Assignment 4 – 2D Arrays, HOF and Sorting

Required Files

- game_of_life_template.py

Notes

- No library has been imported for you. If you need any additional standard library, you may add them youself.
- You are NOT allowed to change the input parameters to the functions (eg., number of parameters.)
- You are NOT allowed to change the name of the function.
- You may reuse the code written in the earlier sub-task for your subsequent sub-tasks.
    - You may assume that the correct implementation is given.
    - If you wish to override the standard implementation of earlier sub-tasks, please save the code for your earlier sub-tasks.

Part 1: Conway's Game-of-Life [100 marks]

The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.

-- Wikipedia

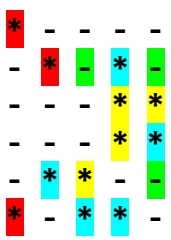
We are interested in observing the state of a Game-of-Life simulation.

The simulation begins with a rectangular society of size $m \times n$ (m rows, n columns), with any of $m \times n$ cells may be dead or alive. Our task is to monitor the evolution of the cells in this society for several generations (as determined by the user) based on the following four Game-of-Life Rules:

1. Any live cell with fewer than two live neighbours dies, as if caused by loneliness.
2. Any live cell with two or three live neighbours lives on to the next generation, as if sustained by friendship.
3. Any live cell with more than three live neighbours dies, as if caused by over-crowdedness.
4. Any dead cell with exactly three live neighbours becomes a live cell in the next generation, as if by reproduction.

Note that cells can have up to 8 neighbours, where a neighbour is a cell that is directly adjacent or diagonal to the cell.

For example, given the following demography of a 6 × 5 society (here a dead cell is represented by `'-'`, and a live cell is represented by `'*'` respectively):



After one evolution, the new generation of the society will look like:

```
- - - - -
- - * * *
- - - - -
- - - - *
- * - - *
- - * * -
```

To help you observe the rules in action, cells that meet any of the four rules have been colour-coded based on the colour of the rules shown above.

After a total of five evolutions, the society looks like:

```
- - - - -
- - - - -
- - - - -
- - - * -
- - * * -
- - * * -
```

The society may also go through migration based on the population census. The migration is essentially based on the number of live cells in each row of the society found after the census. Migration works by sorting the rows in descending order of the census. If rows have the same census value, their relative ordering is maintained (ie., the sorting is "stable").

For example, given the following society:

Society	Census	Rank
- * - * -	2	3
* - - * *	3	2
- * * * *	4	1
* - - - -	1	6
- * * - -	2	4
- - * - *	2	5

We sort the census results in descending order (as shown by the rank). The resulting society after migration would be:

Society	Census	Rank
- * * * *	4	1
* - - * *	3	2
- * - * -	2	3
- * * - -	2	4
- - * - *	2	5
* - - - -	1	6

Your Task

Following is a general overview of how your code would be executed:

```
def main(m=6,n=5, num_of_evolutions = 10, gol = gol):
```

```
    # (1) Build a random society of m by n.
    society = build_society(m, n)
    print("Initial Society:")
    print_society(society)

    # (2) Evolve the society for num_of_evolution iterations.
    result, num_evolutions_to_stability = evolve(society,num_of_evolutions, gol)
    print("After {} evolutions, resulting society:".format(num_of_evolutions_to_stability))
    print_society(result)

    # (3) Perform migration on the society.
    after_migration = migrate(result)
    print("Resulting society after migration:")
    print_society(after_migration)

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

The code snippet:

1) Builds a random *m* × *n* society. That is, it creates a list of *m* lists, each of size *n*, which are randomly populated with `'-'` or `'*'`. (In this assignment, however, you will also be required to build a society based on some model – thus not randomly.)
2) Evolves the society for a set number of iterations, using specific game-of-life rules (the argument `gol` is defined in Task 1C.) After evolution, show the resulting society, and the minimum number of iterations to arrive at that result (stability).
3) Performs a migration on the society.

Example Run

```
Initial Society:
- * * - *
* - * - -
- * - - -
- - * * *
- * * - *
- - - - -
After 10 evolutions, resulting society:
- - - - -
* - - * -
* - * - -
* * * - -
- - - - -
- - - - -
Number of evolutions to achieve stability: 10
Resulting society after migration:
* * * - -
* - - * -
* - * - -
- - - - -
- - - - -
- - - - -
```

Note that if the society is randomly built, then it is not necessary for your program to produce the exact same societies.
Also, the function `print_society(society)` is already given to you. It's good to study this piece of code to get an understanding of the actual encoding of a society.

Task 1A: Building Randomly a Society [10 marks]

In this task, we are going to define the function `build_society(m, n)`, which builds an m × n society randomly populated with live and dead cells.

Assumptions

- m and n ≥ 1
- Output must be a list of lists

Note: As long as the result of your function returns an m × n society randomly populated by dead and live cells, your solution will be accepted. There is also no need to set the random seed for computation.

Example Run

```
>>> build_society(1, 1)
[['-']]
>>> build_society(3, 2)
[['-', '-'], ['-', '-'], ['*', '*']]
>>> print_society(build_society(3, 5))
* * - * -
* - - - *
* * - - -
>>> print_society(build_society(3, 5))
* - - * *
* - - - -
- - * - -
>>> print_society(build_society(3, 5))
- * * * -
- - - - *
* * * * -
```

Task 1B: Modeling a Society [15 Marks]

As we have observed from Task 1A, we have not much control over the life or death of a cell in the society built. In this task, we instead would like to take control over the life or death of every cell when we build a society. For instance, the society built in the last sample run of Task 1A is replicated below:

```
>>> print_society(build_society(3, 5))
- * * * -
- - - - *
* * * * -
```

Here, we want to model a society based on this resultant society. We shall take in this resultant society, which consists three strings of characters, and convert it into proper internal structure of a society: List of lists of characters of `'-'` and `'*'`. To be precise, we copy the printed society and paste it as the argument to our `model_society` function.

Sample runs (outputs are highlighted in blue for clarity):

```
>>> model_society('''
- * * * -
- - - - *
* * * * -
''')
[['-', '*', '*', '*', '-'], ['-', '-', '-', '-', '*'], ['*', '*', '*', '*', '-']]

>>> print_society(model_society('''
- * * * -
- - - - *
* * * * -
'''))
- * * * -

- - - - *

* * * * -
```

Note that in the above, outputs from the function call are highlighted in blue for clarity sake. There is only one input to model_society, which is coloured in green, and it is a **multi-line string**. A multi-line string can contain "end-of-line" characters, and thus appears on the display as having multiple lines. To be expressible a multi-line string, it has to be enclosed by a pair of **three quotation marks**: either ''' or """.

Define the function `model_society(txt)` that takes in a multi-line string and turns it into a society of certain *n* x *m* cells. (*Hint:* You may wish to utilize some of the well-known string methods/functions to complete your task. For instance, methods such as `split`, `replace`, etc. Refer to Python manual to determine what these methods do.)

Task 1C: Coding the Game-Of-Life Rules [15 Marks]

Our next task is to turn the set of Game-of-Life rules into a function. Specifically, we notice that whether a cell will be destined to live or die depends on two factors: (1) its current liveness: either it is alive or dead at the moment; and (2) the number of live neighbors surrounding it.

Write a function **`gol(cell_value, live_neigh)`** that takes in (1) a Boolean value representing live (by True) and dead (by False); and (2) an integer between 0 and 8 (both inclusive) capturing the number of live cells in its (maximum eight) neigbourhood. The function returns True when the cell will be alive in the next evolution, and False otherwise. Your function should define correctly the game-of-life rules, as replicated below:

1. Any live cell with fewer than two live neighbours dies, as if caused by loneliness.
2. Any live cell with two or three live neighbours lives on to the next generation, as if sustained by friendship.
3. Any live cell with more than three live neighbours dies, as if caused by over-crowdedness.
4. Any dead cell with exactly three live neighbours becomes a live cell in the next generation, as if by reproduction.

Sample runs:

```
>>> gol(True, 3)    # Current cell is alive, three live neighbor
True                # The cell is destined to die in the next evolution
>>> gol(False, 3)   # Current cell is dead, three live neighbor
True                # The cell is destined to live in the next evolution
```

Notes about this Task:

1. You should never hard-code the solution.
2. If we can code the current game-of-life rules into a function, we can also change the rules by providing a variant of the function. This creates flexibility in allowing us to set different rules for Game of Life. Effectively, our Game-of-Life program becomes a **Generalized Game of Life.**

Task 1D: Tabulating Game-of-Life Rules [15 Marks]

As we begin to explore different ways of defining game-of-life rules, we want to be certain about the effect of our defined rules. A way to ascertain the effect is to display the results of the rules given the current cell value and the number of live neighbours.

Write a function **`tabulate(rules)`** that takes a function encoding some game-of-life rules, and tabulating all its effects. For instance, following is the result of tabulating the current game-of-life rules which we have defined in Task 1C:

Sample runs:

```
>>> tabulate(gol)
[('live', ['dead', 'dead', 'live', 'live', 'dead', 'dead', 'dead', 'dead', 'dead']),
 ('dead', ['dead', 'dead', 'dead', 'live', 'dead', 'dead', 'dead', 'dead', 'dead'])]
```

The returned value is a list of two elements, each of which is a tuple. It is a representation of the following results:

Current Cell Value	Number of live neighbours								
	0	1	2	3	4	5	6	7	8
Live	Dead	Dead	Live	Live	Dead	Dead	Dead	Dead	Dead
Dead	Dead	Dead	Dead	Live	Dead	Dead	Dead	Dead	Dead

The first component is the value of the current cell, and the second component is a list of nine values, corresponding to the result of applying `gol` with current cell value and the number of live neighbours.

Following sample runs tabulates yet a different game-of-life rules – the rules do exactly the opposite of what the original game-of-life rules did.

```
>>> >>> tabulate(lambda cv,neigh: not gol(cv,neigh))
[('live', ['live', 'live', 'dead', 'dead', 'live', 'live', 'live', 'live', 'live']),
 ('dead', ['live', 'live', 'live', 'dead', 'live', 'live', 'live', 'live', 'live'])]
```

Task 1E: The Destiny of a Cell in the Society [15 Marks]

Before we begin evolving the society, we must be able to determine the liveness of a cell after a single iteration of evolution, based on Game-of-Life rules, or any other eligible rules.

Define a function `destiny(society, coordinates, gol)` that reads a society, the coordinates of a cell in the society and a function representing a set of game-of-life rules. The cell coordinate is a tuple in the format of `(row_num, col_num)`, whose destiny is to be determined by calling this function. The function returns the liveness of the cell, where `'-'` is a dead cell and `'*'` is a live cell.

Assumptions

- For a society of size $m \times n$,
  - $0 \leq$ `coordinates[0]` $<$ `m`
  - $0 \leq$ `coordinates[1]` $<$ `n`
- `m` and `n` ≥ 1
- `gol` is a function from a tuple of cell-value and the number of live neighbours to a boolean value indicating the cell will be alive or dead in the next evolution.

Example Run

```
>>> society = build_society(3,2)
>>> print_society(society)
* *
* -
- -
>>> destiny(society, (0, 0), gol)
'*'
>>> destiny(society, (2, 1), gol)
'-'
```

Note on Coursemology

The input society ***must not be amended***. Also, there is no need to submit the code for game-of-life rules; this will be provided in Coursemology.

Task 1F: Evolving the Society [15 Marks]

Now, we can proceed to evolve a society.

Define a function `evolve(society, n, gol)` which returns (1) the resulting society after `n` generations of evolutions, based on a function defining the Game-of-Life rules; (2) the minimum number of iterations required to arrive at that society, ie., its stability.

A society is stable if one more evolution based on some Game-of-Life rules does not produce a different society from the last one. For example, say you attempt to evolve a society for 10 iterations. A new society is generated during the 5th iteration (evolving 5 times). However, at the 6th iteration, the society produced is not different from the existing one (the society at the 6th iteration is the same as the society at the 5th iteration). In this case, the minimum number of iterations required to arrive at the resulting society, or the number of iterations required to achieve stability, is **5**. Otherwise, in the case where the society continues to evolve for all 10 iterations, then the minimum number of iterations to arrive at that resulting society is hence 10.

Assumptions

- The society is at least of size 1 × 1
- `n` ≥ 1
- `gol` is a function from a tuple of cell-value and the number of live neighbours to a boolean value indicating the cell will be alive or dead in the next evolution.

Note on Coursemology

The input society ***must not be amended***. Also, a correct implementation of `destiny(society, coordinates, gol)` is provided to you on Coursemology, you can simply call it, and ***you're not allowed to re-define it***.

Example Run

```
>>> evolve([['-']], 1000, gol)
([['-']], 0)
>>> society = build_society(2, 3)
>>> print_society(society)
- - *
* * -
>>> res = evolve(society, 5, gol)
>>> print_society(society)
- - *
* * -
>>> print_society(res[0])
- - -
- - -
```

```
>>> res[1]
2
```

Task 1G: Society Migration [15 Marks]

For this final sub-task, we are going to define the `migrate(society)` function, which performs migration on an input society based on the population census, i.e. the number of live cells in each row. The rows in the input society are sorted by the function in descending order of their census value, with the sorted society produced as the output. Note that for rows with the same census value, their relative order must be maintained.

Assumptions

- The society is at least of size 1 × 1

Note

The input society ***must not be amended***.

Example Run

```
>>> society = build_society(6, 5)
>>> print_society(society)
_ * _ _ *
_ * * _ *
* * _ * _
_ _ * _ *
* _ _ _ _
_ * _ _ _
>>> res = migrate(society)
>>> print_society(society)
_ * _ _ *
_ * * _ *
* * _ * _
_ _ * _ *
* _ _ _ _
_ * _ _ _
>>> print_society(res)
_ * * _ *
* * _ * _
_ * _ _ *
_ _ * _ *
* _ _ _ _
_ * _ _ _
```