34} + 10x^{33} + x^{32} - 6x^{31} - 2x^{30} - 10x^{29} - 3x^{28} + 2x^{27} + 9x^{26} - 3x^{25} + 14x^{24} - 8x^{23} - 7x^{21} + 9x^{20} + 3x^{19} - 4x^{18} - 10x^{17} - 7x^{16} + 12x^{15} + 7x^{14} + 2x^{13} - 12x^{12} - 4x^{11} - 2x^{10} + 5x^{9} + x^{7} - 7x^{6} + 7x^{5} - 4x^{4} + 12x^{3} - 6x^{2} + 3x - 6\right).$$

This seems frightening...!

http://www.njohnston.ca/2010/10/a-derivation-of-conways-degree-71-look-and-say-polynomial/

In CS, ***rules*** rule!

(1a) Lists are handled "by **reference**"

(1b) Numbers and strings are handled "by **value**"

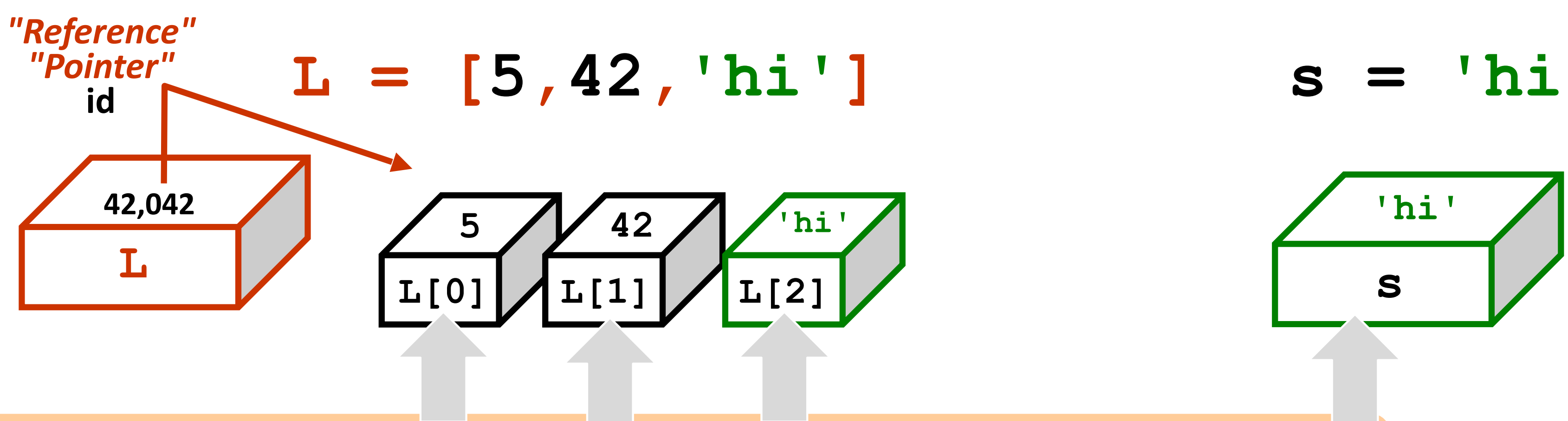
(2) Functions receive inputs "by copy"

two rules,

"interesting" results!

In CS, ***rules*** rule!

(1a) Lists are handled "by **reference**"



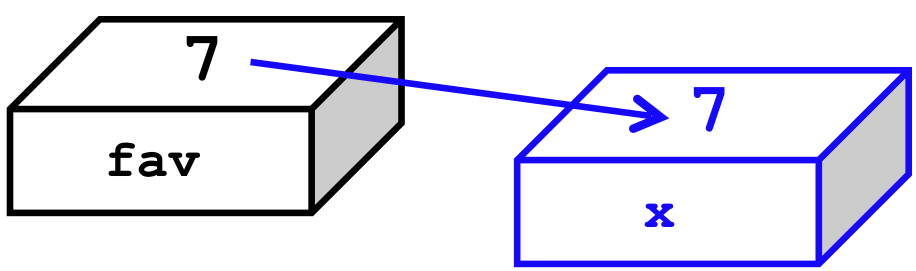
(1b) Numbers and strings are handled "by **value**"

(2) Functions receive inputs "by copy"

The contents of the variable's "box" in memory are <u>copied</u>.

in main:

```
fav = 7
f(fav)
```

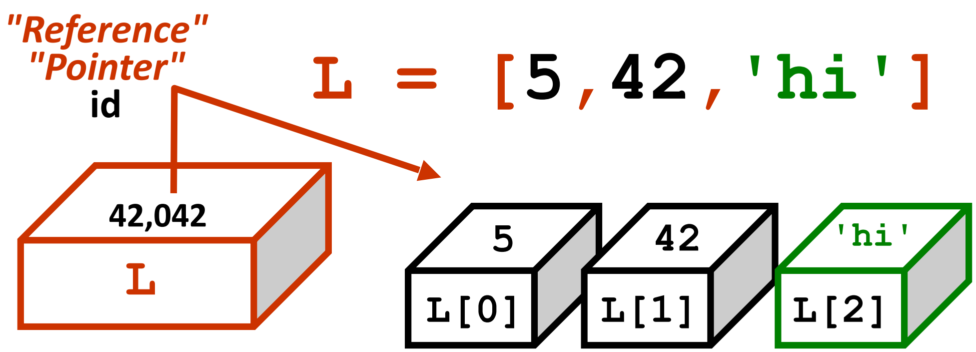



```
def f(x):
```

`x` is a copy of **`fav`**

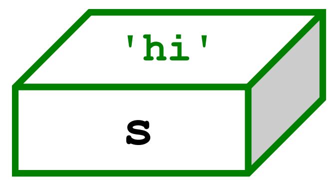
Reference vs. Value

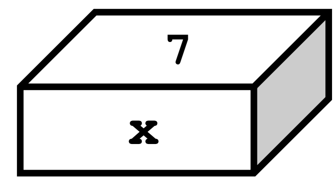
Python's two methods for handling data



Lists are handled **by reference**:
`L` really holds a *memory address*

`s = 'hi'`   `x = 7`





Numeric data and **strings** are handled **by value**: imagine they *hold* the data

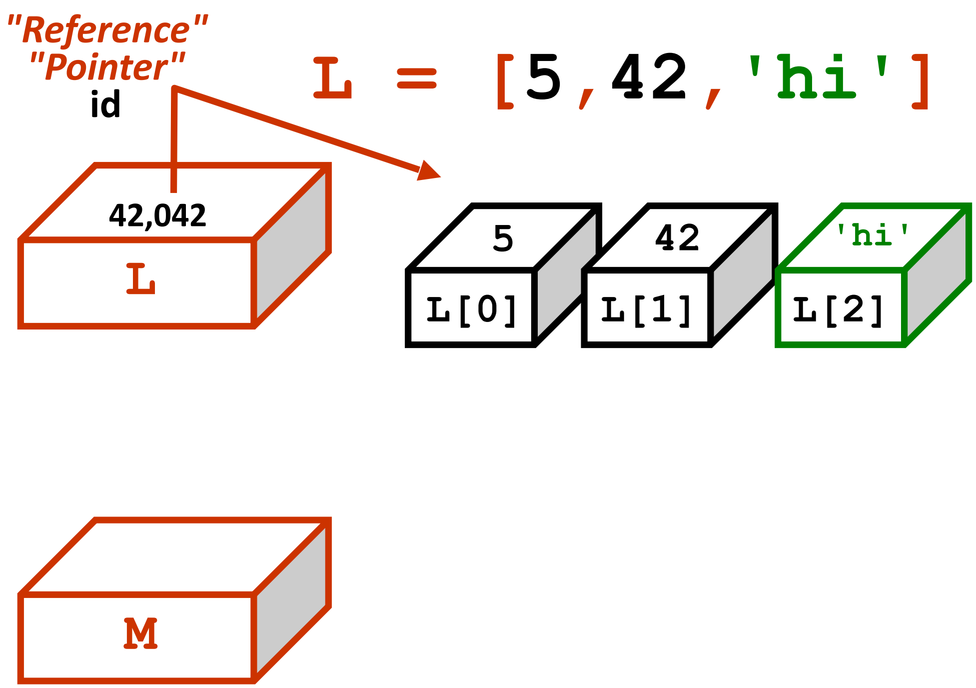
L = [5,42,'hi']; id(L) ...

s = 'hi'; id(s) ...

x = 7; id(x) ...

Shallow vs. Deep

Python's two methods for copying data



L = [5,42,'hi']

M = L

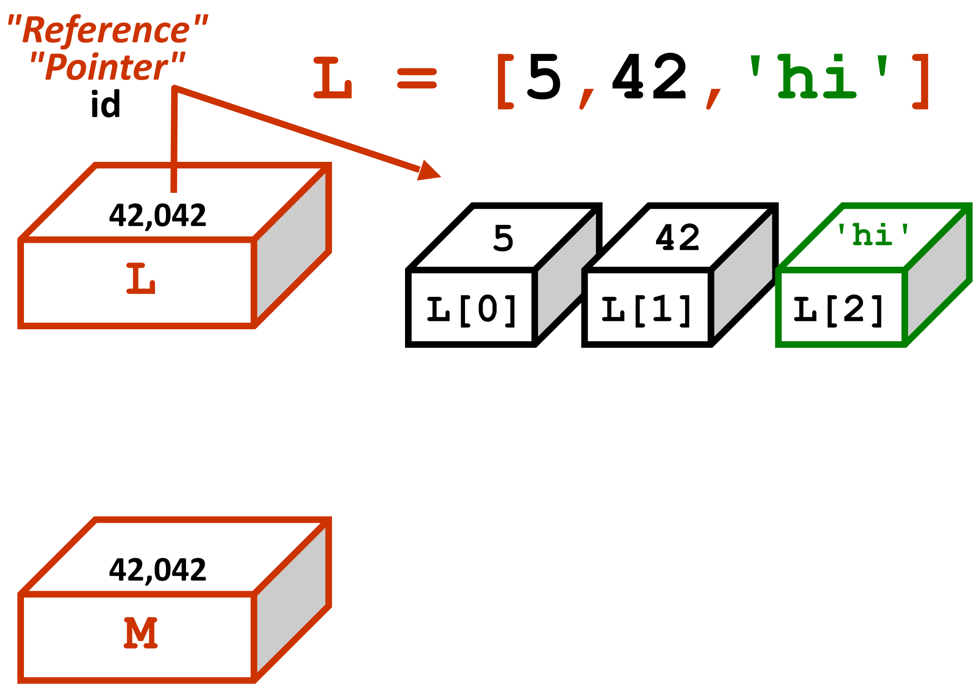
M[0] = 60

What's L[0] ?!

= assignment is "shallow"

Shallow vs. **Deep**

Python's two methods for copying data



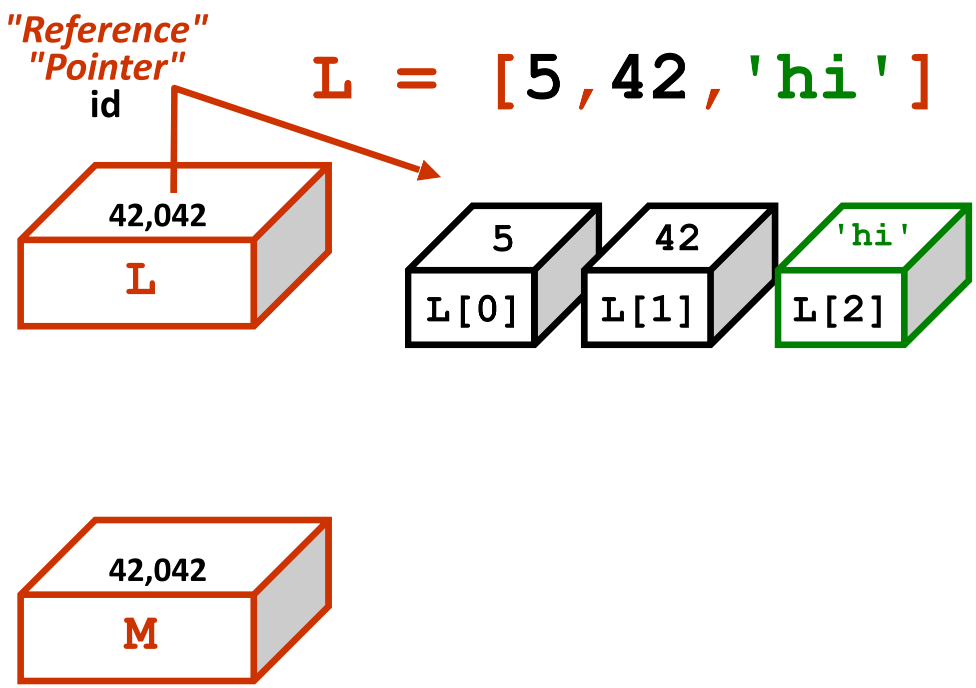

```
from copy import *

L = [5,42,'hi']

M = deepcopy(L)

M[0] = 60
```

What's L[0] ?!

deepcopy is deep!

Shallow vs. **Deep**

Python's two methods for copying data



from copy import *

L = [5,42,'hi']

M = L[:]

M[0] = 60

What's L[0] ?!

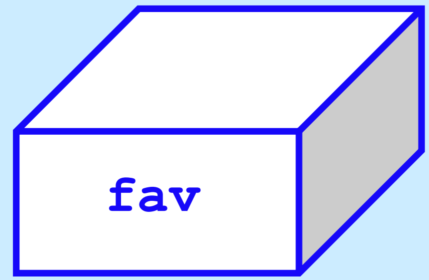
but only one-level
^
slicing is also deep!

import antigravity!

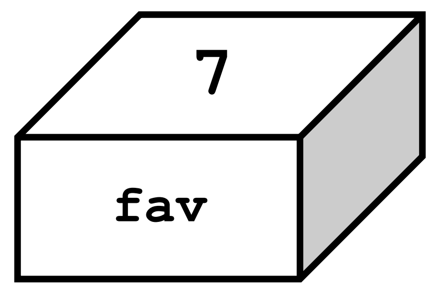
shallow

Python functions: *pass by ^ copy*

```
def conform(fav)

    fav = 42
    return fav
```

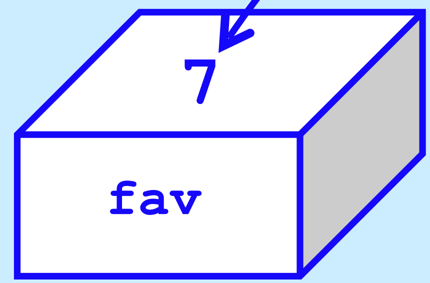



```
def main()

    print(" Welcome! ")

    fav = 7
    fav = conform(fav)

    print(" My favorite # is", fav)
```



Python functions: *pass by* ^shallow^ *copy*

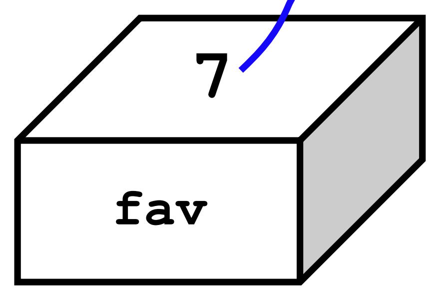
```
def conform(fav)

    fav = 42
    return fav
```



copy of **fav**

"pass by copy" means the contents of **fav** are <u>copied</u> to **fav**

```
def main()

    print(" Welcome! ")

    fav = 7
    fav = conform(fav)

    print(" My favorite # is", fav)
```



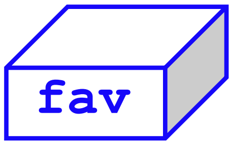
But what if the underlined part were absent... ?

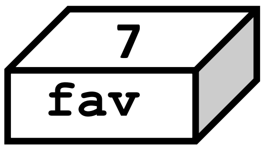
Try it! Rules rule!?

Trace each f'n. What do **`main1`**, **`main2`**, and **`main3`** print?

Numbers: by value. Lists: by reference. Function calls *copy*. Thought experiments: *Don't hand this in...*

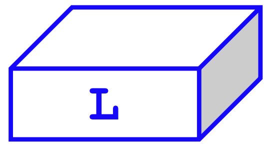
```
def conform1(fav)
  fav = 42
  return fav
```

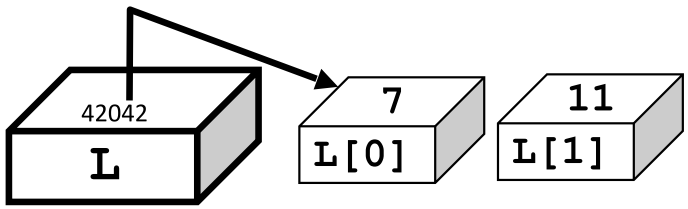





```
def main1()
  fav = 7
  conform1(fav)
  print(fav)
```

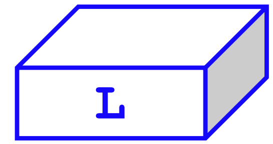
```
def conform2(L)
  L = [42,42]
  return L
```

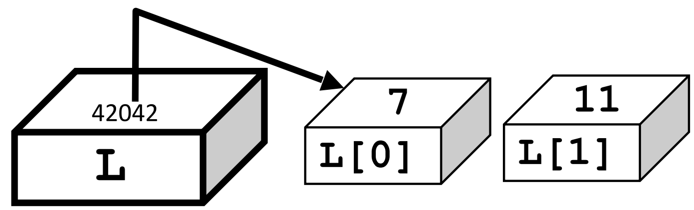





```
def main2()
  L = [7,11]
  conform2(L)
  print(L)
```

```
def conform3(L)
  L[0] = 42
  L[1] = 42
```






```
def main3()
  L = [7,11]
  conform3(L)
  print(L)
```

Notice that there are NO assignment statements after these function calls! The return values *aren't being used*...

Lists are *Mutable*

You can change **the contents** of lists from within functions that take lists as input.

- Lists are **MUTABLE** objects

Those changes will be visible **everywhere**.

Numbers, strings, etc. are IMMUTABLE – they can't be changed, only reassigned.

2D data!

All and only the rules that govern 1D data apply here – no new rules to learn!

~ pure composition

Lists ~ 1D data

```
A = [ 42, 75, 70 ]
```

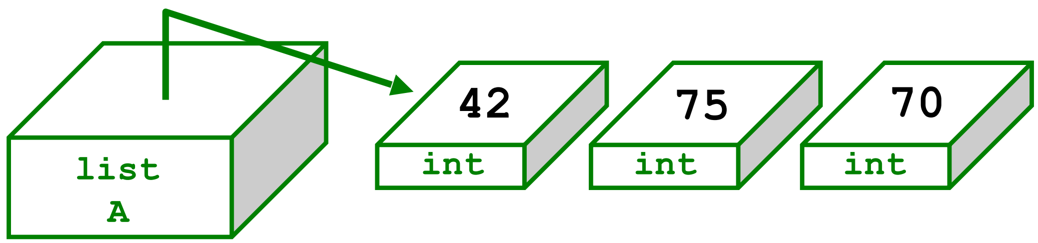
What does `A` "look like" ?

```
len(A)  ?
id(A)   ?
id(A[0]) ?
```

1D lists are familiar – but lists can hold ANY kind of data – ***including lists!***

Lists ~ 1D data

```
A = [ 42, 75, 70 ]
```




```
len(A) ?
id(A)  ?
id(A[0]) ?
```

1D lists are familiar – but lists can hold ANY kind of data – ***including lists!***

Lists ~ 2D data

```
A = [ [1,2,3,4], [5,6], [7,8,9,10,11] ]
```

What does *this* **`A`** "look like" ?

Where's 3?

`len(A)`

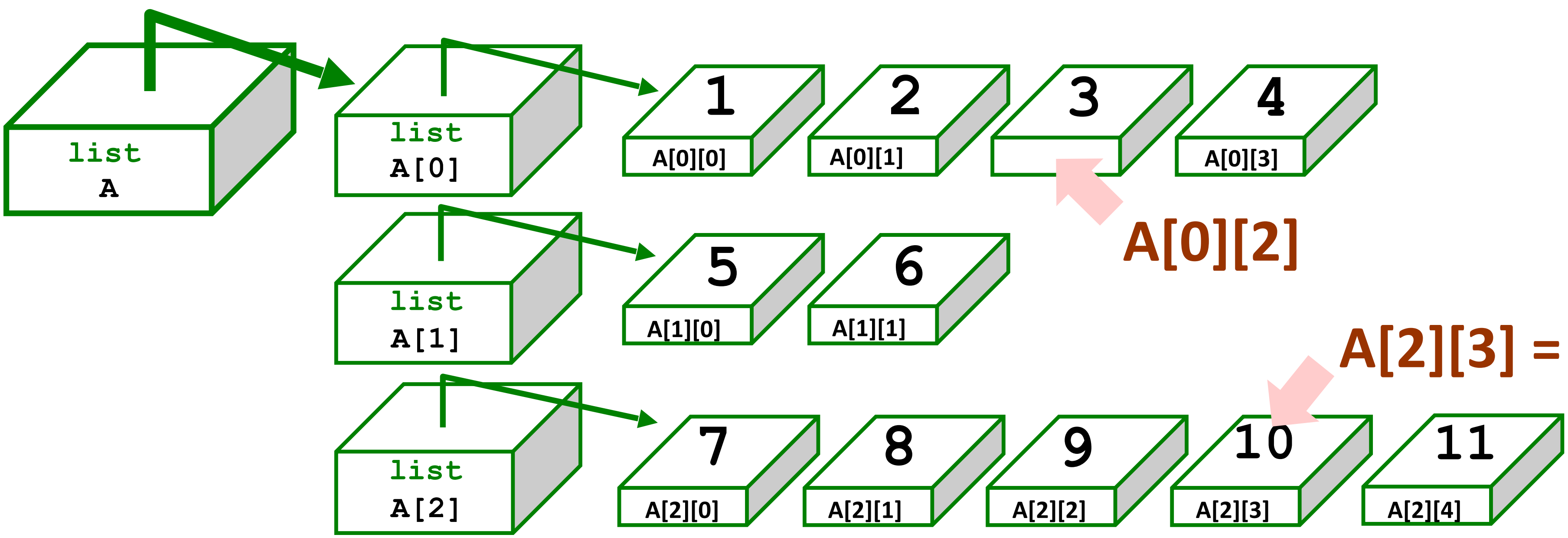
`len(A[0])`

`len(A[1])`

Replace 10 with 42.

Lists ~ 2D data

```
A = [ [1,2,3,4], [5,6], [7,8,9,10,11] ]
```



Where's 3?

`len(A)`

3

4

`len(A[0])`

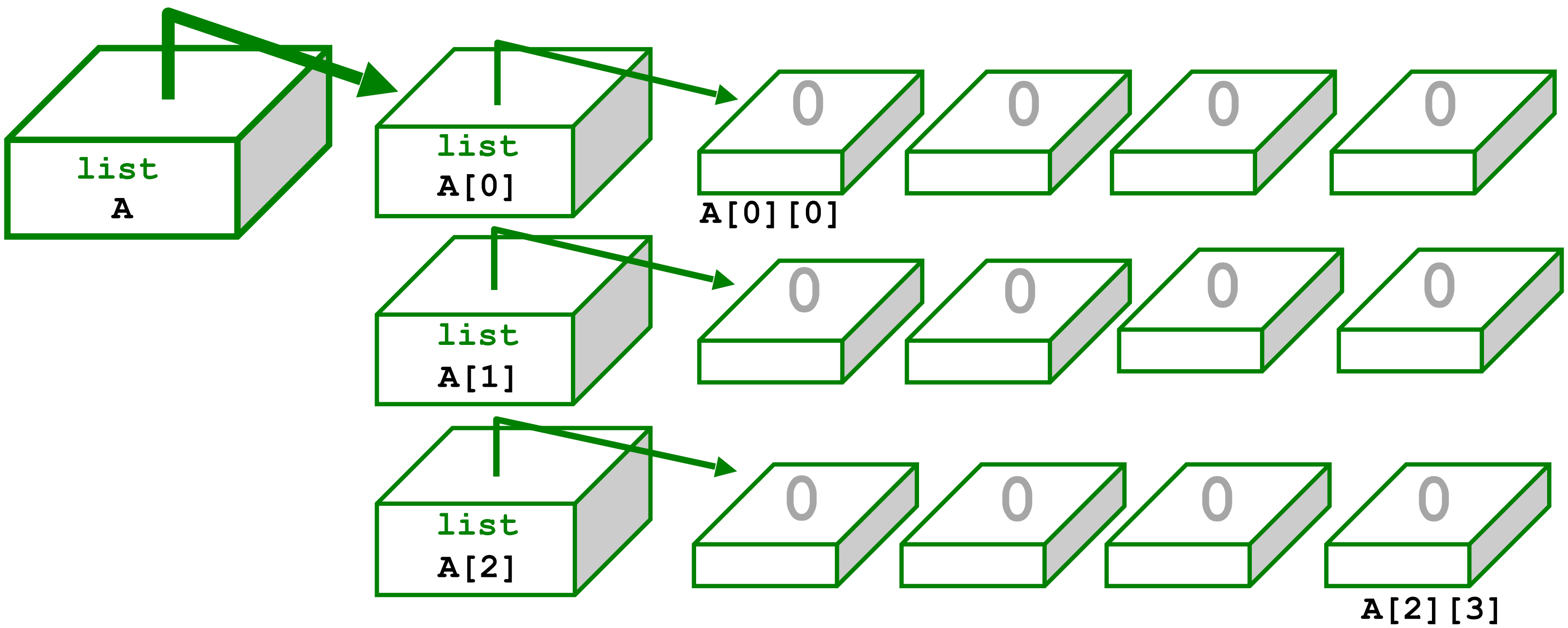
`len(A[1])`

2

Replace 10 with 42.

Rectangular 2D data

```
A = [ [0,0,0,0], [0,0,0,0], [0,0,0,0] ]
```



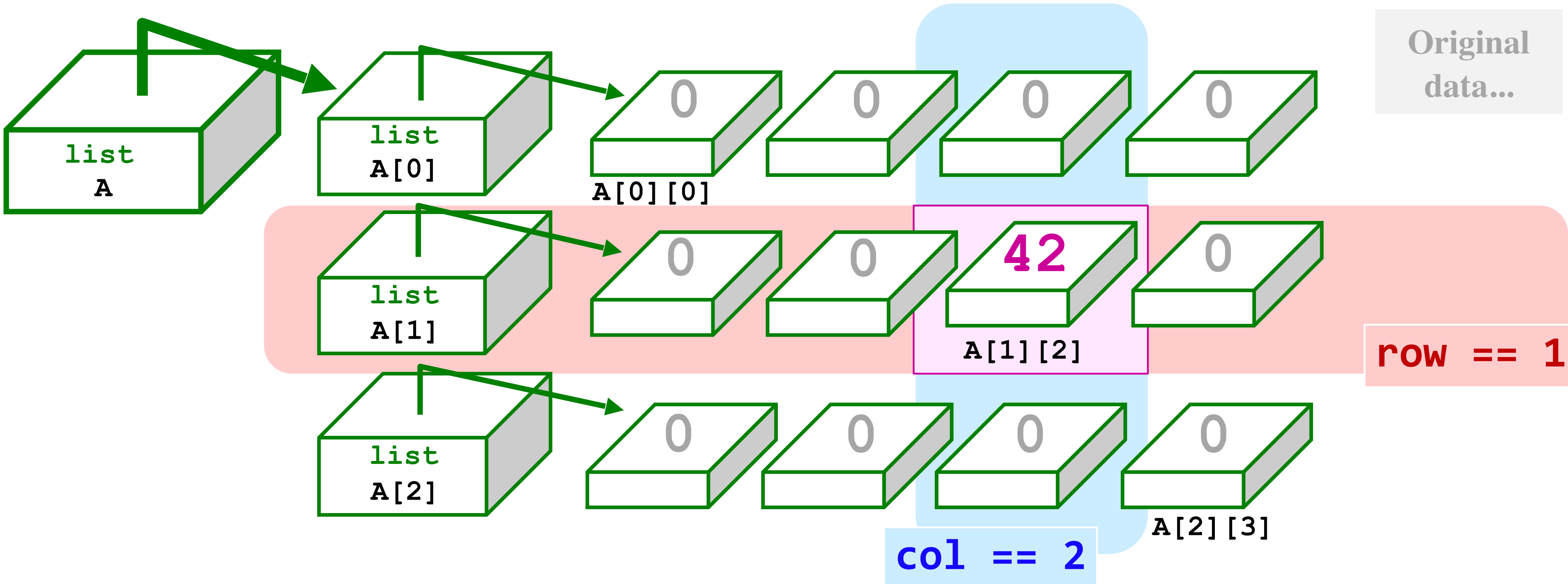
Original data...

```
A[1][2] = 42
```

```
A[r][c] = value
```

Rectangular 2D data

```
A = [ [0,0,0,0], [0,0,0,0], [0,0,0,0] ]
```




```
A[1][2] = 42
```

row == 1

col == 2

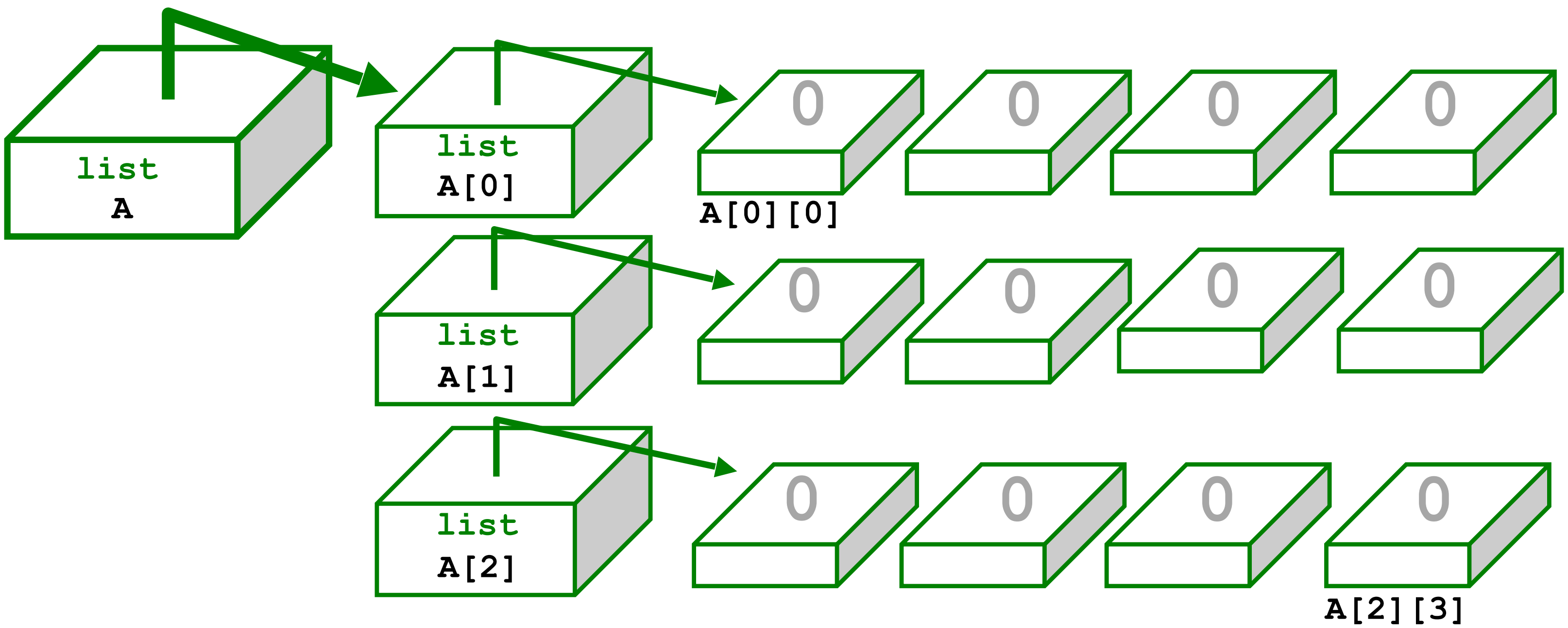
row r

col c

```
A[r][c] = value
```

Rectangular 2D data

```
A = [ [0,0,0,0], [0,0,0,0], [0,0,0,0] ]
```



Original data...

```
NROWS = len(A)    # HEIGHT
NCOLS = len(A[0])  # WIDTH

for r in range( 0,NROWS ):
    for c in range( 0,NCOLS ):
        if r == c:    A[r][c] = 4
        else:         A[r][c] = 2
```

Nested Loops
~ 2d Data

2 in-a-row ?

```
A = [ [4, 2, 2, 2],
      [2, 2, 4, 4],
      [2, 4, 4, 2] ]
```

```
def two_in_a_row(A):
    """ what's happening ? """
    NROWS = len(A)
    NCOLS = len(A[0])
    B = deepcopy( A )

    for r in range( 0,NROWS ):
       for c in range( 0,NCOLS ):
          if c == NCOLS-1:
                B[r][c] = False
          elif A[r][c] == A[r][c+1]:
                B[r][c] = True
          else:
                B[r][c] = False
```

A

	col 0	col 1	col 2	col 3
row 0	4	2	2	2
row 1	2	2	4	4
row 2	2	4	4	2

B

Challenge: How could we *change the code above* to check for two-in-a-row SOUTHWARD -- or DIAGONALLY !?

What does **two_in_a_row(A)** place into B?

hw9pr2...

```
A = [ [4, 2, 2, 2],
      [2, 2, 4, 4],
      [2, 4, 4, 2] ]
```

```
def two_in_a_row(A):
    """ what happens here ? """
    NROWS = len(A)
    NCOLS = len(A[0])
    B = deepcopy( A )

    for r in range( 0,NROWS ):
      for c in range( 0,NCOLS ):
        if c == NCOLS-1:
            B[r][c] = False
        elif A[r][c] == A[r][c+1]:
            B[r][c] = True
        else:
            B[r][c] = False
```

on the far edge!

2 in a row eastward – yes!

not 2 in a row

A

	col 0	col 1	col 2	col 3
row 0	4	2	2	2
row 1	2	2	4	4
row 2	2	4	4	2

B

F	T	T	F
T	F	T	F
F	T	F	F

What **two_in_a_row(A)** places into B...

Challenge: How could we *change the code above* to check for two-in-a-row SOUTHWARD or DIAGONALLY !?!

hw9pr2...

Use as your hw9pr2 starting point... !

```
def two_in_a_row(A
    """ what happe
    NROWS = len(A)
    NCOLS = len(A[
    B = deepcopy(    )

    for r in range( 0,NROWS ):
       for c in range( 0,NCOLS ):
          if c == NCOLS-1:
              B[r][c] = False
          elif A[r][c] == A[r][c+1]:
              B[r][c] = True
          else:
              B[r][c] = False
```

on the far edge!

2 in a row eastward – yes!

not 2 in a row

	col 0	col 1	col 2	col 3
		2	2	2
row 1	2	2	4	4
row 2	2	4	4	2

B

F	T	T	F
T	F	T	F
F	T	F	F

What **two_in_a_row(A)** places into B...

Challenge: How could we *change the code above* to check for two-in-a-row SOUTHWARD or DIAGONALLY !?!

First, try it by eye... *... then, on hw9pr2, w/Python!*

	col 0	col 1	col 2	col 3	col 4	
row 0	' '	'X'	'O'	' '	'O'	
row 1	'X'	'X'	'X'	'O'	'O'	
row 2	' '	'X'	'O'	'X'	'O'	
row 3	'X'	'O'	'O'	' '	'X'	

```
A = [ [' ','X','O',' ','O'],
      ['X','X','X','O','O'],
      [' ','X','O','X','O'],
      ['X','O','O',' ','X'] ]
```

the data doesn't wrap around

checker, start row, start col, LoL

```
inarow_3east('X', 1, 0, A)  →
```

First, try it by eye... *... then, on hw9pr2, w/Python!*

	col 0	col 1	col 2	col 3	col 4

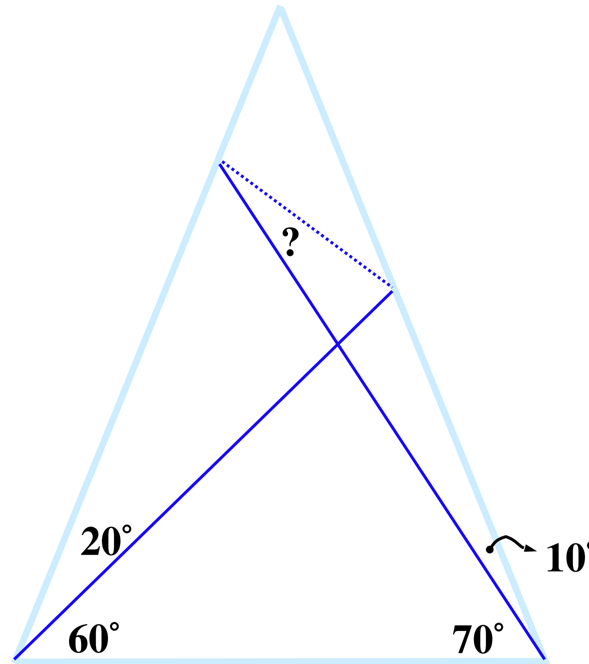
```
A = [ row 0 [' ','X','O',' ','O'],
      row 1 ['X','X','X','O','O'],
      row 2 [' ','X','O','X','O'],
      row 3 ['X','O','O',' ','X'] ]
```

the data doesn't wrap around

```
              checker  start  start  LoL
                       row    col
inarow_3east('X', 1, 0, A)          →   True

inarow_3south('O', 0, 4, A)         →

inarow_3southeast('X', 2, 3, A)     →

inarow_3northeast('X', 3, 1, A)     →
```

hw9pr1 (lab): *Conway's Game of Life*

Geometer @ Princeton



1995

(no trig)

simple rules ~ surprising behavior

The fantastic combinations of John Conway's new solitaire game "life"

not really solitaire...

by Martin Gardner

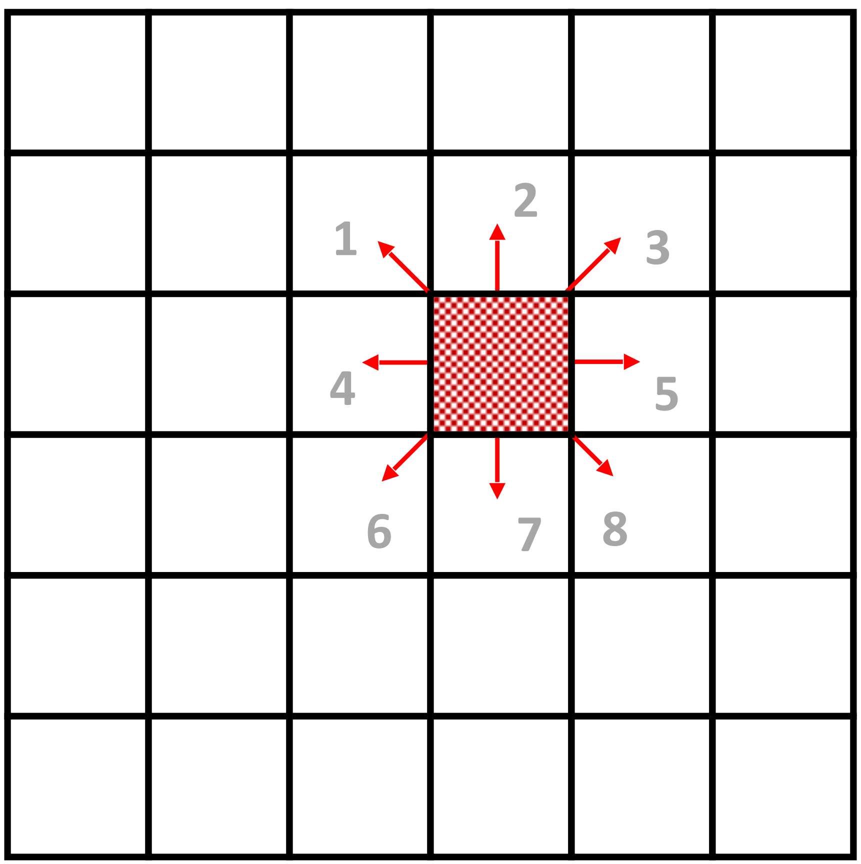
Scientific American 223 (October 1970): 120-123.

1970

Lab Problem: ***Conway's Game of Life***

Grid World

red cells are "alive"



white cells are empty

Evolutionary rules

- Everything depends on a cell's eight neighbors

- Exactly 3 neighbors give birth to a new, live cell.

- Exactly 2 or 3 neighbors keep an existing cell alive.

- Any other # of neig
and the cen

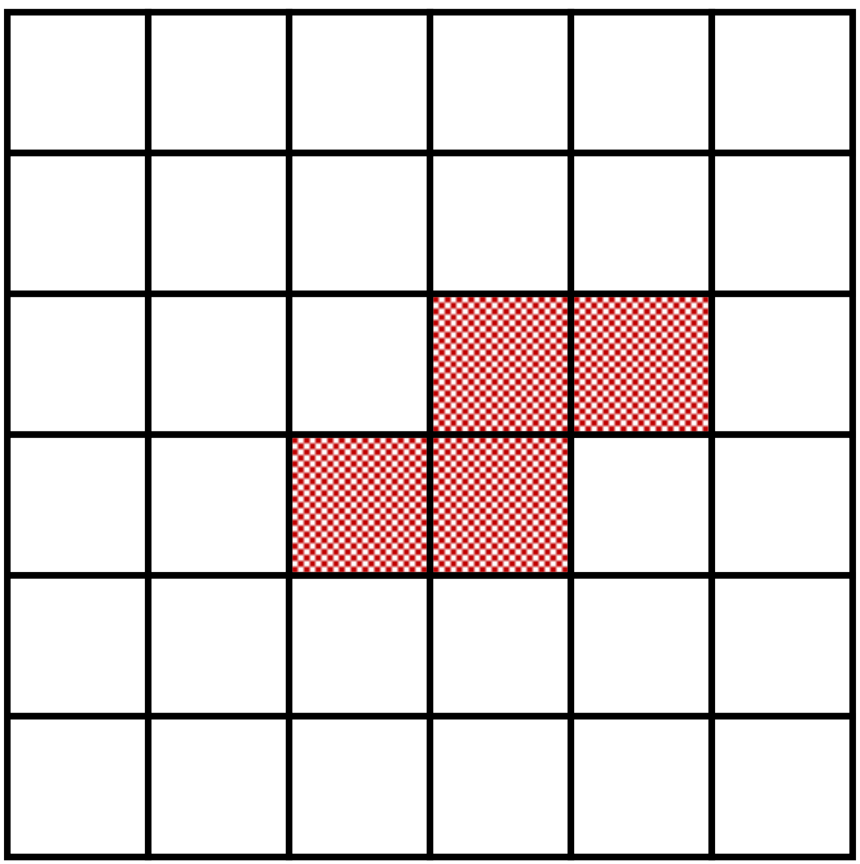
Only 2 rules

Lab Pr… ay's *Game of Life*

"Parent generation"

Grid World

red cells are "alive"



white cells are empty

Evolutionary rules

• Everything depends on a cell's eight neighbors

• Exactly 3 neighbors give birth to a new, live cell.

rule 1

• Exactly 2 or 3 neighbors keep an existing cell alive.

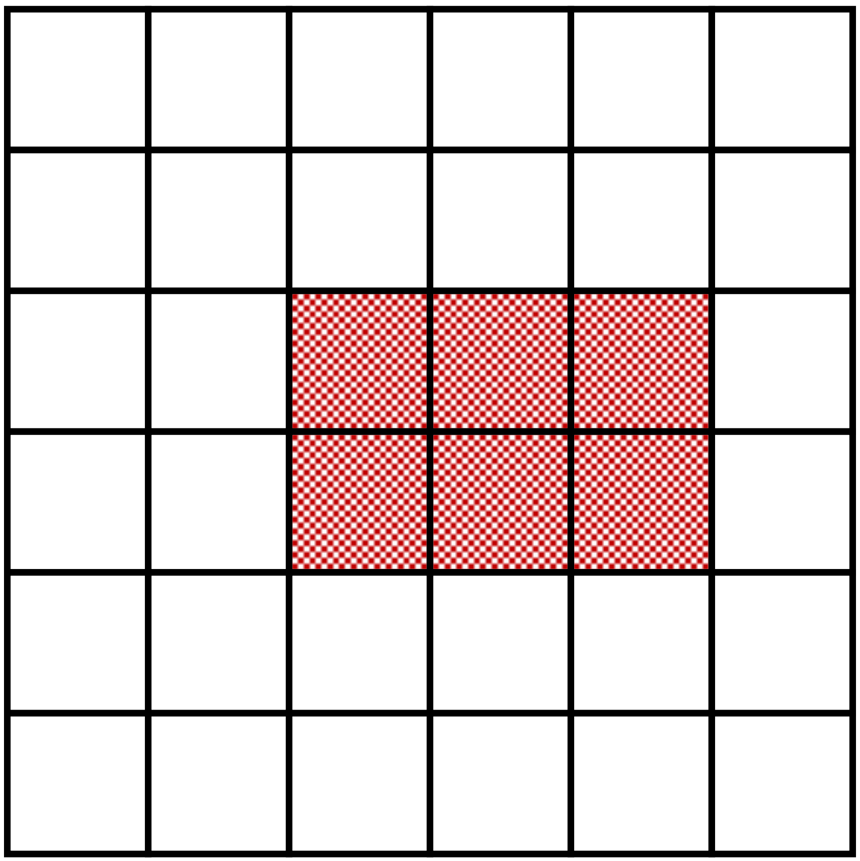
• Any other # of neighbors and the central cell dies…

Lab Probl… *ay's Game of Life*

"Child generation"

Grid World

red cells are "alive"



white cells are empty

Evolutionary rules

- Everything depends on a cell's eight neighbors

- Exactly 3 neighbors give birth to a new, live cell.

- Exactly 2 or 3 neighbors keep an existing cell alive.

- Any other # of neighbors and the central cell dies…

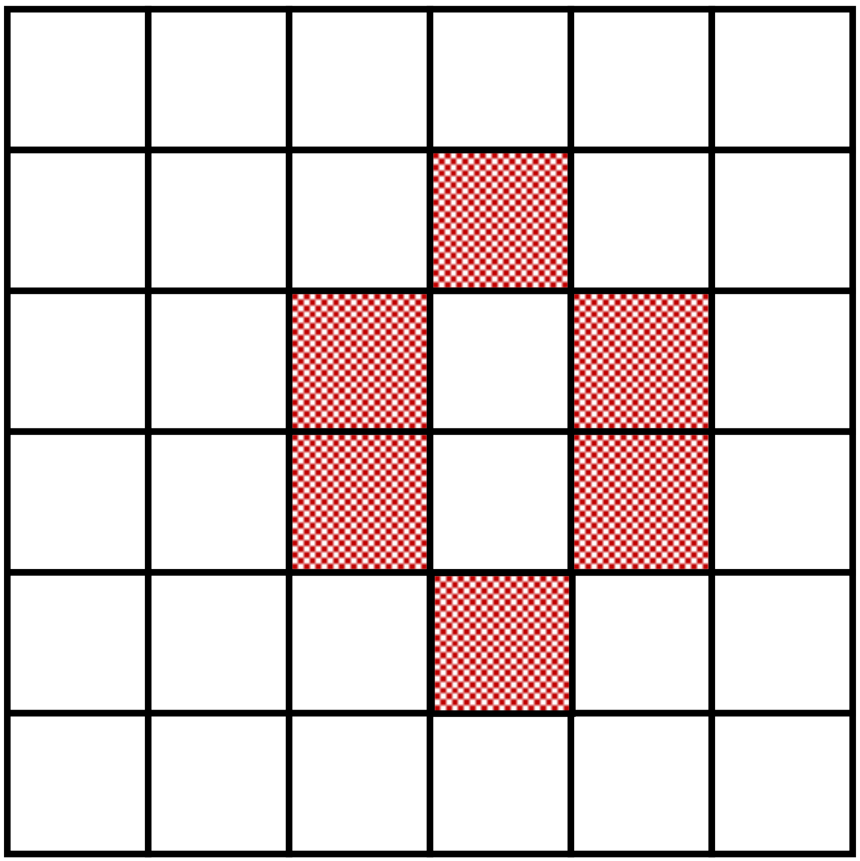
rule 2

Lab Pr… *ay's Game of Life*

"Grandchild generation"

Grid World

red cells are alive



white cells are empty

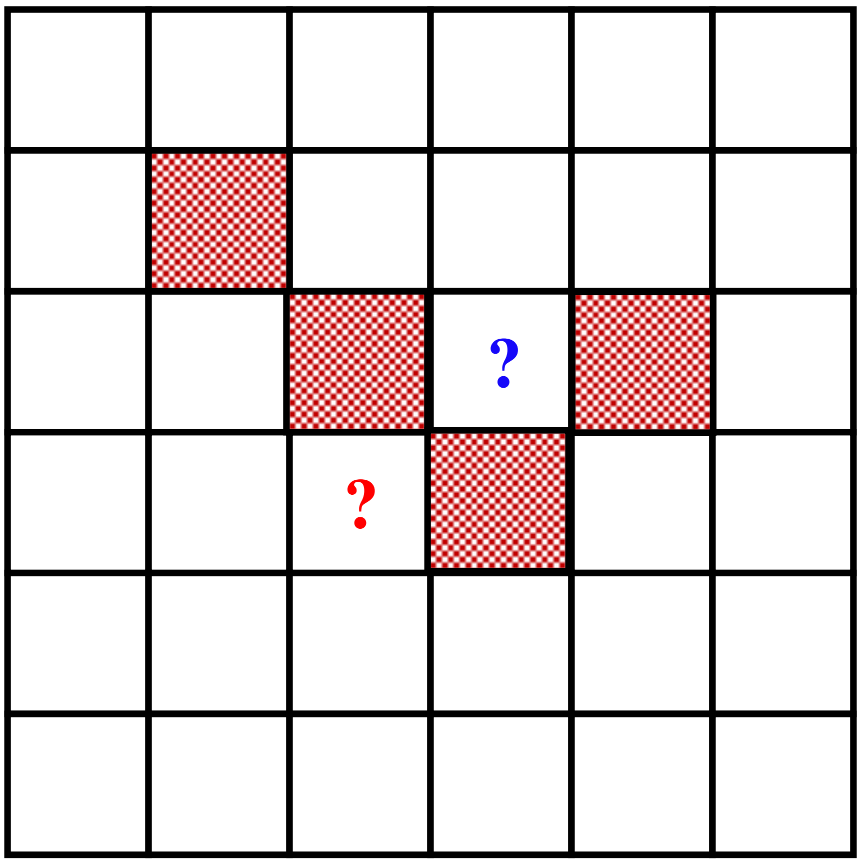
Evolutionary rules

What's next?

- Everything de… cell's eig… …rs
- Exactly 3 neighbors give birth to a new, live cell.
- Exactly 2 or 3 neighbors keep an existing cell alive.
- Any other # of neighbors and the central cell dies…

Lab Problem: *Creating life*

`next_life_generation( A )`



For each cell…

- 3 live neighbors – **life!**
- 2 live neighbors – **same**
- 0, 1, 4, 5, 6, 7, or 8 live neighbors – **death**
- computed all at once, not cell-by-cell, so the ? at left does NOT come to life, but ? does!

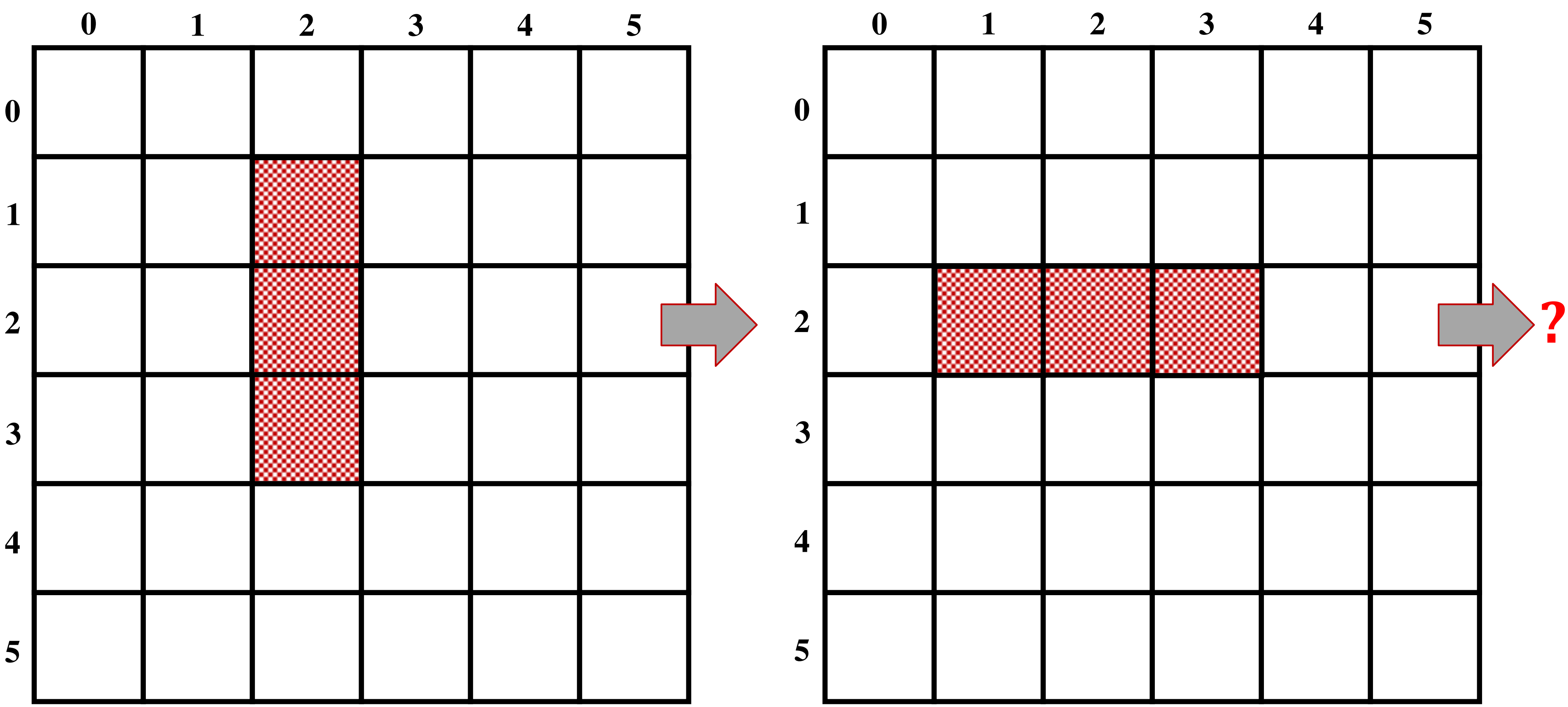
http://www.math.com/students/wonders/life/life.html

Lab Problem: ***Creating life***

`next_life_generation( A )`

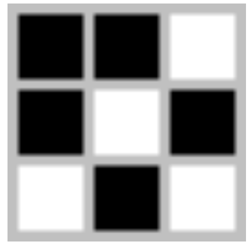
old generation is the input, A

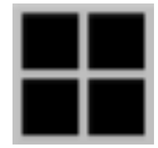
returns the next generation



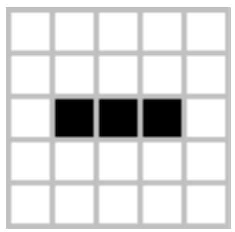
Lab Problem: ***Creating life***

Stable configurations:
"rocks"

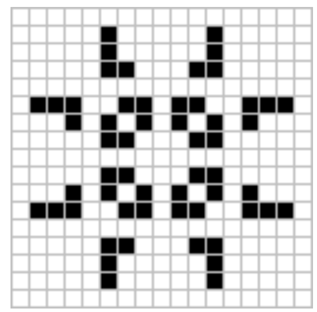




Periodic
"plants"



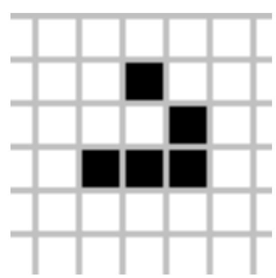
period 2



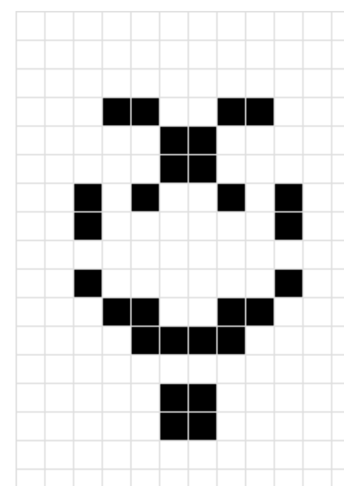
period 3

Self-propagating
"animals"

glider



Copperhead: 2016

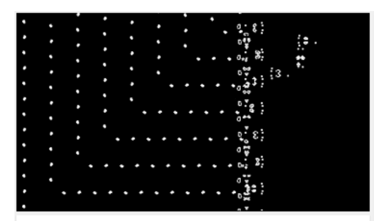


Life @ HMC?

Copperhead: 2016

Life @ HMC!

Life, universally!

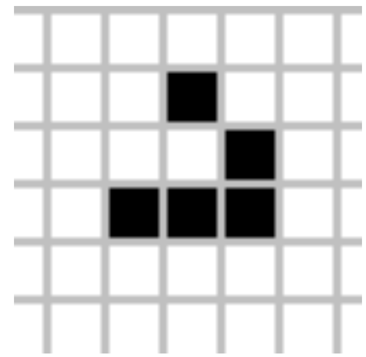


www.youtube.com/watch?v=xP5-iIeKXE8

Lab Problem: *Creating life*

Many life configurations expand forever…

"Gosper glider gun"



"glider"

What is the largest amount of the life universe that can be filled with cells?

How sophisticated can Life-structures get?

www.ibiblio.org/lifepatterns/