Version 1.00

Assignment 2
Cell Simulator

1. Submission Guidelines

Deadline:	11:59 PM on Thursday 16 December 2021
Submission procedure:	Submit only one file labelled `cellsim.py` through blackboard (via TurnItIn)
Version requirement:	Your code must run using **Python 3.10.0 on a PC**
Allowable import modules:	`os`, `time`, `random`, `copy`. No other modules are allowed for this assignment.

2. Overview

Write Python code in `cellsim.py`, which simulates the growth and death of a population of cells. Your code should be importable into another Python script (e.g., `script.py`). However, your code, `cellsim.py`, can import only the modules that are listed above (`os`, `time`, `random`, `copy`)

Write three classes: `Tissue`, `Cell`, and `Cancer`. `Tissue` is represented as a two-dimensional grid with a specified number of rows and columns. Each location on the grid should represent a cell of type `Cell`, `Cancer`, or any other cell type that we'd may define. Each cell is either in a state of being alive or dead. Healthy cells of type `Cell` that are alive are represented with the character '`O`' while cancerous cells of type `Cancer` that are alive are represented with the character '`X`'. Dead cells are represented by '`.`'.

Your code should be written in an object-oriented manner. When marking your code, we will write our own cell types and evaluate whether we can use them with your `Tissue` class type. It should still work.

All example code listed here assumes that the following imports are listed at the top of a python script (e.g., `script.py`):

```
import cellsim
import os
import time
```

3. Class `Tissue`

The class `Tissue` should represent the space where cells grow and die.

3.1. Attribute variables

Objects of type `Tissue` should have the following variable attributes:

`matrix`

A 2-dimensional array of type `list`. Each element is either of type `Cell`, `Cancer`, or another cell type.

`rows`

A number of type `int` representing the number of rows in `matrix`.

`cols`

A number of type `int` representing the number of columns in `matrix`.

`CellType`

The type of cell to use for each element of the `matrix`. Can be `Cell`, `Cancer`, or another cell type.

3.2. Methods

Objects of type `Tissue` should have the following methods defined:

`__init__()`

Initialises an instance of type `Tissue`. The argument into this method should be rows, columns, and the cell type. It should initialise the four attribute variables by setting `rows`, `cols`, and `CellType`; and generating `matrix`.

The rows argument should be a number of type `int` representing the number of rows to generate for `matrix`. The default value of the rows argument should be 1. The columns argument should be a number of type `int` representing the number of columns to generate for `matrix`. The default value of the columns argument should be 1. The cell type argument should be the name of the class that you'd like to use to generate `matrix`. The default cell type should be `Cell`.

The `matrix` generated should have rows and columns specified by `rows` and `cols`, respectively. Each element within the 2-dimensional `matrix` should be an object of type `CellType`. Initialise the cell to being not alive.

The example code...

```
tissue = cellsim.Tissue()
print(tissue.matrix)
print(tissue.rows)
print(tissue.cols)
print(tissue.CellType)
```

should produce the following console output:

```
[[<cellsim.Cell object at 0x1104ebc40>]]
1
1
<class 'cellsim.Cell'>
```

Note: the memory address specified by `0x...` will be different than what's listed above.

The example code...

```
tissue = cellsim.Tissue(6,6,cellsim.Cell)
print(tissue.matrix)
print(tissue.rows)
print(tissue.cols)
print(tissue.CellType)
```

should produce the following console output:

```
[[<cellsim.Cell object at 0x1104eb370>, <cellsim.Cell object at
0x1104eba00>, <cellsim.Cell object at 0x1104eba60>, <cellsim.Cell object at
0x1104ebb20>, <cellsim.Cell object at 0x1104ebb80>, <cellsim.Cell object at
0x1104ebdc0>], [<cellsim.Cell object at 0x1104eb940>, <cellsim.Cell object
at 0x1104ebeb0>, <cellsim.Cell object at 0x1104ebf10>, <cellsim.Cell object
at 0x1104ebf70>, <cellsim.Cell object at 0x1104eb6d0>, <cellsim.Cell object
at 0x1104eb640>], [<cellsim.Cell object at 0x1104ebe20>, <cellsim.Cell
object at 0x1104eb460>, <cellsim.Cell object at 0x1104eae00>, <cellsim.Cell
object at 0x1104eada0>, <cellsim.Cell object at 0x1104ead40>, <cellsim.Cell
object at 0x1104ea200>], [<cellsim.Cell object at 0x1104eb5e0>,
<cellsim.Cell object at 0x1104eaa70>, <cellsim.Cell object at 0x1104eac80>,
<cellsim.Cell object at 0x1104eb040>, <cellsim.Cell object at 0x1104eafe0>,
<cellsim.Cell object at 0x1104eaf80>], [<cellsim.Cell object at
0x1104e9cf0>, <cellsim.Cell object at 0x1104eaec0>, <cellsim.Cell object at
0x1104eae60>, <cellsim.Cell object at 0x1104ea1d0>, <cellsim.Cell object at
0x1104ea170>, <cellsim.Cell object at 0x1104ea110>], [<cellsim.Cell object
at 0x1104eaf20>, <cellsim.Cell object at 0x1104e96c0>, <cellsim.Cell object
at 0x1104e9660>, <cellsim.Cell object at 0x1104e9540>, <cellsim.Cell object
at 0x1104e9c00>, <cellsim.Cell object at 0x1104eab00>]]
6
6
<class 'cellsim.Cell'>
```

Note: the memory addresses specified by `0x...` will be different than what's listed above.

`__str__()`

Define this method so that the `matrix` attribute variable is displayed in the appropriate format. This method should return a variable of type `str`.

The example code...

```
tissue = cellsim.Tissue(6,6,cellsim.Cell)
print(tissue)
```

should produce the following console output:

```
......
......
......
......
......
......
```

`__getitem__()`

Define this method so that the user can extract specific elements from the attribute variable `matrix`.

`__setitem__()`

Define this method so that the user can set specific elements from the attribute variable `matrix`.

`seed_from_matrix()`

Overwrite the four attribute variables using a single argument.

The argument should be a two-dimensional array of type `list`. Each element of the array should be of a cell type, being `Cell`, `Cancer`, or another cell type. The attribute variables of the `Tissue` object should be updated based on the parameters of this array argument.

The example code...

```
tissue = cellsim.Tissue(10,40,cellsim.Cell)
test_matrix = list()
for i in range(10):
    test_matrix.append([])
    for j in range(40):
        test_matrix[i].append(cellsim.Cell(False))
test_matrix[5][5] = cellsim.Cell(True)
test_matrix[5][6] = cellsim.Cell(True)
test_matrix[5][7] = cellsim.Cell(True)
tissue.seed_from_matrix(test_matrix)
print(tissue)
print(test_matrix[5][6])
```

should produce the following console output:

```
........................................
........................................
........................................
........................................
........................................
.....OOO................................
........................................
........................................
........................................
........................................

O
```

`seed_from_file()`

Overwrite the four attribute variables using parameters extracted from a file and the cell type defined in the argument. The arguments into this method should be a file name and a cell type.

The file name should be of type `str` and should represent the file name and its relative path. The cell type argument should be the name of the class that you'd like to use for each element of `matrix`, being either `Cell`, `Cancer`, or any other cell type. The default cell type should be `Cell`.

The file should be a `.txt` file representing a board.

Sample text file, `test_tissue_01.txt`:

```
........................................
........................................
........................................
........................................
........................................
.....OOO........................OOO.....
........................................
........................................
........................................
........................................
```

The example code...

```
tissue = cellsim.Tissue(10,40,cellsim.Cell)
tissue.seed_from_file('test_tissue_01.txt', cellsim.Cell)
print(tissue)
```

should produce the following console output:

```
........................................
........................................
........................................
........................................
........................................
.....OOO........................OOO.....
........................................
........................................
........................................
........................................
```

`seed_random()`

Create a randomly distributed array of cells, which have an approximate confluency. The arguments to this method should be the confluency and the cell type. Use the `random` module to implement this method.

The confluency argument should be of type `float` and should range between `0.0` and `1.0`. This will set the probability that each cell in the array is alive or dead. The cell type argument should be the name of the class that you'd like to use for each element of `matrix`, either `Cell`, `Cancer`, or any other cell type. The default cell type should be `Cell`.

The example code...

```
tissue = cellsim.Tissue(10,40)
tissue.seed_random(0.5,cellsim.Cell)
print(tissue)
```

should produce the following console output:

```
O..OOO..OO.O.OO.O..OO..OOOOOOOOOOOOO...O
..OO...O..O.....OO.OO...OO...OOOO...OO..
...O...OO..O.......OO..O.OOO...OOOO.....
O..OOO.OO.OOO.OO.OOOOOOOO..O..OOO.O...O.
..O.O.O.OO..O.O..OOO.O...O....OOO.OOO.OO
O..OOO..OO...O..O.O.OO.O...OOO....O.OOO.
..O..O.OOO...OO..O.OO.OO.OOOO.O.O..OOOO.
OO.O.OOO...OO.....O.OOOO.O...OOOO..OO...
OOOO.OO.O.O.O..O.O...OO..O.O.OO......OOO
..OO.....OOOOO.O.OOOOO.OO.OO...O..OOOOO.
```

Note: the distribution of alive and dead cells displayed here does not need to exactly match the location of alive and dead cells in your implementation. However, the percentage of living and dead cells should approximately match.

`next_state()`

Based on the current attribute variable `matrix` and its cell types, create an updated version of `matrix`. The change from the current `matrix` to the new `matrix` must be determined by a set of rules defined by each cell and its surrounding cells. So, you must access the `update_cell()` method for each cell element, being either `Cell`, `Cancer`, or another cell type.

The example code...

```
# this code should be run from your terminal or command prompt
# from terminal, move your current working directory to be where
# this script.py and cellsim.py are located
# then run "python3 script.py", where script.py contains the following...
tissue = cellsim.Tissue(10,40)
tissue.seed_random(0.5,cellsim.Cancer)
print(tissue)
for i in range(0,100):
    os.system('clear')  #will be os.system('cls')
    tissue.next_state()
    print(tissue)
    time.sleep(0.1)
```

Should produce the following console output at time step 0:

```
time step: 0
O..OOO..OO.O.OO.O..OO..OOOOOOOOOOOOO...O
..OO...O..O.....OO.OO...OO...OOOO...OO..
...O...OO..O.......OO..O.OOO...OOOO.....
O..OOO.OO.OOO.OO.OOOOOOOO..O..OOO.O...O.
..O.O.O.OO..O.O..OOO.O...O....OOO.OOO.OO
O..OOO..OO...O..O.O.OO.O...OOO....O.OOO.
..O..O.OOO...OO..O.OO.OO.OOOO.O.O..OOOO.
OO.O.OOO...OO.....O.OOOO.O...OOOO..OO...
OOOO.OO.O.O.O..O.O...OO..O.O.OO......OOO
..OO.....OOOOO.O.OOOOO.OO.OO...O..OOOOO.
```

and the following console output at time step 1:

```
time step: 1
..OOO...OOO....OOOOOO..O..OOO....OOOO...
..O...OO..OOO..OOO...O..............O...
............O..O...........OOO....OO.O..
..O..O....O.O.OOOO.....O...O.........OOO
.OO...O.....O.O...........OO....O.O.O..O
.OO.......O.O..OO......O.O......O.O.....
O.O......OO..OO..O.......O......OOO...O.
O..O......OOO.O.OOO......O......O..O...O
O....O..O.....O..O.......O.O.O..O.O....O
...OO....OO.OOO..OOOOO.OOOOOO.O....OO..O
```

and the following console output at time step 99:

```
time step: 99
..........................OO............
..........................OO............
........................................
.......O.....O..........................
.......O.....O..........................
.......OO...OO..........................
........................................
...OOO..OO.OO..OOO......OO..............
.....O.O.O.O.O.O........OO..............
```

```
.......OO...OO.........................
```

4. Class `Cell`

The class `Cell` should represent healthy cells. A cell can be alive or dead, should be printable after overloading the `__str__()` method, defined as being alive or not with a `is_alive()` method, and should determine whether it should be alive or dead in the next step as defined by its current `is_alive()` state and its surrounding 8 cells states.

4.1. Attribute variables

Objects of type `Cell` should have the following variable attributes:

`alive`

A boolean state of the cell being either alive (`True`) or dead (`False`).

4.2. Methods

Objects of type `Cell` should have the following methods defined:

`__init__()`

Initialise instances of type `Cell`.

The argument should be a boolean type, being either `True` or `False`. The argument should be assigned to the attribute variable, `alive`. The default argument should be `False`.

`__str__()`

This should return a character of type `str`, being 'O' if the attribute `alive` is set to `True` and `'.'` if alive is set to `False`.

`is_alive()`

This should return the attribute `alive`, being either `True` or `False`.

`update_cell()`

Determine the next state of the cell given its current surroundings.

The argument should be a 3 by 3 array containing elements of type `Cell`, `Cancer`, or another cell type. Ignore the centre element.

Death. If a cell is alive, it will die under the following circumstances:

- Overpopulation: If the cell has four or more alive neighbours, it dies.
- Loneliness: If the cell has one or fewer alive neighbours, it dies.

Birth. If a cell is dead, it will come to life if it has exactly three alive neighbours.

Stasis. In all other cases, the cell state does not change.

5. Class `Cancer`

The class `Cancer` should represent cancer cells. It should be derived from class `Cell`. A cancer cell can be alive or dead, should be printable by overloading the `__str__()` method, be defined as being alive or not with the `is_alive()` method, and should determine whether it should be alive or dead in the next step as defined by its current `is_alive()` state and its surrounding 8 cells states.

5.1. Attribute variables

Objects of type `Cancer` should have the following variable attributes:

`alive`

A boolean state of the cell being either alive (`True`) or dead (`False`).

5.2. Methods

Objects of type `Cancer` should have the following methods defined:

`__init__()`

Initialise instances of type `Cancer`.

The argument should be a boolean type, being either `True` or `False`. The argument should be assigned to the attribute variable, `alive`. The default argument should be `False`.

`__str__()`

This should return a character of type str, being 'X' if the attribute `alive` is set to `True` and `'.'` if alive is set to `False`.

`__is_alive()`

This should return the attribute `alive`, being either `True` or `False`.

`update_cell()`

Determine the next state of the cell given its current surroundings. **Compared to the same method name for `Cell`, this function has a different rule for overpopulation.**

The argument should be a 3 by 3 array containing elements of type `Cell`, `Cancer`, or another cell type. Ignore the centre element.

Death. If a cell is alive, it will die under the following circumstances:

- Overpopulation: If the cell has **five** or more alive neighbours, it dies.
- Loneliness: If the cell has one or fewer alive neighbours, it dies.

Birth. If a cell is dead, it will come to life if it has exactly three alive neighbours.

Stasis. In all other cases, the cell state does not change.

6. Coding rules

Do not declare any variables in the global space of `cellsim.py`. The only global space definitions should be class `Tissue`, class `Cell`, and class `Cancer`. Modules may be imported at the global scope, but are restricted to `os`, `time`, `random`, `copy`. Your code should be written in the spirit of object-oriented programming.

7. Marking criteria

We will mark your submitted `cellsim.py` code according to the following categories:

(1) Implementation and evidence of coding knowledge (majority of your marks)

(2) Coding efficiency

(3) Coding style and commenting

We will use `import cellsim` near the top of our script to test your code. We will run several test conditions against each of your classes and methods. We will investigate what was assigned in your attribute variables. We will test far more test cases than the examples we have included in this assignment sheet.