[-35, -24, -13, -2, 9, 20, 31, ?]

[26250, 5250, 1050, 210, ?]

[90123241791111 , 93551622, 121074, 3111, ?]

[1, 11, 21, 1211, 111221, ?]

What's next?

I'm glad you asked!

What's the meaning of Life?

Simple rules can create complex results...

$$z = z^2 + c$$

The rule: ***Don't follow this rule.***

Hw 9: due Mon., 04/02 *hw9 is mostly lab ~ join for lab!*

How about that costume?!

The ***read it and weep*** sequence

```
       1
      11
      21
    1211
  111221
  312211
13112221

     ...
```

`str` vs. **`int`**

When does the first **4** appear?

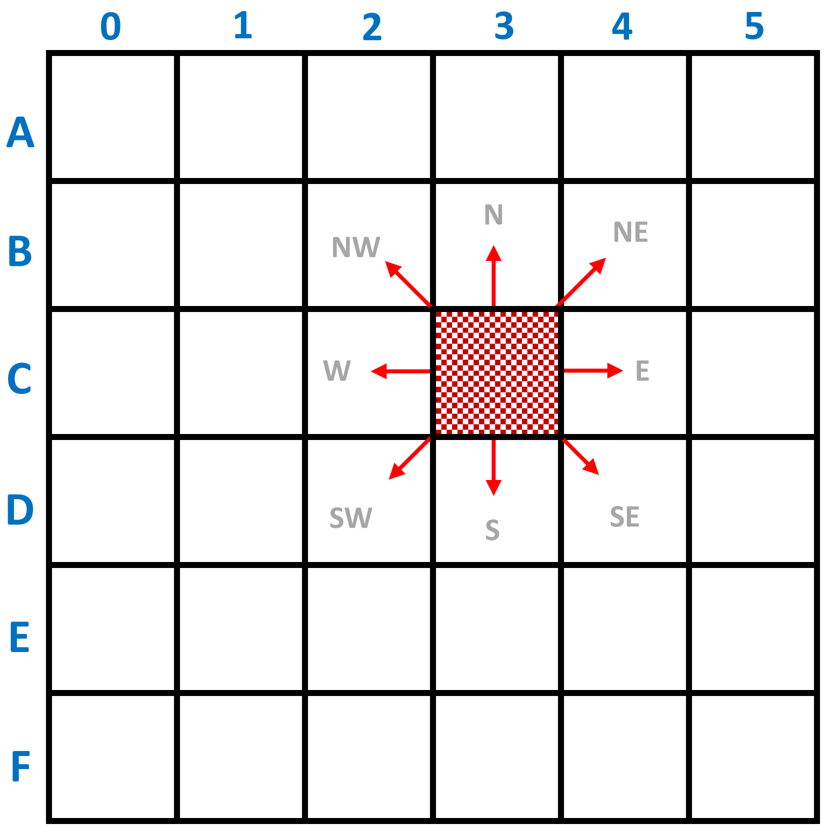
How fast do these terms grow?

Extra extra credit this wk9

hw9pr1 lab: *Conway's Game of Life*

Grid World

red cells are "alive"



white cells are empty

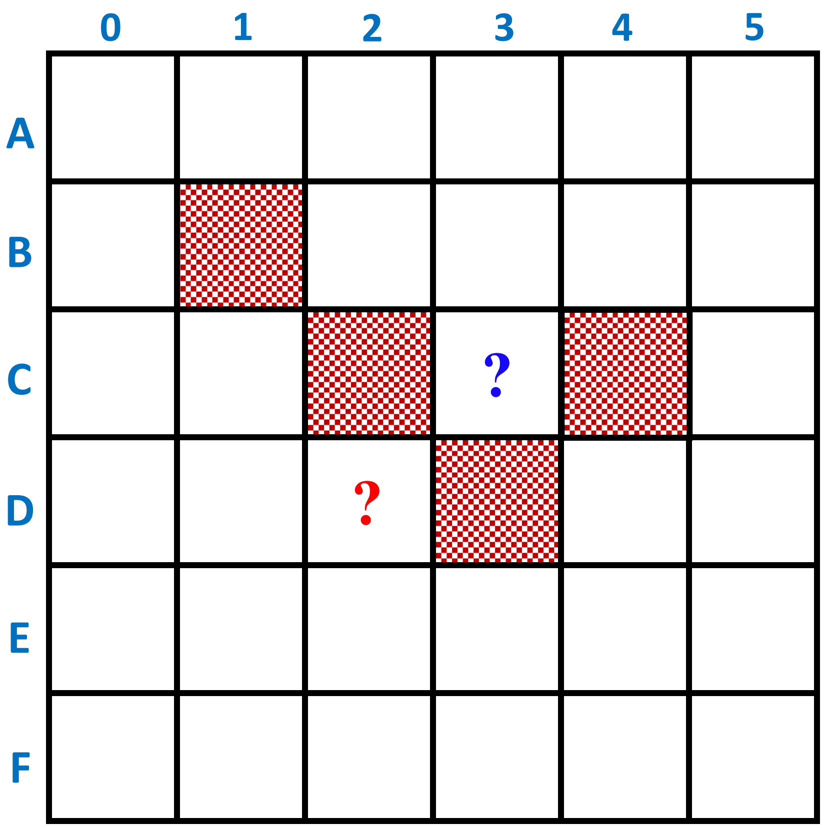
Evolutionary rules

- 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 neighbors and there's no life...

Only 2 rules of life...

hw9pr1 lab: *Creating Life*

`next_life_generation( A )`



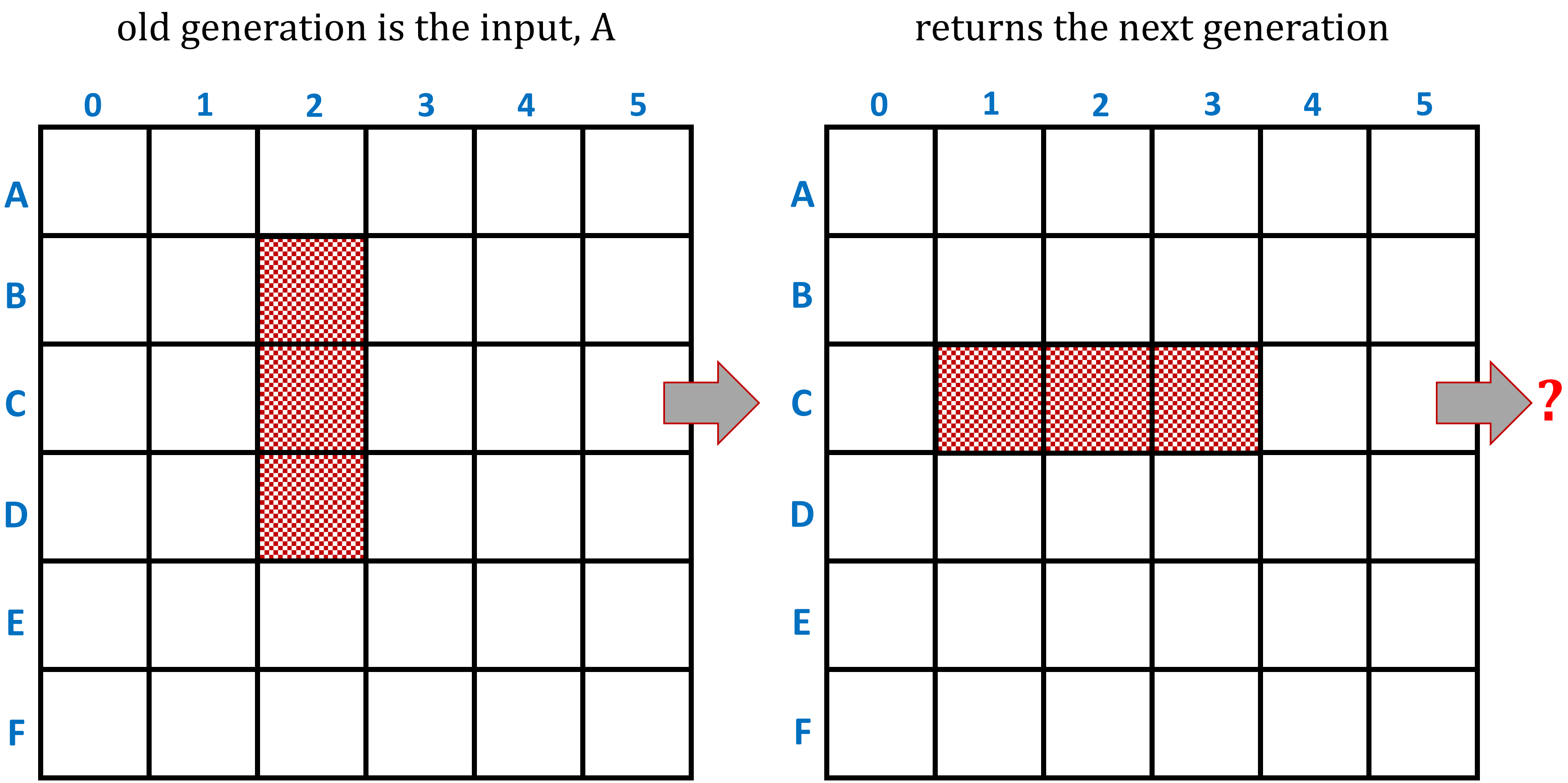
For each cell…

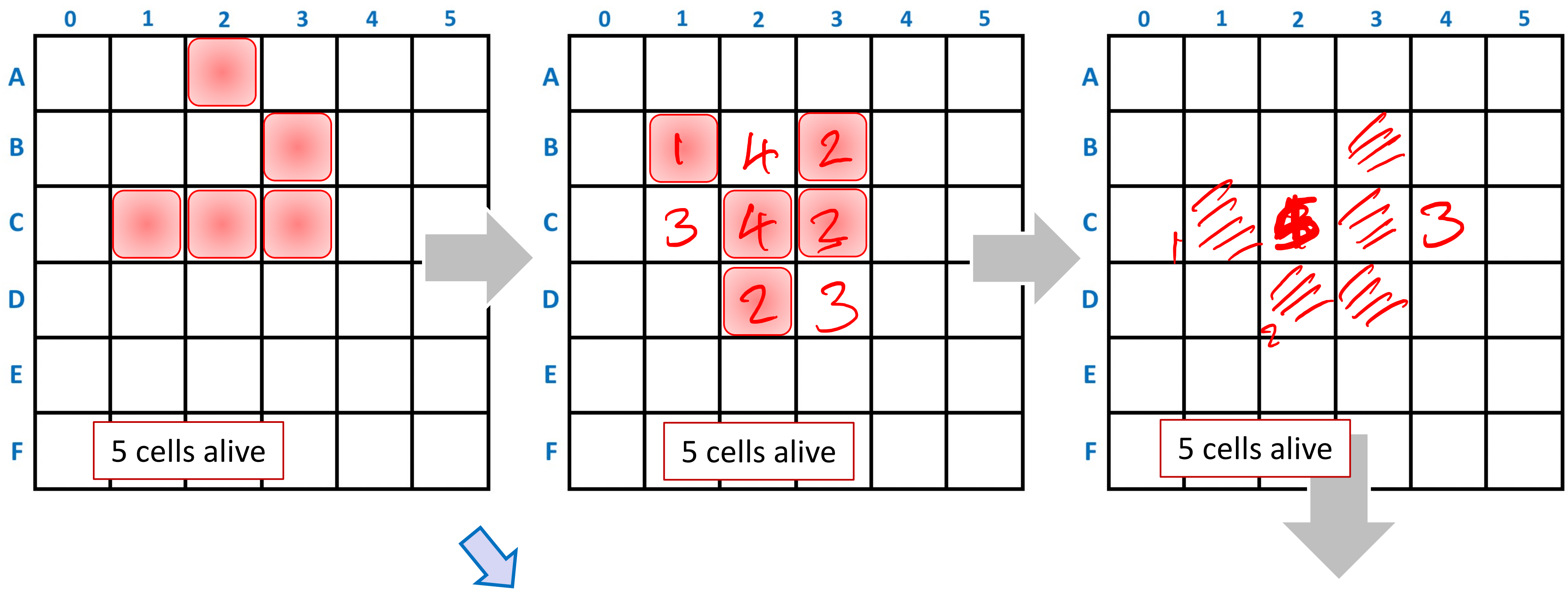
- 3 live neighbors – **life!**
- 2 live neighbors – **same**
- 0, 1, 4, 5, 6, 7, or 8 live neighbors – **death**
- computed all at once, ***not*** cell-by-cell,
- so the ? at left DOES come to life, but the ? does *not*...

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

hw9pr1 lab: ***Creating Life***

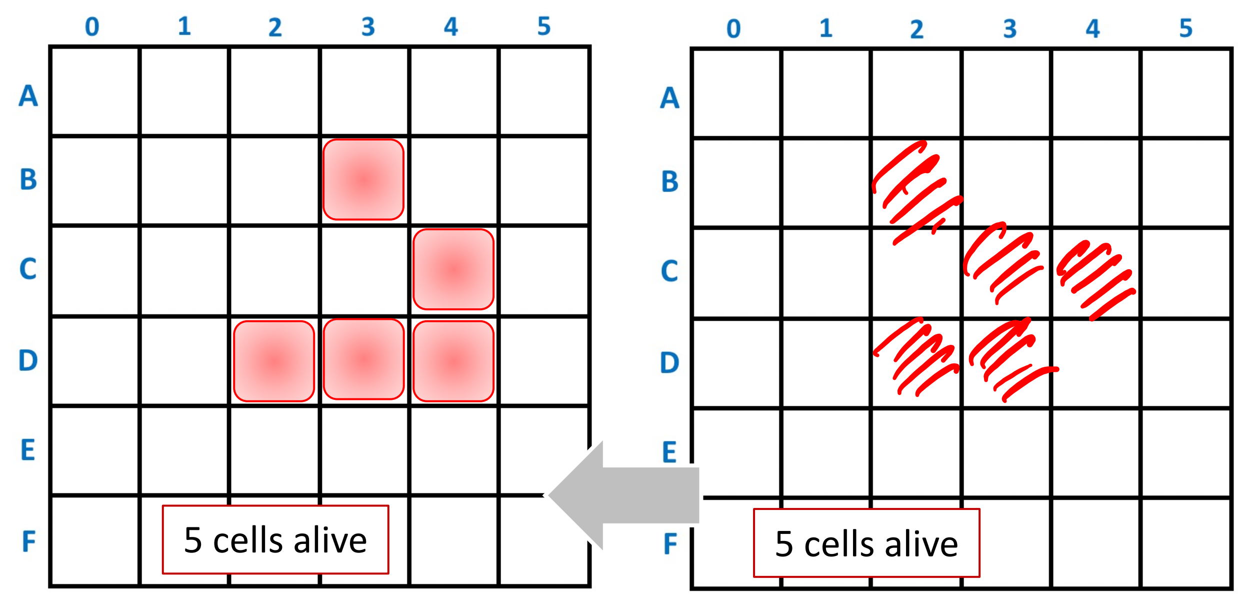
`next_life_generation( A )`

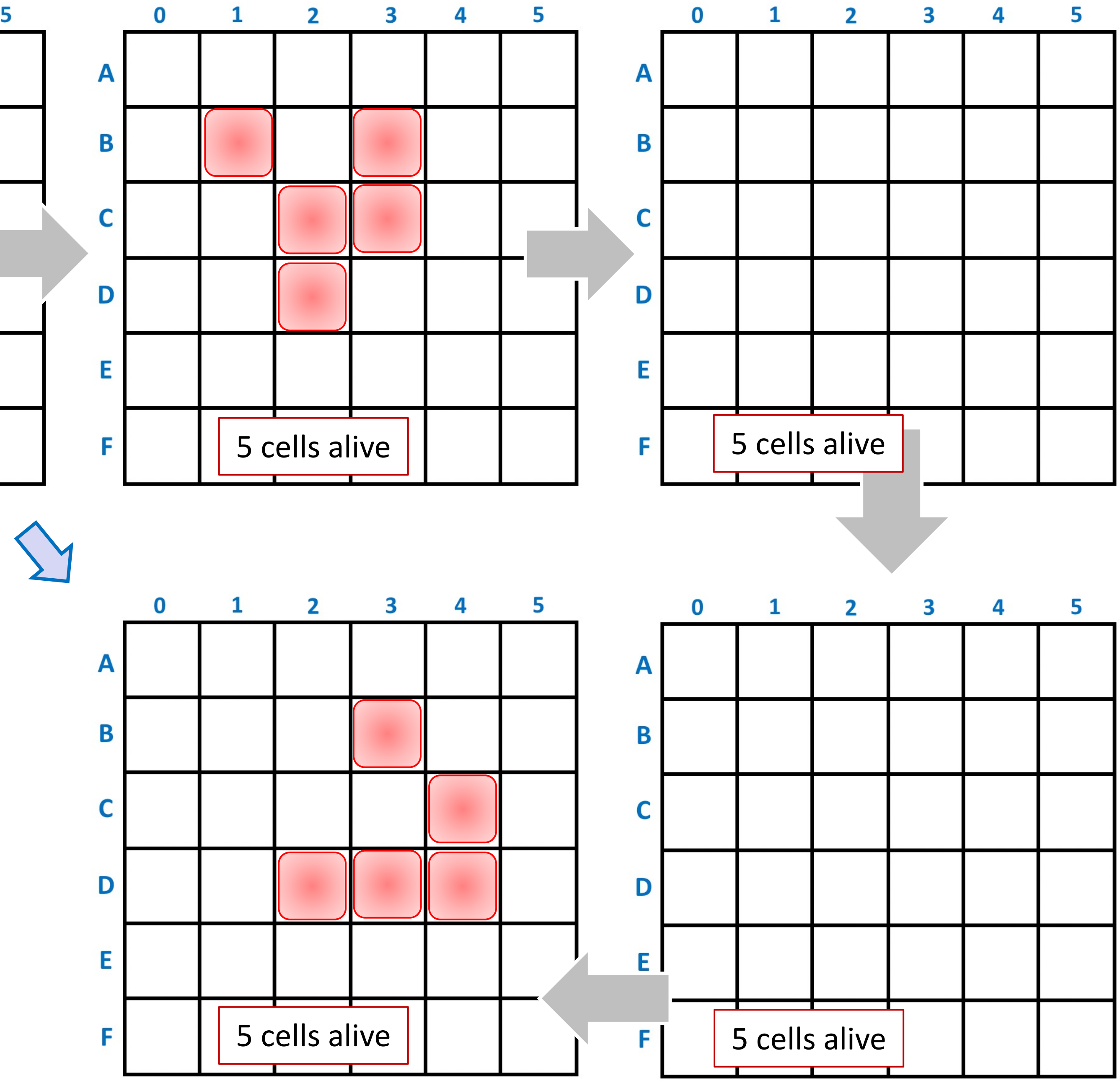




Follow "the glider"

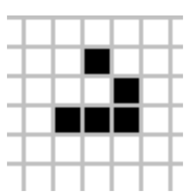
Life's *simplest* self-propagating form...





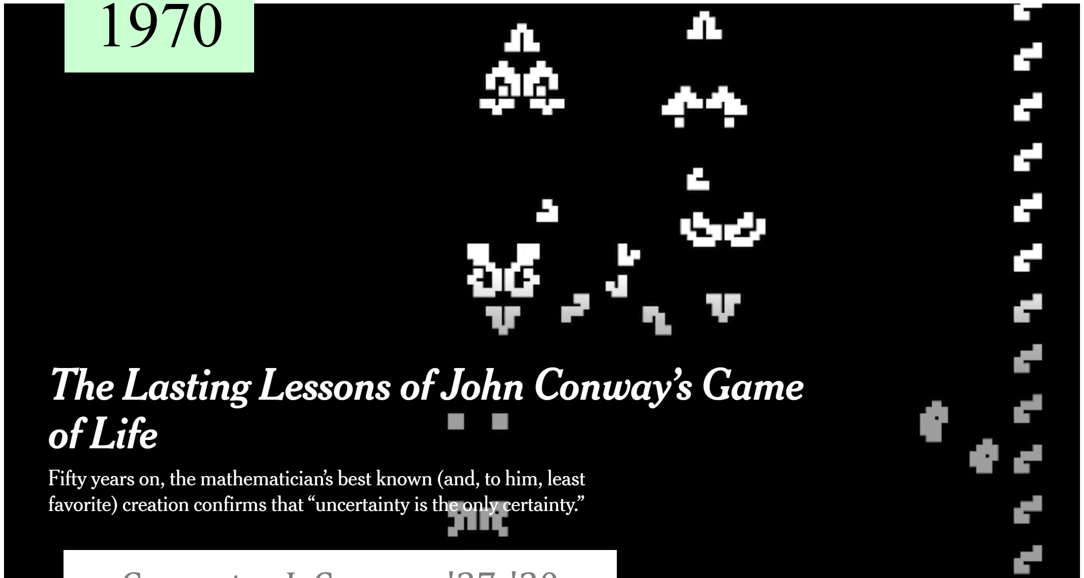
Follow "the glider"

Life's *simplest* self-propagating form...

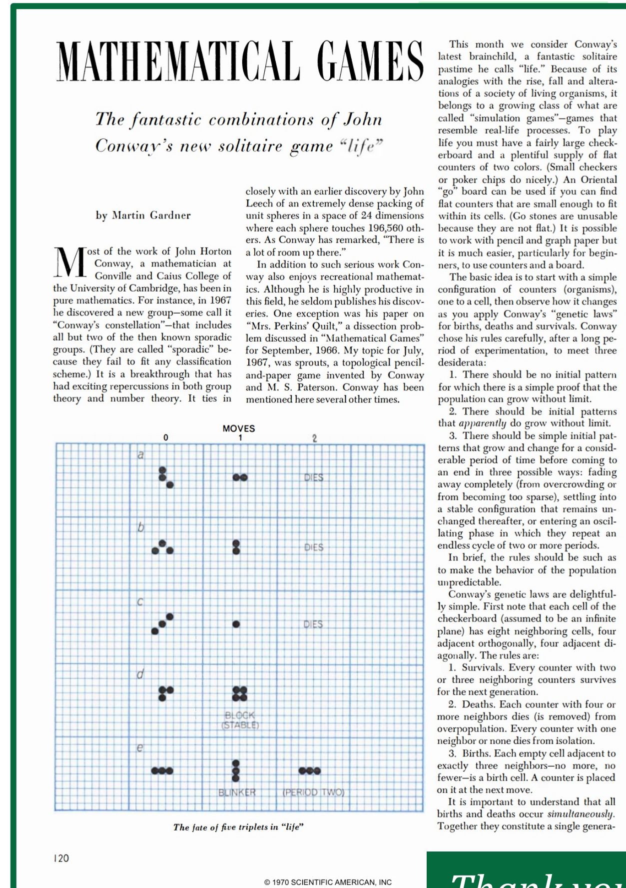


hw9pr1: *Conway's Game of Life*

1970



Geometer J. Conway '37-'20



MATHEMATICAL GAMES

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

by Martin Gardner

Most of the work of John Horton Conway, a mathematician at Gonville and Caius College of the University of Cambridge, has been in pure mathematics. For instance, in 1967 he discovered a new group—some call it "Conway's constellation"—that includes all but two of the then known sporadic groups. (They are called "sporadic" because they fail to fit any classification scheme.) It is a breakthrough that has had exciting repercussions in both group theory and number theory. It ties in closely with an earlier discovery by John Leech of an extremely dense packing of unit spheres in a space of 24 dimensions where each sphere touches 196,560 others. As Conway has remarked, "There is a lot of room up there."

In addition to such serious work Conway also enjoys recreational mathematics. Although he is highly productive in this field, he seldom publishes his discoveries. One exception was his paper on "Mrs. Perkins' Quilt," a dissection problem discussed in "Mathematical Games" for September, 1966. My topic for July, 1967, was sprouts, a topological pencil-and-paper game invented by Conway and M. S. Paterson. Conway has been mentioned here several other times.

This month we consider Conway's latest brainchild, a fantastic solitaire pastime he calls "life." Because of its analogies with the rise, fall and alterations of a society of living organisms, it belongs to a growing class of what are called "simulation games"—games that resemble real-life processes. To play life you must have a fairly large checkerboard and a plentiful supply of flat counters of two colors. (Small checkers or poker chips do nicely.) An Oriental "go" board can be used if you can find flat counters that are small enough to fit within its cells. (Go stones are unusable because they are not flat.) It is possible to work with pencil and graph paper but it is much easier, particularly for beginners, to use counters and a board.

The basic idea is to start with a simple configuration of counters (organisms), one to a cell, then observe how it changes as you apply Conway's "genetic laws" for births, deaths and survivals. Conway chose his rules carefully, after a long period of experimentation, to meet three desiderata:

1. There should be no initial pattern for which there is a simple proof that the population can grow without limit.

2. There should be initial patterns that *apparently* do grow without limit.

3. There should be simple initial patterns that grow and change for a considerable period of time before coming to an end in three possible ways: fading away completely (from overcrowding or from becoming too sparse), settling into a stable configuration that remains unchanged thereafter, or entering an oscillating phase in which they repeat an endless cycle of two or more periods.

In brief, the rules should be such as to make the behavior of the population unpredictable.

Conway's genetic laws are delightfully simple. First note that each cell of the checkerboard (assumed to be an infinite plane) has eight neighboring cells, four adjacent orthogonally, four adjacent diagonally. The rules are:

1. Survivals. Every counter with two or three neighboring counters survives for the next generation.

2. Deaths. Each counter with four or more neighbors dies (is removed) from overpopulation. Every counter with one neighbor or none dies from isolation.

3. Births. Each empty cell adjacent to exactly three neighbors—no more, no fewer—is a birth cell. A counter is placed on it at the next move.

It is important to understand that all births and deaths occur *simultaneously*. Together they constitute a single genera-

The fate of five triplets in "life"

120

Thank you, Emma!

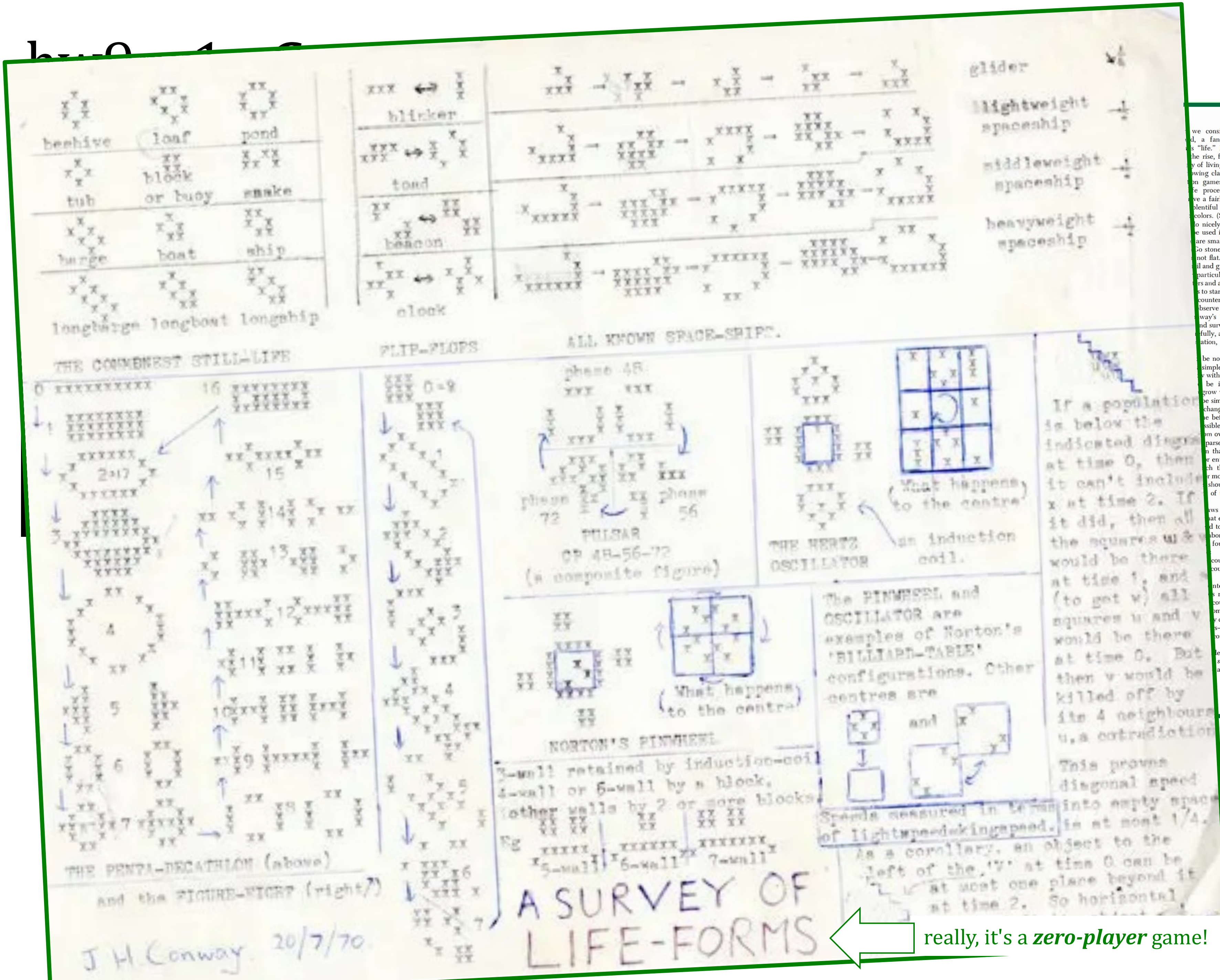
Simple rules ~ Surprising results

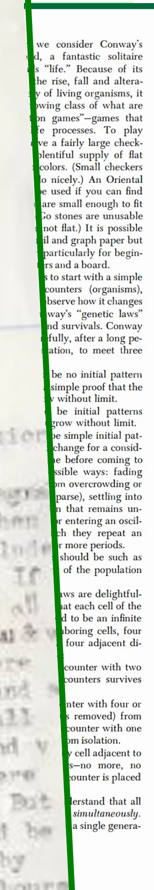
The fantastic combinations of John Conway's new solitaire game "life"

by Martin Gardner

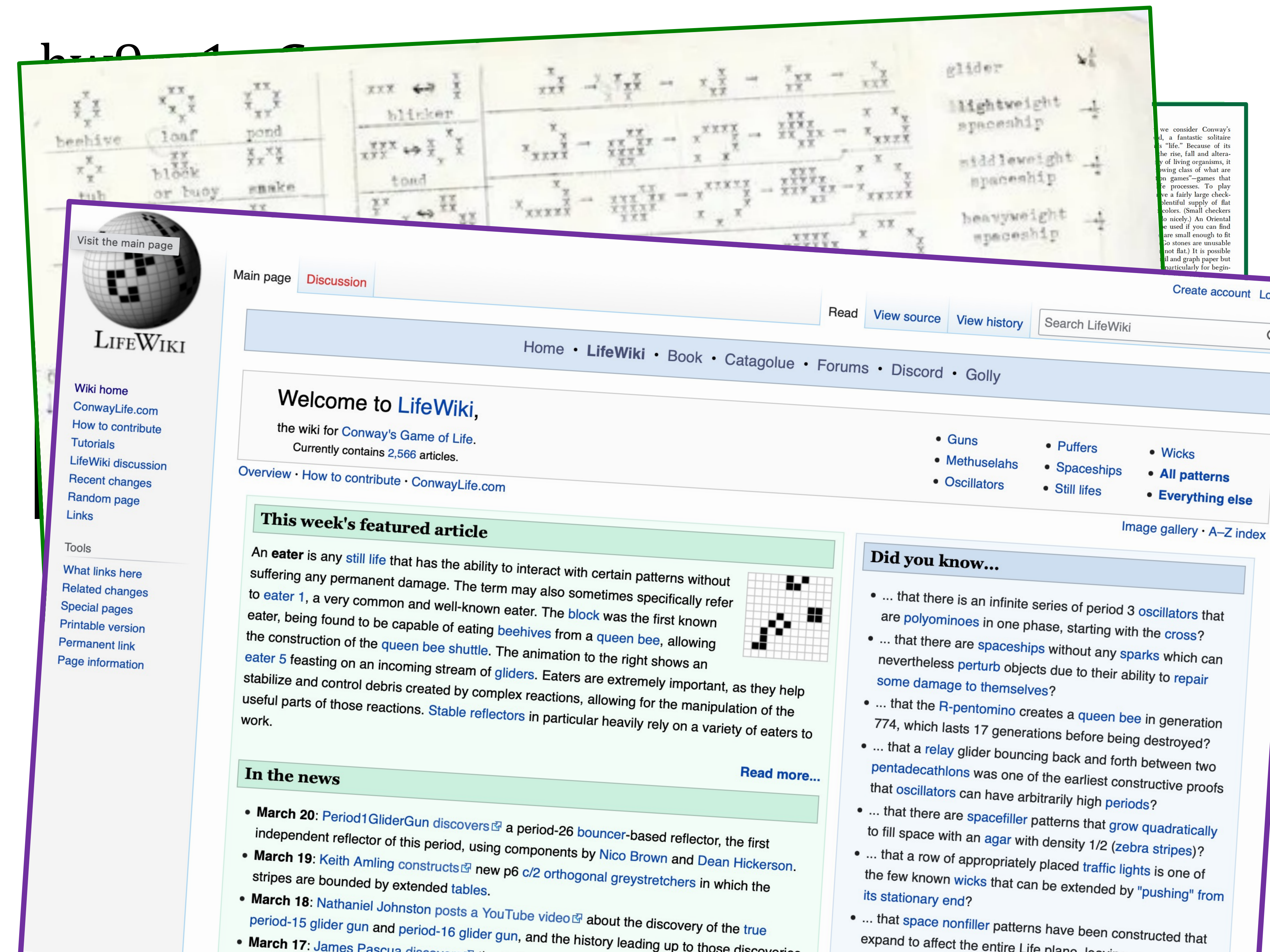
not really solitaire...

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





really, it's a ***zero-player*** game!

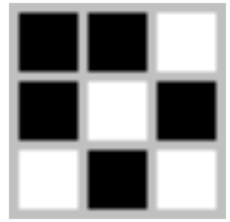


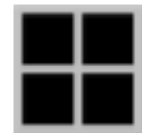
beehive
loaf
pond
tub
block or buoy
snake
blinker
toad
glider
lightweight spaceship
middleweight spaceship
heavyweight spaceship
Visit the main page
LifeWiki
Main page
Discussion
Read
View source
View history
Search LifeWiki
Create account
Home • LifeWiki • Book • Catagolue • Forums • Discord • Golly
Wiki home
ConwayLife.com
How to contribute
Tutorials
LifeWiki discussion
Recent changes
Random page
Links
Tools
What links here
Related changes
Special pages
Printable version
Permanent link
Page information
Welcome to LifeWiki,
the wiki for Conway's Game of Life.
Currently contains 2,566 articles.
Overview · How to contribute · ConwayLife.com
Guns
Methuselahs
Oscillators
Puffers
Spaceships
Still lifes
Wicks
All patterns
Everything else
Image gallery · A–Z index
This week's featured article
An eater is any still life that has the ability to interact with certain patterns without suffering any permanent damage. The term may also sometimes specifically refer to eater 1, a very common and well-known eater. The block was the first known eater, being found to be capable of eating beehives from a queen bee, allowing the construction of the queen bee shuttle. The animation to the right shows an eater 5 feasting on an incoming stream of gliders. Eaters are extremely important, as they help stabilize and control debris created by complex reactions, allowing for the manipulation of the useful parts of those reactions. Stable reflectors in particular heavily rely on a variety of eaters to work.
In the news
Read more...
March 20: Period1GliderGun discovers a period-26 bouncer-based reflector, the first independent reflector of this period, using components by Nico Brown and Dean Hickerson.
March 19: Keith Amling constructs new p6 c/2 orthogonal greystretchers in which the stripes are bounded by extended tables.
March 18: Nathaniel Johnston posts a YouTube video about the discovery of the true period-15 glider gun and period-16 glider gun, and the history leading up to those discoveries
March 17: James Pascua
Did you know...
... that there is an infinite series of period 3 oscillators that are polyominoes in one phase, starting with the cross?
... that there are spaceships without any sparks which can nevertheless perturb objects due to their ability to repair some damage to themselves?
... that the R-pentomino creates a queen bee in generation 774, which lasts 17 generations before being destroyed?
... that a relay glider bouncing back and forth between two pentadecathlons was one of the earliest constructive proofs that oscillators can have arbitrarily high periods?
... that there are spacefiller patterns that grow quadratically to fill space with an agar with density 1/2 (zebra stripes)?
... that a row of appropriately placed traffic lights is one of the few known wicks that can be extended by "pushing" from its stationary end?
... that space nonfiller patterns have been constructed that expand to affect the entire Life plane

hw9pr1 lab: ***Creating Life***

Stable configurations:

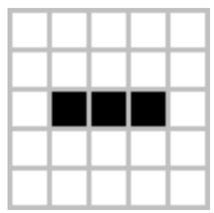
"rocks"



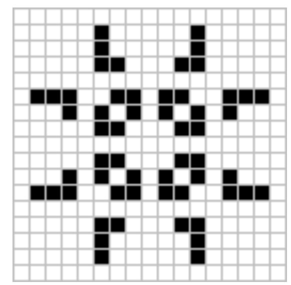


Periodic

"plants"



period 2

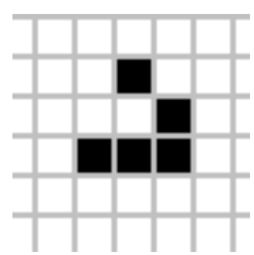


period 3

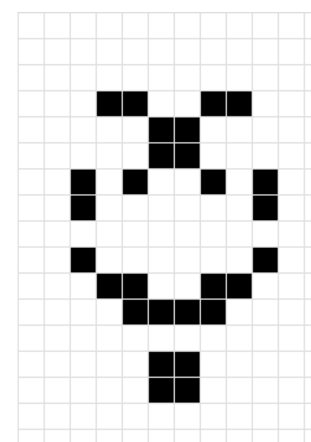
Self-propagating

"animals"

glider



Copperhead: 2016



Life lessons…

- Incredibly simple rules can allow
  arbitrarily complex
  computational structures

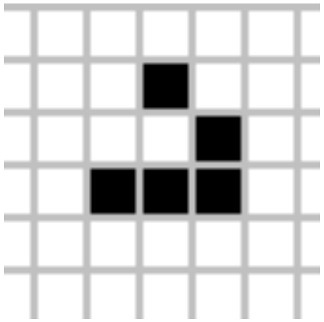
- Just because you know
  "how it works" (at a low level)
  doesn't mean you know
  "what it is" or "what it's *really doing*" (at a high level)

Life @ HMC?

hw9pr1 lab: ***Creating Life***

Many life configurations expand forever...

"Gosper glider generator" (or "gun")

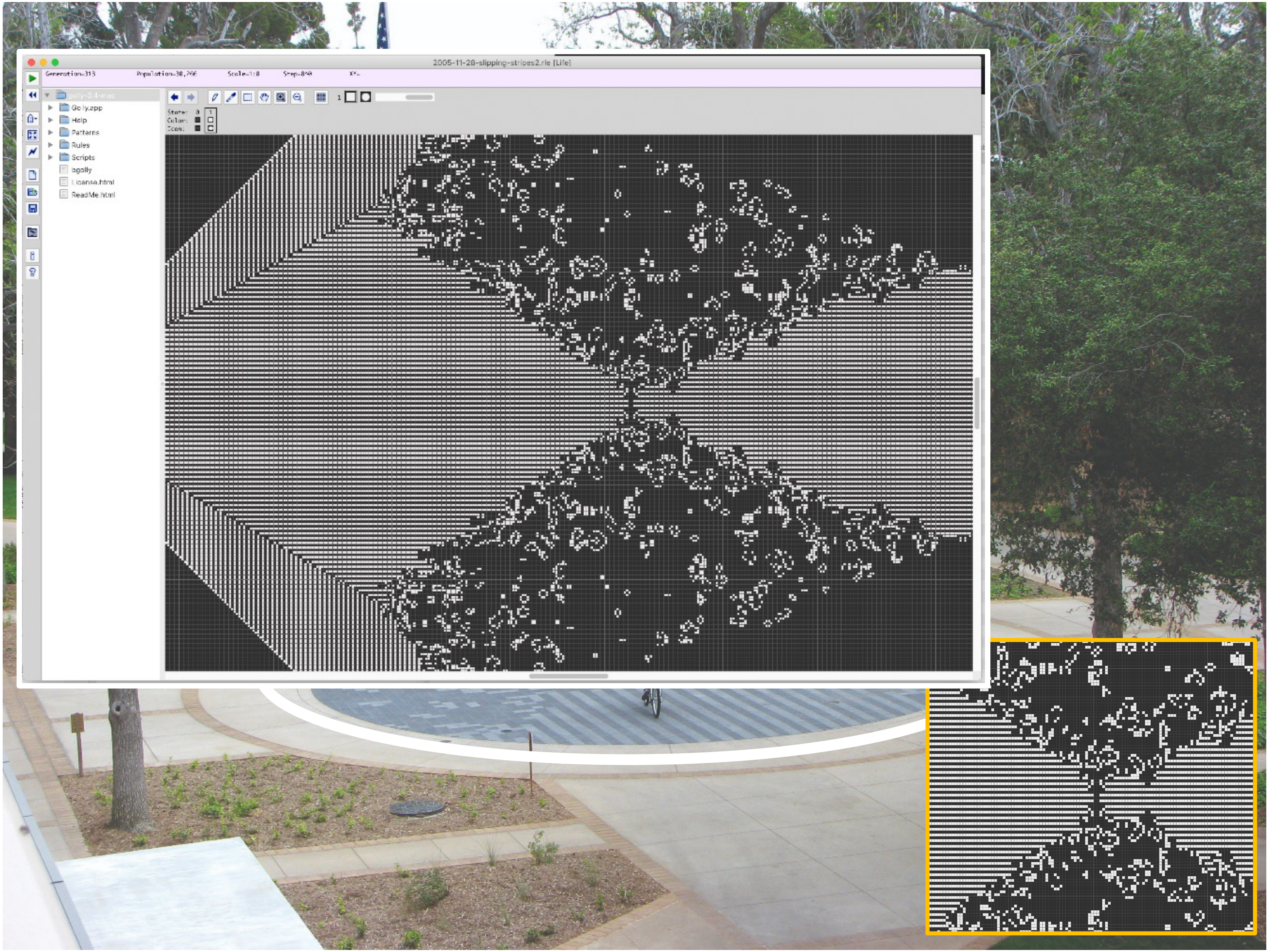


"glider"

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/


2005-11-28-slipping-stripes2.rle [Life]
Generation=313
Scale=1:8
Golly.app
Help
Patterns
Rules
Scripts
bgolly
License.html
ReadMe.html

Life @ HMC!

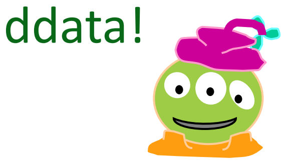
with 42' diameter!

2D

Data

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

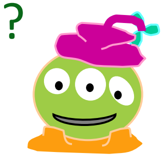
2D **Data**



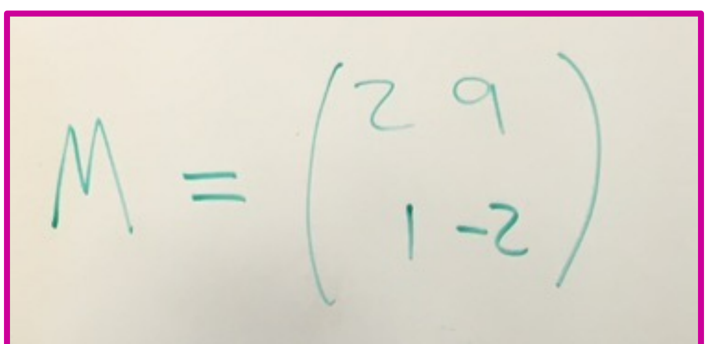
Math + CS: shareful siblings!

2D data

rows? cols?



```
M[0][0]    M[0][1]    M[1][0]    M[1][1]

M = [[2,9], [1,-2]]

      M[0]        M[1]
```



2D data

rows! cols!

M[0][0] M[0][1] M[1][0] M[1][1]

```
M = [[2,9], [1,-2]]
```

M[0] M[1]

Handling 2D data requires <u>*no new rules*</u>!

Mutable vs Immutable

Diversion!

Which of these make sense?

- `X = 42`
- `42 = 7`
- `"wow" = "what"`
- `"wow"[1] = '?'`
- `[3.14, 2.17, 1.44][1] = 1.62`

Mutable vs Immutable

Diversion!

Which of these make sense?

You can't alter a number or a string, only make a new one

- `X = 42`
- `42 = 7`
- `"wow" = "what"`
- `"wow"[1] = '?'`
- `[3.14, 2.17, 1.44][1] = 1.62`

You can modify **variables** *and* you can modify **list elements**!

Example: Double all the values

Three ways:

```
for i in range(len(L)):
    L[i] *= 2
```

Change elements of L

```
L = [x*2 for x in L]
```

Store new list in L

```
M = [x*2 for x in L]
```

Make new var, M

Looking at Pythons innards!

From the Python documentation…

`id`(*object*)

- Return the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.

  For immutable objects, operations that compute new values may return a pre-existing object with the same value, while for mutable objects this is not allowed

CPython implementation detail: This is the address of the object in memory.

Looking at Pythons innards!

```python
def isSameMemory(x, y):
    print("The id of x is", id(x))
    print("The id of y is", id(y))
    if id(x) == id(y):
        print("=> They are the same object")
    else:
        print("=> They are different objects")
```

From the Python documenta

`id`(*object*)

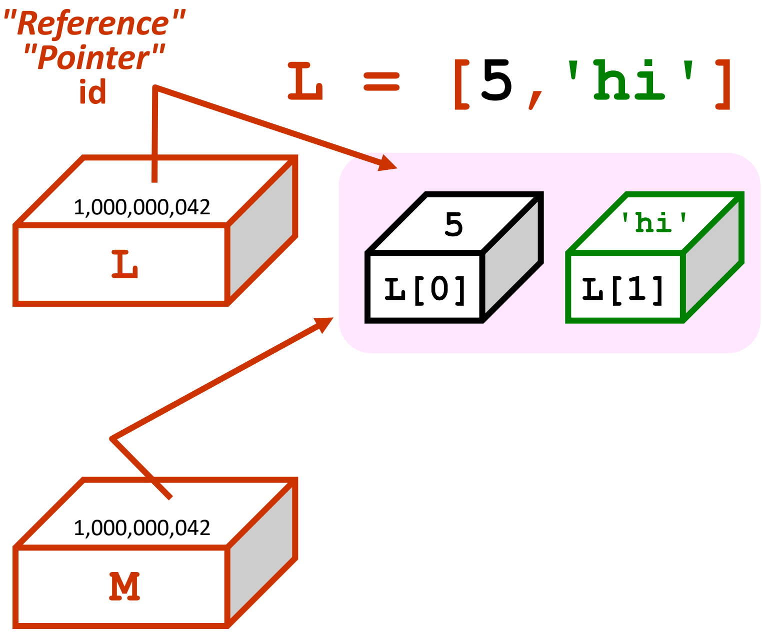
- Return the “identity” of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.

  For immutable objects, operations that compute new values may return a pre-existing object with the same value, while for mutable objects this is not allowed

CPython implementation detail: This is the address of the object in memory.

Shallow vs. Deep

Python's two methods for copying data



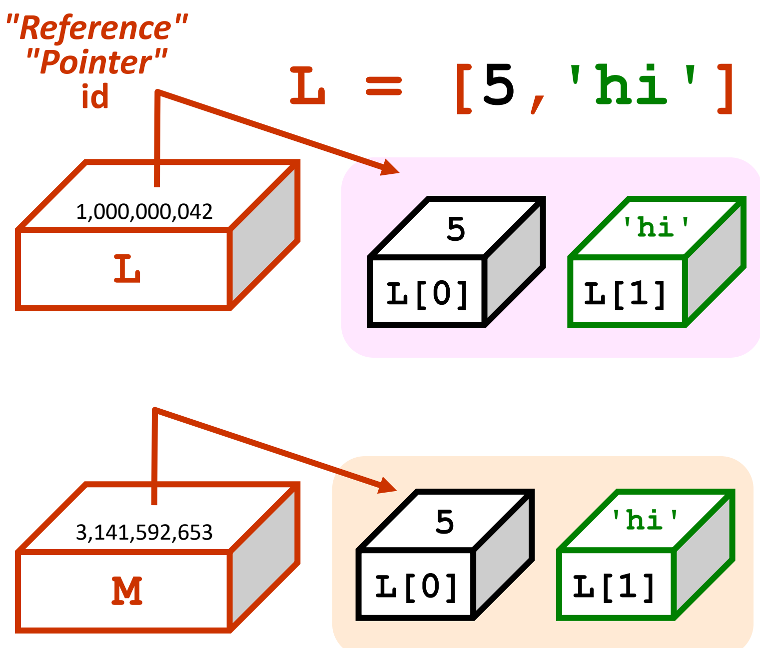

```
L = [5,'hi']
M = L
M[0] = 42
```

What's L[0] ?!

L and M are the same *memory address*

Shallow vs. **Deep**

Python's two methods for copying data



L and M are *different* memory addresses

```
L = [5,'hi']
M = L[:]
M[0] = 42
```

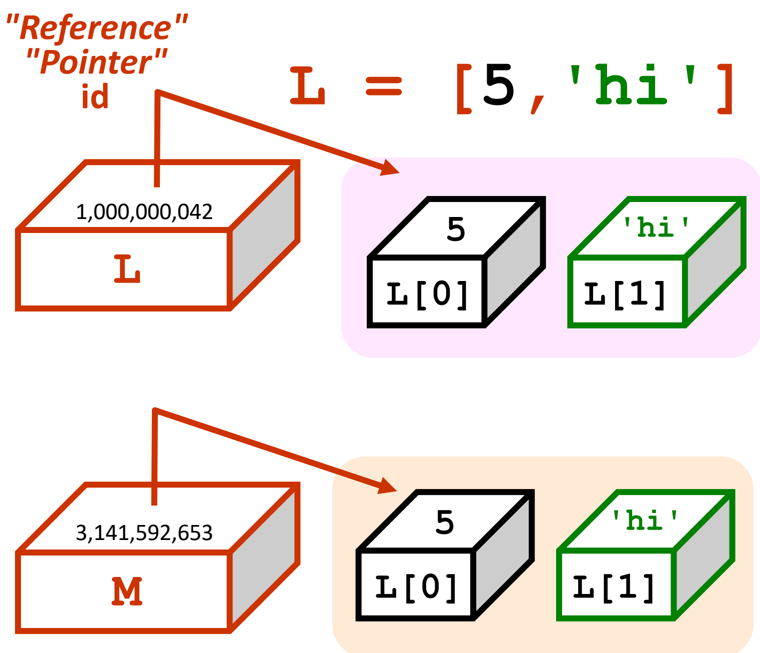
What's L[0] ?!

slicing makes a **copy**

but only one-level deep

Shallow vs. **Deep**

Python's two methods for copying data



L and M are *different* memory addresses

```
from copy import *

L = [5,'hi']

M = deepcopy(L)

M[0] = 42
```

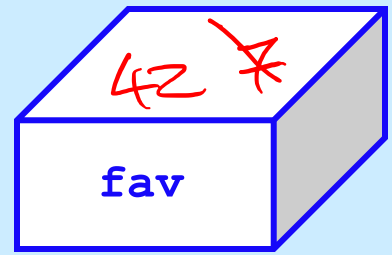
What's L[0] ?!

deepcopy is deep!

shallow

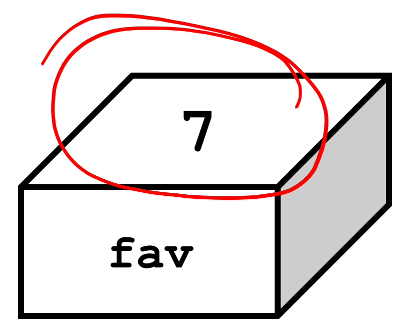
Python functions: *pass by copy*

```
def conform(fav)

    fav = 42
    return fav
```



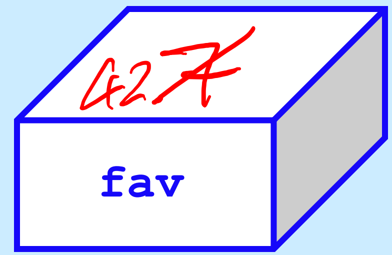
this line is the "abstraction boundary" between conform and main

```
def main()

    print(" Welcome! ")

    fav = 7
    fav = conform(fav)

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



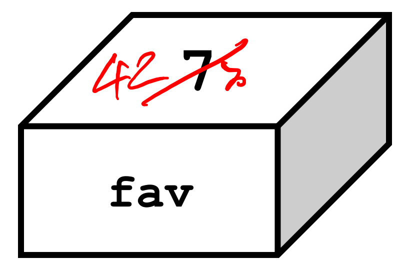
Python functions: *pass by* ^shallow *copy*

```
def conform(fav)

    fav = 42
    return fav
```



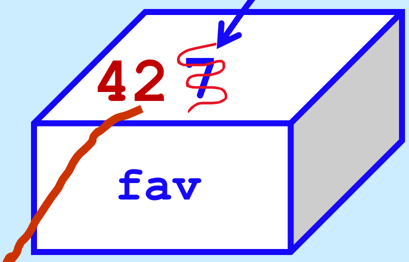
this line is the "abstraction boundary" between conform and main

```
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**

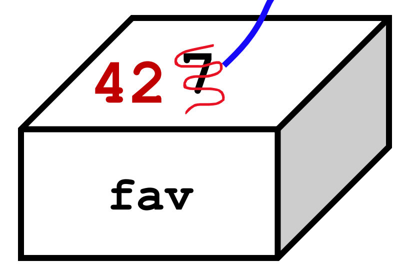
this line is the "abstraction boundary" between conform and main

```
def main()

    print(" Welcome! ")

    fav = 7
    fav = conform(fav)

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

42 is returned

"pass by copy" means the contents of **fav** are *copied* to **fav**

42

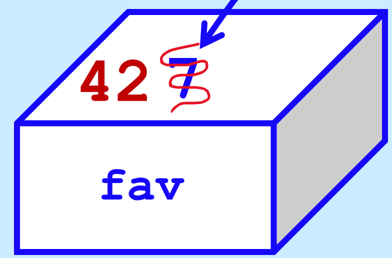


The original 7 is "clobbered."

What if we didn't have the underlined part re-assigning fav?

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

```python
def conform(fav)

    fav = 42
    return fav
```

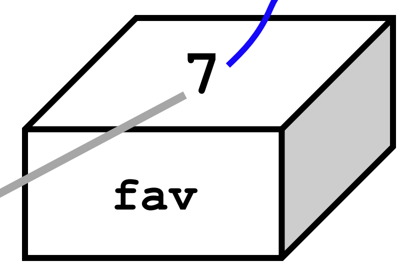


copy of **fav**

this line is the "abstraction boundary" between conform and main

```python
def main()

    print(" Welcome! ")

    fav = 7
    conform(fav)

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

"pass by copy" means the contents of **fav** are *copied* to **fav**



No assignment here!

7

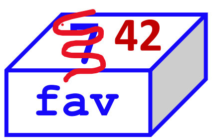
The original 7 is still here – and still used.

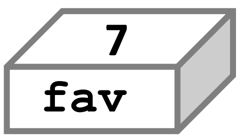
Name(s) ______________________________

Rules, Rules, Rules!?

Trace each function. What do **conformOne** and **conformTwo** produce?

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

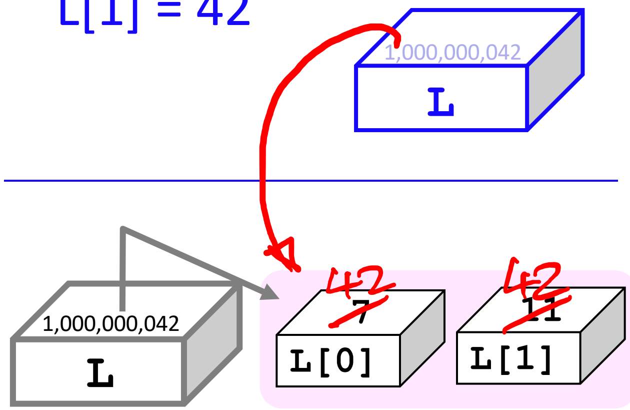





```
def main()
    fav = 7
    conform(fav)
    print(fav)
```

7

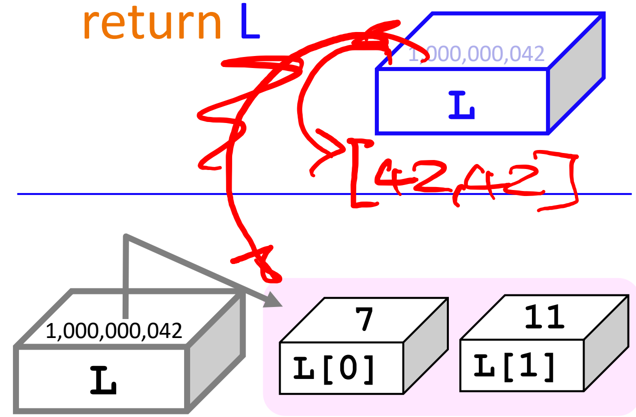
```
def conformOne(L)
    L[0] = 42
    L[1] = 42
```




```
def mainOne()
    L = [7,11]
    conformOne(L)
    print(L)
```

[42,42]

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




```
def mainTwo()
    L = [7,11]
    conformTwo(L)
    print(L)
```

[7, 11]

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 and strings are IMMUTABLE – they can't be changed
(but the "box" that *holds* them can be!)

2D data?

Even with 3 eyes,
this looks 1d!

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

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

~ pure composition

2D data?

Even with 3 eyes,
this looks 1d!

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

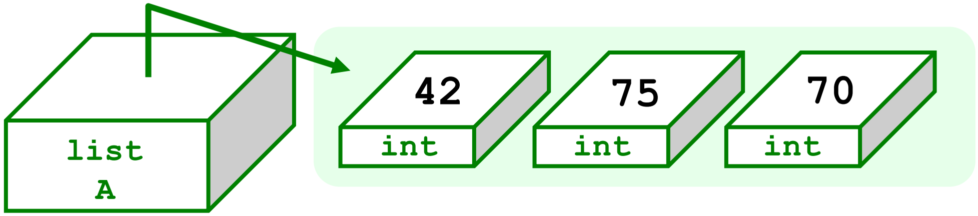
All and only the rules that govern 1D data apply here – no new rules to learn!

~ pure composition

What does `A` "look like" ?

1D data ~ Lists

```
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!***

2D data ~ Lists *of* Lists

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

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

I think I've seen this story before!

Where's 3?

`len(A)`

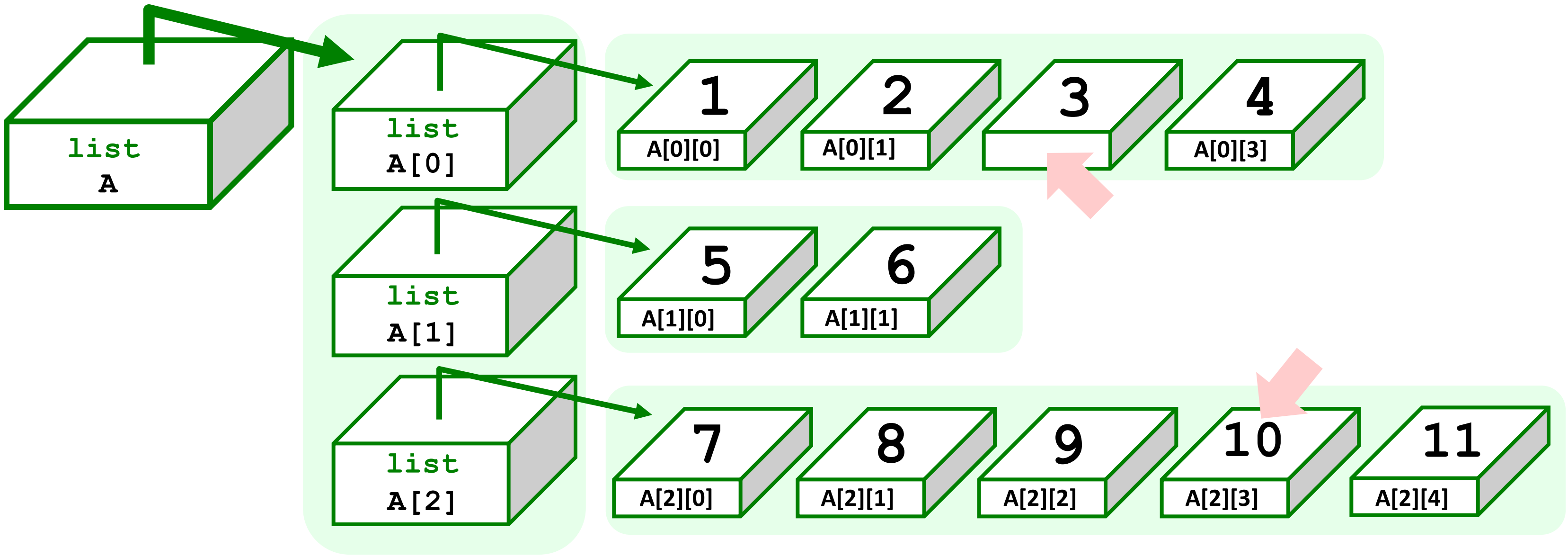
`len(A[0])`

`len(A[1])`

Replace 10 with 42.

2D data as *Lists of Lists*

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



What are 3's "coordinates"?

`len(A)`

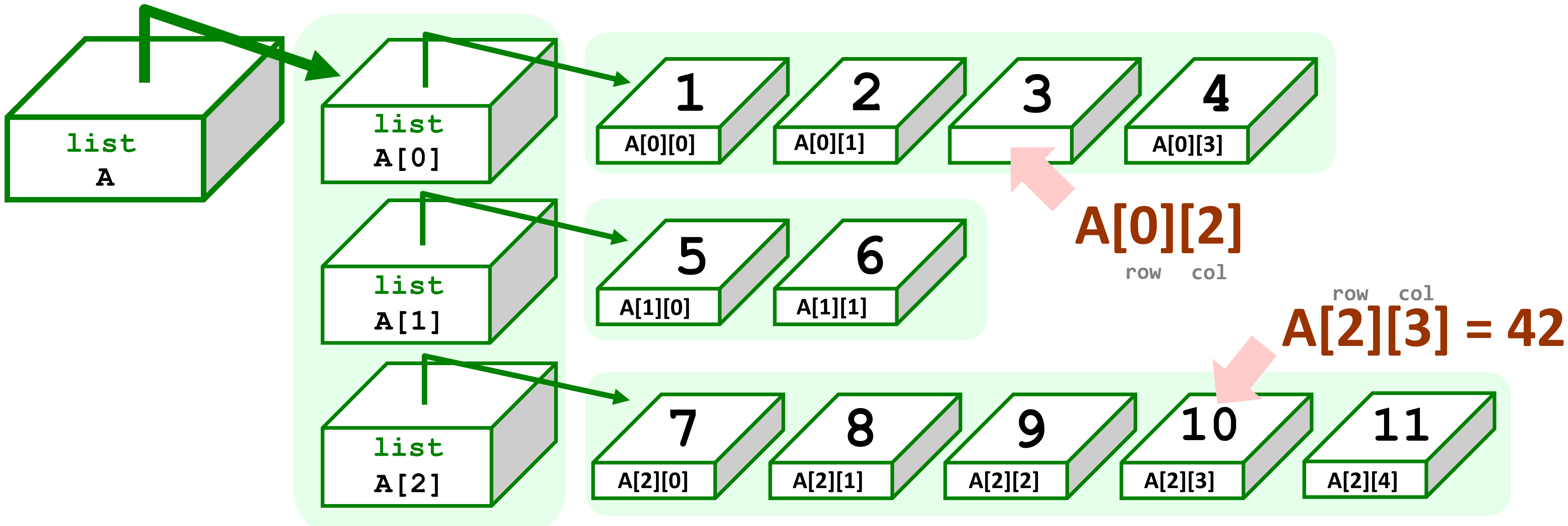
`len(A[0])`

`len(A[1])`

Replace 10 with 42.

2D data as *Lists of Lists*

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



row col
A[0][2]

What are 3's "coordinates"?

3 *rows*

`len(A)`

`len(A[0])` 4 *cols*

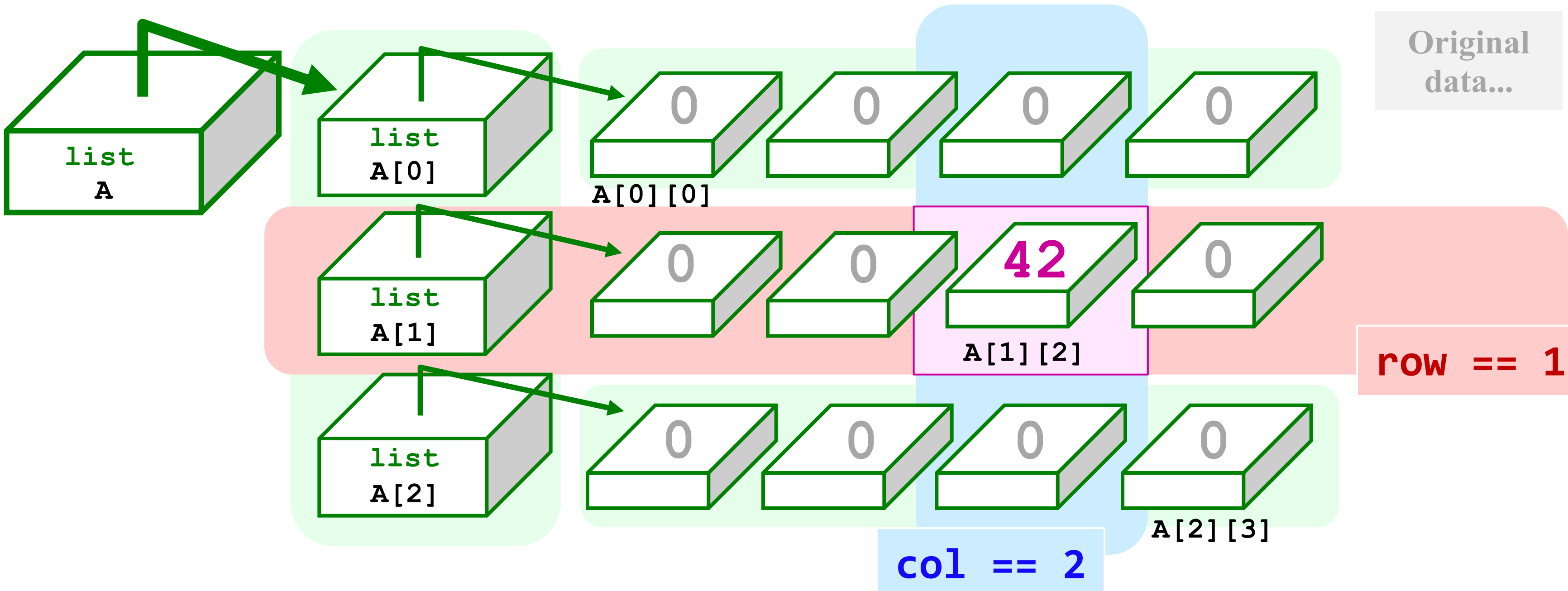
`len(A[1])` 2 *cols*

Replace 10 with 42.

A[2][3] = 42
row col

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

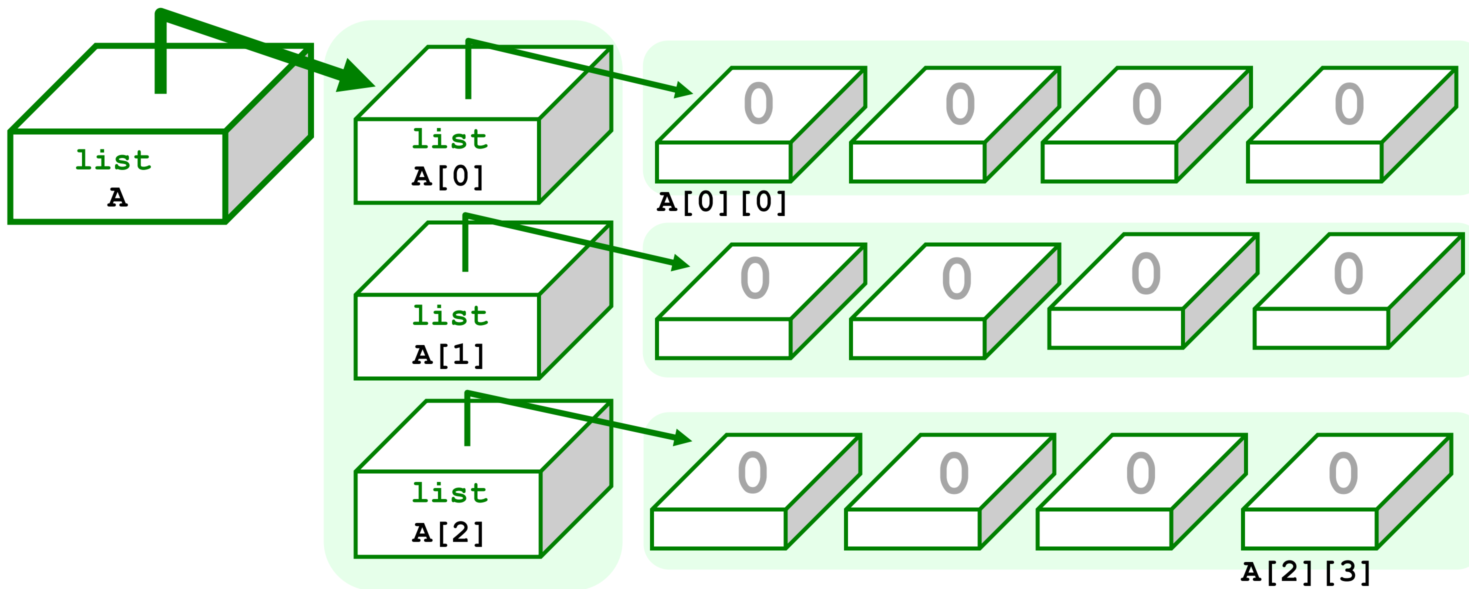
col c

A[r][c] = value

Rectangular 2D data

Original data...

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




```
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
```

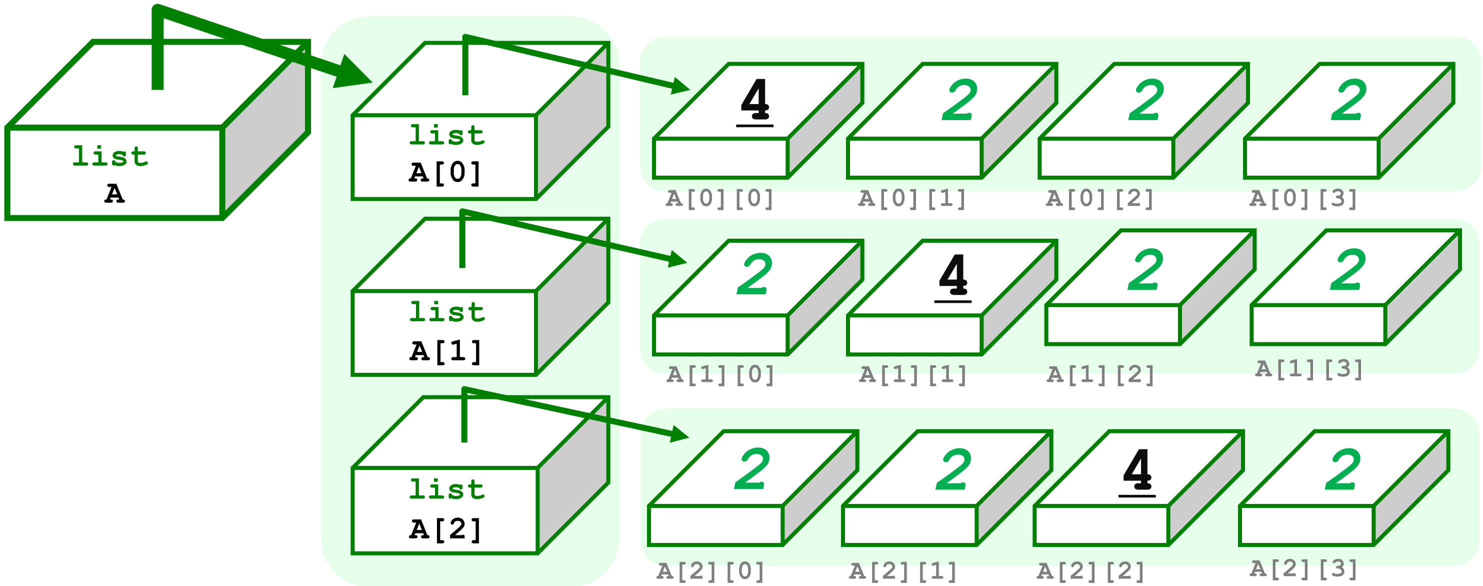
Nested Loops ~ 2d Data

How many 4's?
How many 2's?

Rectangular 2D data

Changed data ...

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




```
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

How many 4's?

How many 2's?

2 North!

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

```
def two_in_a_row_North(A):
    """ let's see... """
    NROWS = len(A)
    NCOLS = len(A[0])
    B = deepcopy( A )

    for r in range( 0,NROWS ):
       for c in range( 0,NCOLS ):

          if r == 0:
                B[r][c] = False
          elif A[r][c] ==                    ★ North :
                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

	col 0	col 1	col 2	col 3
row 0	F	F	F	F
row 1	F	T	F	F
row 2	T	F	T	F

What `elif` will produce these?

Extra:

How could we *change the starred code above* ★ to check for two-in-a-row EAST or DIAGONALLY !?

East: `A[r][c] ==` ★ East

N.East: `A[r][c] ==` ★ N.East

what would check these?

2 North!

Answers

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

```
def two_in_a_row_North(A):
    """ let's see... """
    NROWS = len(A)
    NCOLS = len(A[0])
    B = deepcopy( A )

    for r in range( 0,NROWS ):
       for c in range( 0,NCOLS ):

          if r == 0:
                B[r][c] = False
          elif A[r][c] == A[r-1][c]:
                B[r][c] = True
          else:
                B[r][c] = False
```

★ North

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

	col 0	col 1	col 2	col 3
row 0	F	F	F	F
row 1	F	T	F	F
row 2	T	F	T	F

What `elif` will produce these?

Extra:

How could we *change the starred code above* ★ to check for two-in-a-row EAST or DIAGONALLY !?

East: `A[r][c] == A[r][c+1]` ★ East

N.East: `A[r][c] == A[r-1][c+1]` ★ N.East

what would check these?

2 North!

Answers

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

```
def two_in_a_row_North(A):
    """ let's see... """
    NROWS = len(A)
    NCOLS = len(A[0
    B = deepcopy( A

    for r in range(
        for c in rang

            if r == 0:
                B[r][c] = False
            elif A[r][c] == A[r-1][c]:
                B[r][c] = True
            else:
                B[r][c] = False
```

Use as hw9pr2's starting point... !

★ North

B

	col 0	col 1	col 2	col 3
row 0	F	F	F	F
row 1	F	T	F	F
row 2	T	F	T	F

What `elif` will produce these?

Extra:

How could we *change the starred code above* ★ to check for two-in-a-row EAST or DIAGONALLY !?

East: `A[r][c] == A[r][c+1]` ★ East

N.East: `A[r][c] == A[r-1][c+1]` ★ N.East

what would check these?

What about N-in-a-row?

Let's try it...

```
              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 does **not** "wrap around"

```
                 checker  start row  start col  LoL
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)   →
```

First, try it by eye... *... then, by Python!*

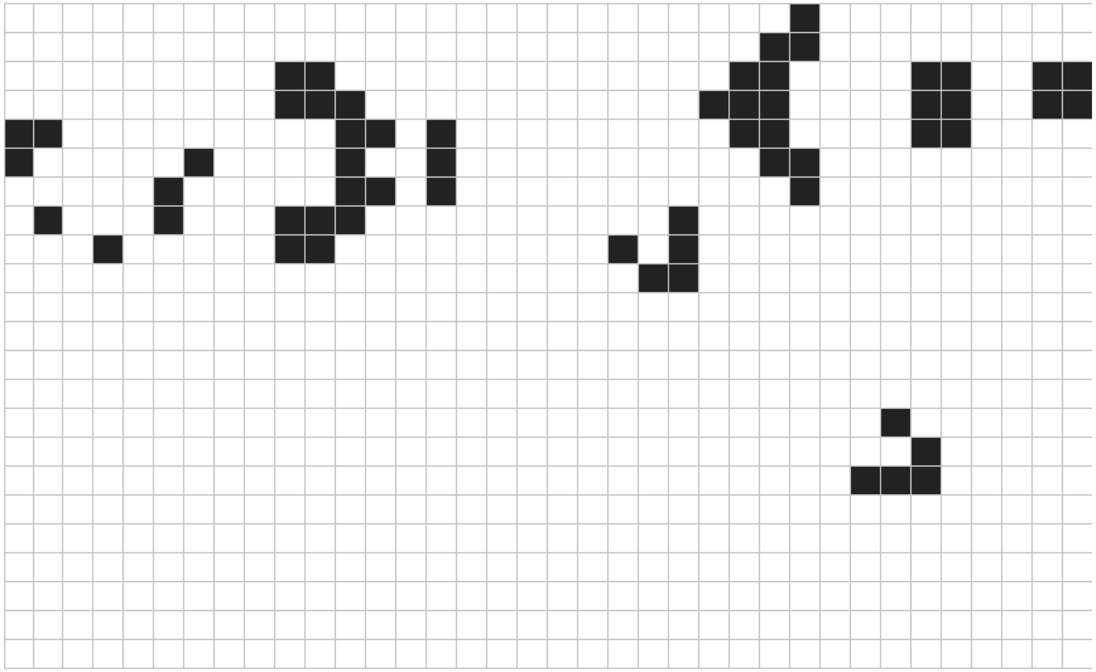
col 0, col 1, col 2, col 3, col 4

```
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 does **not** "wrap around"

checker, start row, start col, LoL

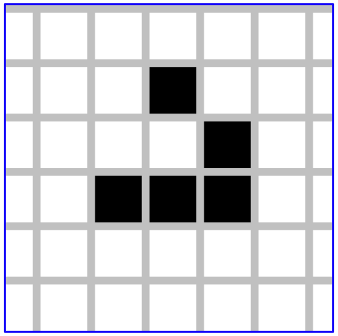
```
inarow_3east('X', 1, 0, A)        →  True
inarow_3south('O', 0, 4, A)       →  True
inarow_3southeast('X', 2, 3, A)   →  False
inarow_3northeast('X', 3, 1, A)   →  False
```



This week we're

Lifing it up

in lab!

so  on over...

glide