**CIS 229 – Python Programming – Programming Project**

**Conway’s Game of Life**

Overview

In this assignment, the student will be coding a script that implements John Conway’s “Game of Life” simulation. The Game of Life is a set of simulation rules that are meant to model cellular life. The game takes place on a two-dimensional, square grid for a board. Within the grid, each square is referred to as a cell and can assume one of two states, either DEAD or LIVING. The game board “evolves” from one generation to the next through a series of simple rules that are meant to model the life and death cycles of real-life entities.

When completing this assignment, the student should demonstrate mastery of the following concepts:

* File Input
* Two Dimensional Lists
* Exception handling
* Type casting

Assignment

The simulation begins by taking an initial input state from a data file. The input file will contain a series of ordered pairs that represent the cells on the board that start in the LIVING state. As the simulation progresses, each cell examines its neighbors (at most 8 neighbors per cell) and determines if a living cell should persist/die or if a dead cell should spring new life. Consider the following data file:

DATA FILE

0 0

1 1

1 2

2 0

2 1

0 1 2 3

---------------

0 \* | D | D | D

---+---+---+---

1 D | \* | \* | D NOTE: Coordinates are given as (row, column)

---+---+---+--- where rows are numbered from top -> bottom

2 \* | \* | D | D and columns are numbered from left -> right

---+---+---+---

3 D | D | D | D

Since this data is coming directly from a file, this is considered the zeroth “generation” of the simulation. As the board moves from one generation to the next, decisions about the life and death of the cells will be made based on the following rules:

Rule #1: A DEAD cell with exactly 3 neighbors will become a LIVING cell.

Rule #2: A LIVING cell with 2 or 3 LIVING neighbors will remain LIVING.

Rule #3: A LIVING cell with 1, 4, 5, 6, 7 or 8 neighbors will become DEAD.

Rule #4: A DEAD cell without exactly 3 living neighbors will remain DEAD.

In other words, we must take into consideration the number of living neighbors each cell has along with its current state to make a decision about whether it should be marked as DEAD or LIVING in the next round of the game. The given configuration would result in the following changes from one generation to the next...

0 1 2 3 0 1 2 3

--------------- ---------------

0 \* | D | D | D 0 D | \* | D | D

---+---+---+--- ---+---+---+---

1 D | \* | \* | D 🡺 1 D | D | \* | D

---+---+---+--- ---+---+---+---

2 \* | \* | D | D 2 \* | \* | \* | D

---+---+---+--- ---+---+---+---

3 D | D | D | D 3 D | D | D | D

* The cell at (0, 0) becomes DEAD as a result of Rule #3.
* The cell at (0, 1) becomes LIVING as a result of Rule #1.
* The cell at (0, 2) remains DEAD as a result of Rule #4.
* The cell at (0, 3) remains DEAD as a result of Rule #4.
* The cell at (1, 0) remains DEAD as a result of Rule #4.
* The cell at (1, 1) becomes DEAD as a result of Rule #3.
* The cell at (1, 2) remains LIVING as a result of Rule #2.
* The cell at (1, 3) remains DEAD as a result of Rule #4.
* The cell at (2, 0) remains LIVING as a result of Rule #2.
* The cell at (2, 1) remains LIVING as a result of Rule #2.
* The cell at (2, 2) becomes LIVING as a result of Rule #1.
* The cell at (2, 3) remains DEAD as a result of Rule #4.
* The cell at (3, 0) remains DEAD as a result of Rule #4.
* The cell at (3, 1) remains DEAD as a result of Rule #4.
* The cell at (3, 2) remains DEAD as a result of Rule #4.
* The cell at (3, 3) remains DEAD as a result of Rule #4.

Now that the rules of the simulation are understood, we can lay the foundation for coding a script that runs the simulation. The board entity will need to be modeled with a two-dimensional list. Each entry in the list will contain a state value that indicates if a cell is LIVING or DEAD. Consider the following example driver that would run the program:

EXAMPLE DRIVER

# Global

INPUT\_FILE = 'P03S-StartingState.dat'

EMPTY = 0 # filler marker for board generation

ROW\_NUM = 10 # Number of rows to generate

COL\_NUM = 10 # Number of columns to generate

MARK\_DEATH = ' ' # Marker for dead cells

MARK\_LIVE = '\*' # Marker for living cells

def main():

"""Driver handles main control flow for program"""

# Build the board

board = create2Dlist(ROW\_NUM, COL\_NUM)

# Set the board to all death markers

board = intializeBoard(board, MARK\_DEATH)

# Insert live markers from file

board = populateBoardFromFile(board, INPUT\_FILE, MARK\_LIVE)

# get # of generations to run from user

genNumber = getGens()

# loop through the remainder of the generations

for i in range(genNumber):

# display the results

display(board, i + 1)

# run through a generation

generationShift(board, MARK\_DEATH, MARK\_LIVE)

main()

The primary tasks the simulation needs to carry out are tucked away into functions. Many functions have already been written in assignment 8. We will now complete the games logic:

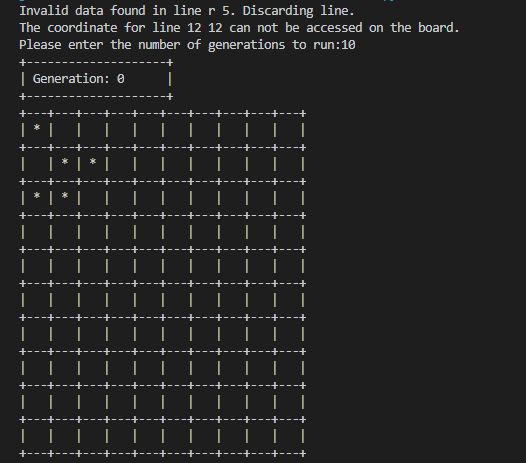
GenerationalShift(board, markDead, markAlive) – This function will take a board and apply the aforementioned rules governing the life and dead of cells. Since the board is a two-dimensional list, any changes made in this function will persist in the driver.

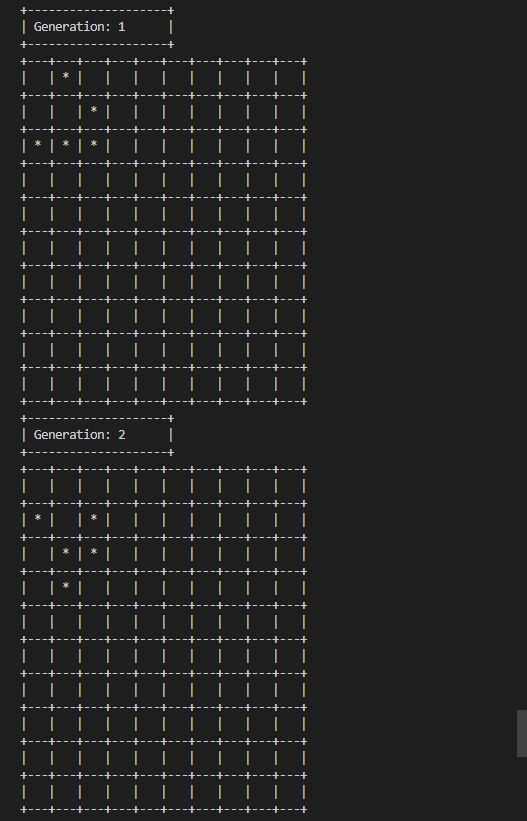
getGens() – this function will request the number of generations to run from the user. Use appropriate error handling techniques to ensure the program does not proceed until the user has entered an integer value.

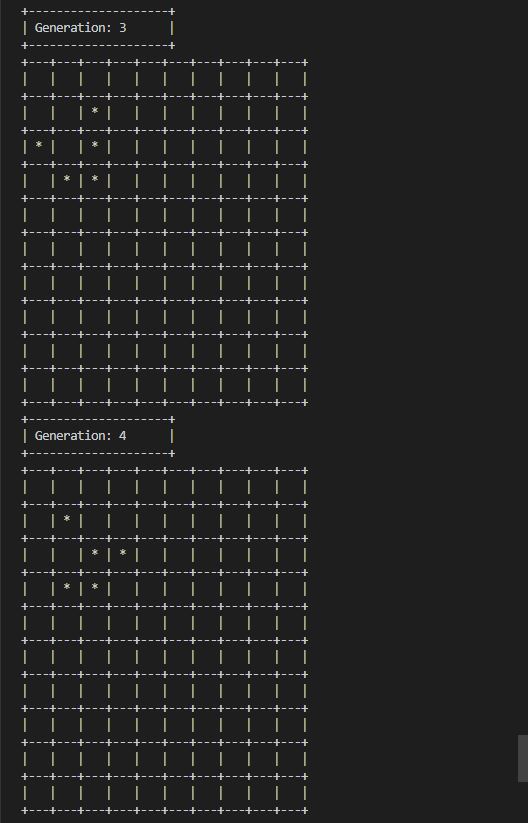
When the program begins, it will need to read the data from a file on the hard disk and determine where the mark cells as LIVING. If data is encountered that cannot be parsed as an “int” data type, all data for that particular line in the file should be ignored and a message should be displayed indicating that an error occurred. Additionally, information about the number of rows and columns should be provided in global constants declared in the program. If coordinates are encountered in the file that cannot be logically placed on the board, an error message should also be displayed.

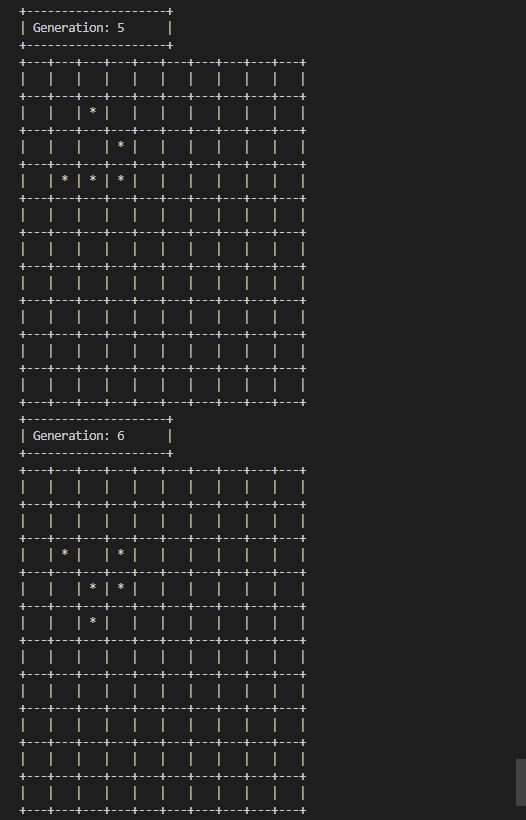
Once the board has been loaded, a prompt should ask the user for the number of generations that he/she would like to see (including the zeroth generation). After a number has been provided, the program should commence to spawn off the requested number of generations and display the board for each generation state. Tasks should be logically organized into functions that exactly model or closely imitate the ones outlined in the provided driver code. The input provided in the game rules is a special pattern that will create a glider shape that moves from the upper-left to bottom-right corner of the board. A sample data file (with errors) is provided with this assignment and should be used to test your program.

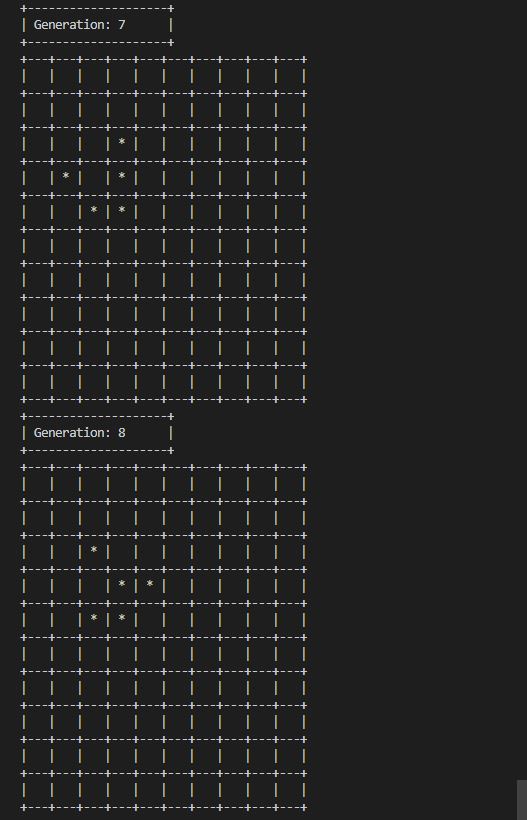
Sample Run

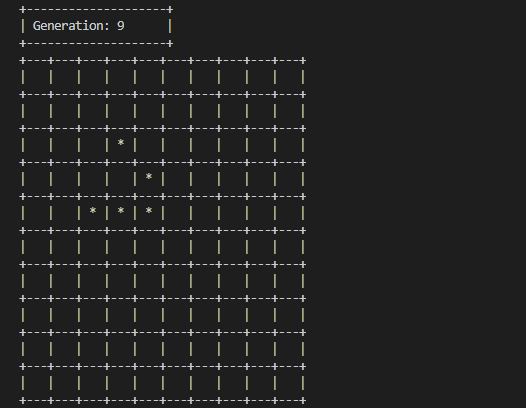












Assessment

This assignment will be assessed based on the provided grading rubric.