Outline

1. The Game of Life
   - simulating cellular growth
   - cellular automata
2. Representing the Grid
   - lists of lists
   - running at the command prompt
   - counting life neighbors
3. a GUI for the Game of Life
   - defining the layout of the GUI
   - defining the functionality of the GUI

MCS 260 Lecture 34
Introduction to Computer Science
Jan Verschelde, 6 April 2016

cellular automata
the game of life

1. The Game of Life
   - simulating cellular growth
   - cellular automata
2. Representing the Grid
   - lists of lists
   - running at the command prompt
   - counting life neighbors
3. a GUI for the Game of Life
   - defining the layout of the GUI
   - defining the functionality of the GUI

The Game of Life

simulating cellular growth

Our application is the "game of life" defined by the mathematician John Horton Conway.

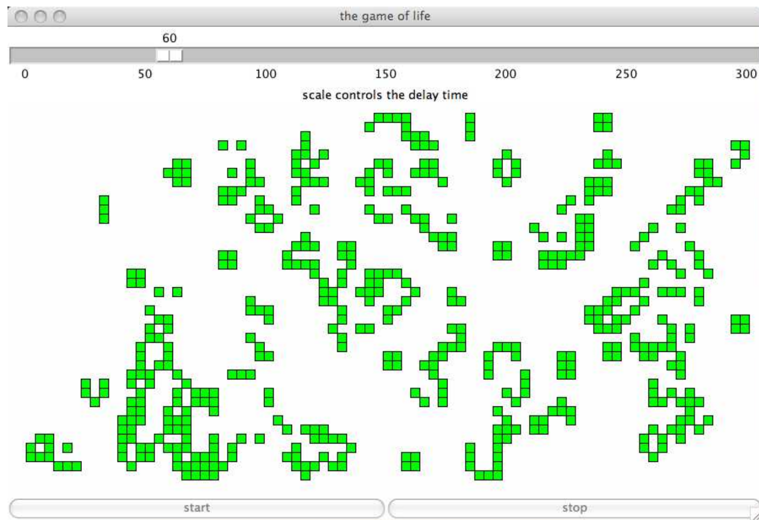
Consider a rectangular grid of squares.
At any square, there can be a living cell or not.

The rules of the game determine whether life or not:

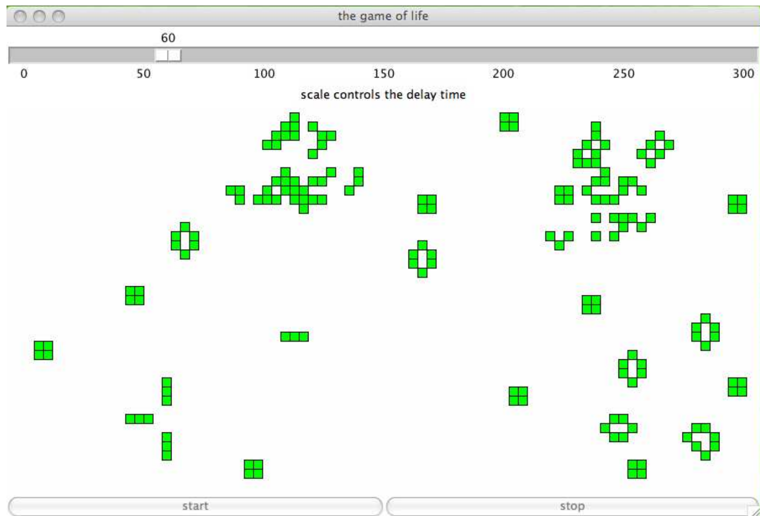
1. An empty cell becomes alive if it has 3 neighbors.
2. A living cell can either die or survive, as follows:
    1. die by loneliness, if the cell has one or no neighbors;
    2. die by overpopulation, if the cell has ≥ 4 neighbors;
    3. survive, if the cell has two or three neighbors.

Complex patterns may already occur from simple rules.

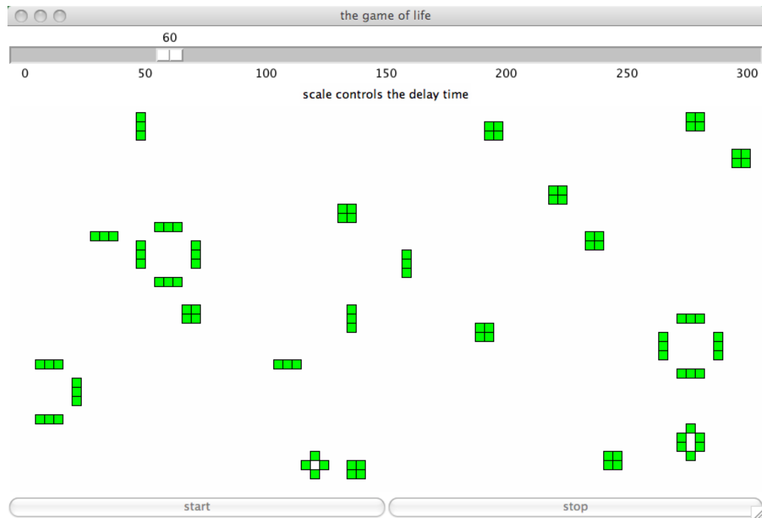
early in the game



formation of patterns



patterns at the end



cellular automata
the game of life

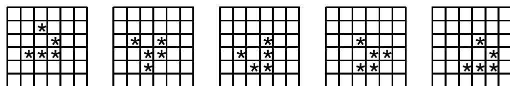
1. The Game of Life
   - simulating cellular growth
   - cellular automata
2. Representing the Grid
   - lists of lists
   - running at the command prompt
   - counting life neighbors
3. a GUI for the Game of Life
   - defining the layout of the GUI
   - defining the functionality of the GUI

Cellular Automata

A cellular automaton is a discrete model of cells.
Each cell has a state (alive or dead) on a regular grid.
Rules define the transition of the states.

Given rules and configurations of cells,
one discovers patterns, such as gliders.

A glider in the game of life (* in live cell):



While cellular automata simulate cell growth,
they give ways to explore the nature of computation.
Stephen Wolfram: *a new kind of science*, 2002.

Python Scripts for the Game

Ingredients in the program:

1. representing the rectangular grid:
   - we use a grid of r rows and c columns,
   - define the rule of an element of the grid.
2. the Graphical User Interface (GUI):
   - determine the layout of the GUI,
   - functionality: draw cells in an animation.

cellular automata
the game of life

1. The Game of Life
   - simulating cellular growth
   - cellular automata
2. Representing the Grid
   - lists of lists
   - running at the command prompt
   - counting life neighbors
3. a GUI for the Game of Life
   - defining the layout of the GUI
   - defining the functionality of the GUI

lists of lists

We can define the grid as a lists of rows:

- every row is a list of boolean values,
- every row has the same number of elements.

Making a list of 5 `False` values with a list comprehension:

```
>>> L = [False for _ in range(5)]
```

A double list comprehension to generate a list of 3 rows,
every row has 2 elements:

```
>>> g = [[False for _ in range(2)] for _ in range(3)]
>>> for row in g: print(row)
...
[False, False]
[False, False]
[False, False]
```

printing a list of lists

More list comprehensions:

```
>>> g = [[False for _ in range(2)] for _ in range(3)]
>>> for row in g: print(row)
...
[False, False]
[False, False]
[False, False]
>>> for row in g: print([' %d' % item for item in row])
...
[' 0', ' 0']
[' 0', ' 0']
[' 0', ' 0']
>>>
```

The decimal format of a boolean gives a 0 or a 1.

printing the grid

```
def write(grid):
    """
    Writes the lists of lists of booleans
    in grid using 0 for False and 1 for True.
    """
    for row in grid:
        for item in row:
            print(' %d' % item, end='')
        print('')
```

Observe:

- the double loop for a two dimensional structure,
- suppressing the newline in `print( ..  , end='')`.

cellular automata
the game of life

1. The Game of Life
   - simulating cellular growth
   - cellular automata
2. Representing the Grid
   - lists of lists
   - running at the command prompt
   - counting life neighbors
3. a GUI for the Game of Life
   - defining the layout of the GUI
   - defining the functionality of the GUI

at the command prompt

```
$ python game.py
give number of rows : 10
give number of columns : 20
Initial State :
 1 1 0 0 1 1 0 1 0 0 1 0 1 0 1 1 0 0 1 0
 1 0 0 1 1 0 1 0 1 0 1 1 1 1 0 1 0 1 1 0
 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 1 1 0 0
 0 1 1 1 0 1 0 1 0 0 0 1 0 0 0 1 1 0 1 1
 0 1 0 0 0 0 0 0 1 0 1 1 0 1 1 0 0 1 0 1
 0 1 0 1 0 0 0 0 1 1 1 1 0 1 1 1 1 1 1 1
 0 1 0 1 1 1 1 1 1 0 0 0 0 1 0 1 1 1 1 1
 1 1 1 0 0 1 0 0 0 0 1 0 1 1 0 1 0 0 1 0
 1 1 0 1 0 0 1 1 1 1 0 0 0 1 0 0 0 0 1 1
 0 1 0 0 0 0 1 0 1 0 1 1 1 0 0 0 0 0 0 1
continue ? (y/n)
```

the script game.py

The function `main()` of game.py simulates the game of life as long as the user wants to continue.

The last line of `game.py` is

```
if __name__ == "__main__":
    main()
```

In addition to writing the grid,
`game.py` defines a module exporting

1. initialize a random grid `grid`,
2. computing the neighbors of `grid[i][j]`,
3. applying the rule to update the `grid`.

In `life.py`, the GUI for the game of life:

```
import game
```

initializing with random values

```
from random import randint

def random_grid(rows, cols):
    """
    Returns a list of rows.
    Every row has as many random booleans
    as the value of cols.
    """
    result = []
    for _ in range(rows):
        result.append([bool(randint(0, 1)) \
           for _ in range(cols)])
    return result
```

cellular automata
the game of life

1. The Game of Life
   - simulating cellular growth
   - cellular automata
2. Representing the Grid
   - lists of lists
   - running at the command prompt
   - counting life neighbors
3. a GUI for the Game of Life
   - defining the layout of the GUI
   - defining the functionality of the GUI

counting life neighbors

[i-1,j-1]	[i-1,j]	[i-1,j+1]
[i,j-1]	[i,j]	[i,j+1]
[i+1,j-1]	[i+1,j]	[i+1,j+1]

```
k = 0
if A[i-1][j]: k += 1; if A[i-1][j-1]: k += 1
if A[i+1][j]: k += 1; if A[i+1][j-1]: k += 1
if A[i][j-1]: k += 1; if A[i-1][j+1]: k += 1
if A[i][j+1]: k += 1; if A[i+1][j+1]: k += 1
```

if `0 < i < len(A)-1` and `0 < j < len(A[i])-1`

applying the rule

```
def update(grid):
    """
    Applies the rule of the game of life to grid
    and returns a new grid.
    """
    result = [[False for _ in row] for row in grid]
    for i in range(len(grid)):
        for j in range(len(grid[i])):
            nbr = neighbors(grid, i, j)
            if grid[i][j]:
                if nbr < 2 or nbr > 3:
                    result[i][j] = False
                else:
                    result[i][j] = True
            else:
                if nbr == 3:
                    result[i][j] = True
    return result
```

cellular automata
the game of life

1. The Game of Life
   - simulating cellular growth
   - cellular automata
2. Representing the Grid
   - lists of lists
   - running at the command prompt
   - counting life neighbors
3. a GUI for the Game of Life
   - defining the layout of the GUI
   - defining the functionality of the GUI

a GUI for the Game of Life

Components of the GUI:

1. a scale to change the drawing speed
   The variable defined by the scale is the time the GUI waits to update the canvas.
2. a label to document the scale
3. a canvas to draw the cells
   The canvas has three parameters:
   1. the number of rows in the grid,
   2. the number of columns in the grid,
   3. the size of each square.

We set the parameters at the start of the GUI.

start of the code for the GUI

```
from tkinter import Tk, Scale, Canvas, Button, Label
from tkinter import IntVar, W, E, N, S, ALL
import game

class GameOfLife(object):
    """
    GUI to the game of life
    """
    def __init__(self, wdw, r, c, m, delay):
```

data attributes

```
def __init__(self, wdw, r, c, m, delay):
    """
    The GUI has a canvas, scale, start and stop button.
    The scale controls the speed of the animation,
    initialized by the variable delay on input.
    The canvas has r rows and c columns of squares of
    size m.
    """
    wdw.title('the game of life')
    self.rows = r        # number of rows on canvas
    self.cols = c        # number of columns on canvas
    self.size = m        # size of the squares on canvas
    self.dly = IntVar()
    self.grow = False    # state of the animation
    self.grid = []
```

layout of the GUI

```
self.scl = Scale(wdw, orient='horizontal', \
    from_=0, to=300, tickinterval=50, resolution=1, \
    length=self.cols, variable=self.dly)
self.scl.grid(row=0, column=0, columnspan=2, \
    sticky=W+E+N+S)
self.scl.set(delay)
self.cnv = Canvas(wdw, width=m*self.cols+2*m, \
    height=m*self.rows+2*m,bg ='white')
self.lbl = Label(wdw, \
    text="scale controls the delay time")
self.lbl.grid(row=1, column=0, columnspan=2)
self.cnv.grid(row=2, column=0, columnspan=2)
self.bt0 = Button(wdw, text='start', \
    command = self.start)
self.bt0.grid(row=3, column=0, sticky=W+E)
self.bt1 = Button(wdw, text='stop', \
    command = self.stop)
self.bt1.grid(row=3, column=1, sticky=W+E)
```

cellular automata
the game of life

1. The Game of Life
   - simulating cellular growth
   - cellular automata
2. Representing the Grid
   - lists of lists
   - running at the command prompt
   - counting life neighbors
3. a GUI for the Game of Life
   - defining the layout of the GUI
   - defining the functionality of the GUI

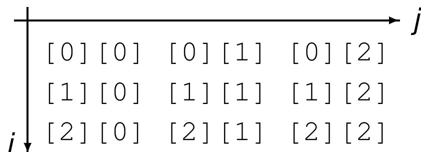
starting and stopping

```
def start(self):
    """
    At the start of the animation, a random
    configuration of cells is generated and
    the animate method is called.
    """
    self.grid = game.random_grid(self.rows, self.cols)
    self.grow = True
    self.draw_cells()
    self.animate()

def stop(self):
    """
    Stops the animation.
    """
    self.grow = False
```

grid to canvas

Mapping the grid to canvas coordinates:



The row index is i
while the column index is j.

On canvas: y is i and x is j.

drawing cells

First all is wiped out from canvas,
then for every living cell, a green rectangle is drawn.

```
def draw_cells(self):
    """
    Draws the cells on canvas.
    """
    msz = self.size
    self.cnv.delete(ALL)
    for i in range(self.rows):
        for j in range(self.cols):
            if self.grid[i][j]:
                (xpt, ypt) = (msz*(j + 1), msz*(i + 1))
                self.cnv.create_rectangle(xpt, ypt, \
                    xpt+msz, ypt+msz, fill="green")
```

the animation

As long as `self.grow` is `True`:

1. we update the grid of cells
2. wait to refresh the canvas
3. draw the cells and update the canvas

```
def animate(self):
    """
    Performs the animation.
    """
    while self.grow:
        self.grid = game.update(self.grid)
        self.cnv.after(self.dly.get())
        self.draw_cells()
        self.cnv.update()
```

the `main()` in `life.py`

```
def main():
    """
    Sets the dimensions of the canvas,
    number of rows, columns, and size of squares,
    before launching the main event loop of the GUI.
    """
    top = Tk()
    # rraw = raw_input('Give number of rows (e.g. 40) : ')
    # craw = raw_input('Give number of columns (e.g. 80) :
    # mraw = raw_input('Give size of squares (e.g. 10) : ')
    # (row, col, msz) = (int(rraw), int(craw), int(mraw))
    (row, col, msz) = (40, 80, 10)
    dly = 60  # delay for next frame
    show = GameOfLife(top, row, col, msz, dly)
    top.mainloop()
```

some reflections

The scripts `game.py` and `life.py`
follow an incremental modular design:

- the module `game.py` defines the rules
  and allows for early testing
- the GUI is in `life.py`,
  rendering separate from rules of the game

Limitation: cells are squares, dictated by the grid.

It is hard to give cells some individuality,
allowing for slight variations of the rules,
e.g.: introducing randomness.

Summary and Assignments

Read start of chapter 11 of *Python Programming*.

Assignments:

1. In an object oriented version of the module `game.py`, use the code for the function `random_grid` in the constructor of the grid.
2. In an object oriented version of the module `game.py`, use the code for the function `write` to define the string representation of the grid.
3. In an object oriented version of the module `game.py`, use the code for the functions `neighbors` and `update` to define the transition to the next state of the grid. You can name the method `update()`.
4. Modify the function `neighbors` so that the grid has the topology of a doughnut: the neighbors at the far left edge are at the far right edge and the neighbors at the lower edge are at the upper edge.

more assignments

5. Modify the initialization of the game so there is at least one glider.
6. Add an entry widget to the GUI to display the number of live cells in each stage.
7. Instead of a boolean list of lists to represent the grid, use a list of lists of integers. A square with no life holds a zero. A living cell is encoded by a positive number equal to the stage in which it was created. Describe the changes that need to be made to the scripts.
8. Change the drawing of the cells in the GUI so that newborn cells are colored in red. Surviving cells remain displayed in green.