CSCI 235, Lab Exercise 7, Iterators

Deadline: 28.10.2022 at 11PM

Solutions must be submitted into Moodle as a single text file, called **lab07.py**. Do not use any archiver.
Goal of this exercise is to make you further acquainted with Python, to use iterators, and to implement some nice things. We will implement Conway's Game of Life in Python. (See `en.wikipedia.org/wiki/Conway%27s_Game_of_Life`)

The Game of Life is based on a two dimensional grid in which cells can be either on or off. In this grid, one can make figures that evolve over time. At each point in time, the state of the next time point is computed. This results in the figures starting to move (they come to life). The next state of a cell depends on its current state, and the states of its direct neighbours. The rules are as follows: Assume that the cells are an $m \times n$ matrix S, where m is the number of rows (vertical size of the grid) and n is the number of columns (horizontal size of the grid).

- If $S[i][j]$ is on, and it has 2 or 3 neighbours that are on, it will remain on in the next state.
- If $S[i][j]$ is off, and it has three neighbours that are on, it will be turned on in the next state.
- In all other cases, $S[i][j]$ will be off in the next state.

We will create a `Life` class, print methods, and a method that computes the next state. Start your file `lab07.py` with

```
import time
from typing import List, Tuple, Generator

class Life :
   state : List[ List[ bool ]]
   m : int
   n : int
```

Tasks

1. Write a constructor

```
def __init__( self, m : int, n : int ) :
```

that initializes the class fields `state`, `m`, and `n`. It must initialize `state` with a list of length m containing lists of length n filled with `False`. The easiest way to initialize `state` is by using `append`. If you try to do it in a smarter way (for example by using `*`) you will get into trouble with the reference semantics of Python later.

In order to print the constructed state, add

```
def __repr__( self ) -> str :
   return str( self. state )
```

Example:

```
lf = lab07. Life(3,4)
lf
```

must give the following output:

```
[[False, False, False, False], [False, False, False, False],
 [False, False, False, False]]
```

2. In order to compute the next state, we will create an iterator

```
def neighbours( self, i : int, j : int ) -> Generator[
                             Tuple[ int, int ], None, None ]:
```

that iterates through the neigbours of point (i, j). This iterator must be a member function the `Life` class.

In principle, the neighbours are the points in the set

$$\{(i{-}1, j{-}1), (i{-}1, j), (i{-}1, j{+}1), (i, j{-}1), (i, j{+}1), (i{+}1, j{-}1), (i{+}1, j), (i{+}1, j{+}1)\},$$

but points that are outside of the grid should not be generated. So, `neighbours(i,j)` should only generate points (i', j') with $0 \leq i' < m$ and $0 \leq j' < n$. Examples:

```
lf = lab07. Life(4,5)
for pnt in lf. neighbours( 2, 3 ):
   print( pnt )

# generates
(1, 2)   (1, 3)   (1, 4)
(2, 2)   (2, 4)   (3, 2)
(3, 3)   (3, 4)
```

```
for pnt in lf. neighbours( 0, 3 ):
   print( pnt )

# generates
(0, 2)   (0, 4)   (1, 2)   (1, 3)   (1, 4)

for pnt in lf. neighbours( 3, 4 ):
   print( pnt )

# generates
(2, 3)   (2, 4)   (3, 3)
```

3. Next create a method

   ```
   def nextstate( self ) -> None:
   ```

   that computes the next state. Iterate through all positions (i, j) with $0 \leq i < m$ and $0 \leq j < n$. For each (i, j), count the neighbours that are on, using `neighbours(i,j)`, and determine if (i, j) will be on in the next state.

   First compute a new grid `next : List[ List[ bool ]]` and at the end, assign
   `self. state = next`. The easiest way to add elements to `next` is by using `append`.

4. Write a method

   ```
   addfigure( self, i : int, j : int, figure : List[ str ] ) -> None:
   ```

   that adds a figure, represented as a list of strings, at position (i, j). Any character that is not a dot or a space should result in the corresponding cell being on. If $i < 0$, $j < 0$, or a part of the figure would be placed outside of the grid, `addfigure` must raise a `ValueError`. Examples are given below.

5. Write a method `def __str__( self ) -> str :` that prints the grid in a nice fashion. Cells that are on should be printed as `#` and cells that are off should be printed as a dot (`.`).

6. If you made everything correctly, the following code will work (put it in file **lab07.py**):

   ```
   def start( ) :
      toad = [ ".###",
               "###." ]

      blinker = [ "###" ]
   ```

```
block = [ "..##..",
          "..##.." ]

glidergun = [ "...................................#............",
              ".................................#.#............",
              ".......................##......##............##.",
              "......................#...#....##............##.",
              "...........##........#.....#...##...............",
              "...........##........#...#.##....#.#............",
              ".....................#.....#.......#............",
              "......................#...#.....................",
              ".......................##......................." ]

lf = Life( 50, 80 )

lf. addfigure( 10,10, glidergun )
lf. addfigure( 30, 10, toad )
lf. addfigure( 40, 15, blinker )

while True:
   print( lf )
   print( "press Ctrl-C to stop" )
   lf. nextstate( )
   time. sleep( 0.25 )
```

If you want you can add more figures.