CPSC 231 Assignment 4

Due Date: Friday, November 29, 2019, at 11:59 PM

Weight: 8%

Collaboration

Discussing the assignment requirements with others is a reasonable thing to do, and an excellent way to learn. However, the work you hand-in must ultimately be your work. This is essential for you to benefit from the learning experience, and for the instructors and TAs to grade you fairly. Handing in work that is not your original work, but is represented as such, is plagiarism and academic misconduct. Penalties for academic misconduct are outlined in the university calendar.

Here are some tips to avoid plagiarism in your programming assignments.

1. Cite all sources of code that you hand-in that are not your original work. You can put the citation into comments in your program. For example, if you find and use code found on a web site, include a comment that says, for example:
   ```
   # the following code is from
   https://www.quackit.com/python/tutorial/python_hello_world.cfm.
   ```
   Use the complete URL so that the marker can check the source.
2. Citing sources avoids accusations of plagiarism and penalties for academic misconduct. However, you may still get a low grade if you submit code that is not primarily developed by yourself.
3. Discuss and share ideas with other programmers as much as you like, but make sure that when you write your code that it is your own. A good rule of thumb is to wait 20 minutes after talking with somebody before writing your code. If you exchange code with another student, write code while discussing it with a fellow student, or copy code from another person's console, then this code is not yours.
4. **Collaborative coding is strictly prohibited.** Your assignment submission must be strictly your code. Discussing anything beyond assignment requirements and ideas is a strictly forbidden form of collaboration. This includes sharing code, discussing code itself, or modelling code after another student's algorithm. **You can not use (even with citation) another student's code.**
5. We will be looking for plagiarism in all code submissions, possibly using automated software designed for the task. For example, see Measures of Software Similarity (MOSS - https://theory.stanford.edu/~aiken/moss/).

Remember, if you are having trouble with an assignment, it is always better to go to your TA and/or instructor to get help than it is to plagiarize.

Late Penalty:

Late assignments will not be accepted.

Submission Instructions:

Your ***Board.py* file** must be submitted electronically. **Use the Assignment 4 dropbox in D2L** for the electronic submission. You can submit multiple times over the top of a previous submission. Do not wait until the last minute to attempt to submit. You are responsible if you attempt this and time runs out.

Description

The game of X's and O's, Tic-Tac-Toe, or Noughts and Crosses has many different names. Two players are given a square array of size 3 and take turns entering their symbol into the grid. The first one to 3 in a row (across, down, or diagonal) is the winner. This game is solved; meaning there is a known strategy such that a 'perfect' player can never lose the game. Two 'perfect' opposing players will always play to a draw in the game, as well.

We will be implementing a flexible version of the game. The user will be able to play the base game on a 3 by 3 grid but will also have the option to play with row and column combinations selected from the sizes of 3, 4, or 5. For example, boards can be 3 by 5 or even 5 by 5 in size.

You will be given two modules: *tictactoe.py* and *Board.py*. *tictactoe.py* module handles the management of the game state, input from the user, drawing the game, and an AI system to play against. Your job is to complete the **Board class** (i.e. *Board.py*). There are ten different methods required for this class, along with an __init__ constructor, all of which are described in the following sections. Your job will be to complete the implementation of ***Board.py*** by implementing each of its methods to specifications indicated in the assignment. At the same time, you will also be expected to comment and document this class file.

Note, you must follow the implementation instructions exactly and must not change the existing portions of the code. You should not rename the provided files or any of their methods. If your method has a different name, takes a different number of parameters, or returns a different value than expected, then my code will not be able to call it successfully, and the game will not work. In this case, your submission will not receive a grade higher than an F.

If you break your program by doing the bonus, you will lose marks. If you do the bonus, please submit a *Board.py* file for the regular assignment and then a *tictactoebonus.py/BoardBonus.py* for the Bonus part of the assignment.

Usage:

```
>python tictactoe.py <rows> <cols> <difficulty> <piece> <-h or -a>
```

<rows> is the number of rows in the board, should be 3, 4, or 5

<cols> is the number of columns in the board, should be 3, 4, or 5

<difficulty> is the computer opponent's difficulty. It can be either 0 for Random moves, 1 for Immediate wins, 2 for the opponent looking ahead 1 move, and 3 for look-ahead 2 moves. (4 for Lookahead to end of the game for 3x3 boards).

<piece> is either X or O

<-h> is an optional flag that enables your program's hint feature, which tells the human player where to play next.

<-a> is an optional flag that enables an advanced hint feature (instead of -h), which indicates to the human player the location of the best possible move. This feature is already programmed for you for 3x3 boards.

Part 1: Creating the Board

Use a two-dimensional Python list to represent the game board. Each element in the list will be an integer that indicates what type of game piece currently resides at that location. An empty board location is represented by an integer 0, which is already defined in the Board class as the constant EMPTY. The X piece is represented in the code by the constant X with the value 1. The O piece is represented in the code by the constant O with the value 2.

The following example game state would be represented with the example Python list that follows.

X	O	X
O	O	X
	X	

```
[ [1,2,1],
  [2,2,1],
  [0,1,0] ]
```

Create the constructor (__init__)
The constructor will take 2 integer parameters (of course remember the first parameter of self):

1. the number of **rows** in the board (default = 3)
2. the number of **columns** in the board (default = 3)

Your constructor must create a two-dimensional list with the indicated number of rows and columns. Every value in the two-dimensional list must be filled with an EMPTY constant.

Run the *tictactoe.py* game after implementing this constructor. My automated tests, inside *tictactoe.py*, will give you feedback on whether or not your constructor works. Do not proceed to Part 2 until this constructor passes all of my tests. If your program has a syntax error or logic error, your program may crash during the tests. My implementation of this constructor is about 6 lines of code without any comments or blank lines.

Remember to add comments before this constructor (and any methods you implement) that describe what the constructor does and describes the parameters and the return value. You should also add in-line comments in this and any future method that explains their core functionality.

Create rows(self) and **cols(self)** methods that return the number of rows in the board and the number of columns, respectively.

Part 2: Checking If a Game Board Location is Open and Assigning a Game Piece

We cannot play with the game yet. To do so, we need to be able to play a piece on the board. To do this, we need to complete two more simple methods. The first involves checking what is in a board location. The second will involve assigning a piece to a location on the board.

Create a method named *canPlay* (notice the use of a lowercase c and an uppercase P).

The method will take 2 parameters:

1. a number for a specific **row** to look in
2. a number for a specific **column** to look in

Your method must return a Boolean value (in Python, this is a bool type, which is either True or False). If the location on the board at the given row and column is EMPTY (i.e. 0), then return True, otherwise, return False. Your method body is likely to be anywhere between 1 and 4 lines of code without comments.

Create a method named play (notice the use of a lowercase p).

The method will take 3 parameters:

1. a number for a specific **row** to look in
2. a number for a specific **column** to look in
3. the **piece** to play in the indicated location

This method will not return anything. It will only put the indicated piece into the board at the indicated row and column. You do not have to worry about checking if there is anything at this location already. Overwrite any existing value in this location, even if it is an X or O. (There should not be if the game is running correctly!). Your method body should only take a single line to complete.

Run the *tictactoe.py* game after implementing these methods. My automated tests will give you feedback on whether or not you have these methods working. Do not proceed to Part 3 until these methods pass all of the tests.

Part 3: Checking if the board is full

You should be able to run and play the game at this point. However, you may notice that the game does not know to stop when the game board is full and does not know how to identify winners. You may also notice that your code is failing the remaining tests. We will deal with these situations in the next parts of the assignment. In this part, we will deal with determining if the game board is full.

Complete the method named *full* (notice the use of a lowercase f).

The method will take no parameters.

This method is responsible for determining if the game board has any empty locations remaining. Your method will return a Boolean value. This value will be True if there are no EMPTY locations in the board, or False if there is at least one EMPTY location remaining in the board.

You will have to use a nested loop to loop through all the row and column combinations on the game board. If you check all locations and none were EMPTY, then the method can return True. However, if at any point during the search you encounter a non-EMPTY location, then the method can return False.

My implementation of this method is 5 lines of code. Do not proceed to Part 4 until this method passes all of my tests.

Part 4: Checking if a player has a win in a specific row or column

The game should now run and terminate once the board is full, but still cannot identify who has won. To perform this check, we will implement a couple of helper methods to assist another method in a future part of the assignment. We will start with two methods, one to check if a player has a win in a specific row, and a second to check if a player has a win in a specific column.

Create a method named *winInRow* (notice the use of a lowercase w. but uppercase I and R).

The method will take 2 parameters:

1. a number for a specific **row** to look in
2. the **piece** type of one of the players

This method should return True if there are 3 pieces side by side in the indicated **row** of the indicated **piece** type. Remember that we have the possibility of rows of lengths 3, 4, or 5. **You will have to create an implementation that finds 3 in a row in all of these situations**. You cannot just find the total number of pieces of the same type in the row. In a row of length 4, there may be 3 pieces with an EMPTY or opponent piece in between them, which prevents them from being 3 in a row. There are many ways to accomplish this; the key challenges are that in some games, rows will be 3 cells long, in others 4 or 5 cells and that there may be blanks or opponent pieces separating the piece type being checked.

Create a method named *winInCol* (notice the use of a lowercase w. but uppercase I and C).

This method has many similarities to the previous one. Instead of keeping your row the same and looking in different column indices, you will keep the column the same and change your row indices.

The method will take 2 parameters:

1. a number for a specific **column** to look in
2. the **piece** type of one of the players

This method should return True if there are 3 consecutive pieces in the indicated column. Remember that we have the possibility of columns of length 3, 4, and 5. You will have to create an implementation that finds 3 consecutive pieces in all of these situations. You cannot just find the total number of pieces of the same type in the column. In a column of length 4, there may be 3 pieces with an EMPTY or opponent piece in between them, which prevents them from being 3 in a row.

There are a few ways of doing these methods, including checking specific indices with several if-statements, or through a loop implementation. Either is a valid solution. Be aware that these methods are asking about a specific row, column, and piece type. **Returning True for 3 in a row for another piece type or in another row or column is a wrong implementation and in many cases, will be indicated by my tests failing.**

Part 5: Checking if a player has a win in a diagonal

We have a final sub-method to check if a player has a win in any diagonal direction.

Create a method named *winInDiag* (notice the use of a lowercase w. but uppercase I and D).

The method will take 1 parameter:

1. the **piece** type of one of the players

This method should return True if there are 3 consecutive pieces in a forward slash diagonal and a backward slash diagonal. Remember that we have the possibility of board diagonals of length 3, 4, and 5. You will have to create an implementation that finds 3 consecutive pieces in all of these situations. You cannot just find the total number of pieces of the same type in the diagonal. In a diagonal of length 4, there may be 3 pieces with an EMPTY or opponent piece in between them, which prevents them from being 3 in a row.

This method is more complicated than the previous row and column versions. You will notice that you are changing both your row and column at the same time. For one direction of diagonal, the row/columns will increase/decrease in the same direction. For the other direction, when the row goes up, the column will go down and vice versa. There are a few ways of doing this method, including checking specific indices with several if-statements, or through a loop implementation. Either is a valid solution. Be aware, that these methods are asking about a specific piece type. **Returning True for 3 in a row for another piece type or not in a diagonal is a wrong implementation, and my tests will fail.**

Part 6: Checking if a player has won

We will now use your sub-methods to determine if a player has won the current board. To check whether a player has won a game, you must check if the player has a win in any row, column, or diagonal direction.

Complete the method named *won* (notice the use of a lowercase w).

The method will take 1 parameter:

1. the **piece** type of one of the players

The method will return True if the player has a win in the board, or False if the player has not won. The following pseudo-code should help. Note that the if-statements are asking checking conditions for which we already created sub-methods. Make use of your sub-methods in those locations. Your solution should be the same length as this pseudo-code when complete.

```
For every row in the board
        If win in that row
                Return True
For every column in the board
        If win in that column
                Return True
If win in diagonal
        Return true
Otherwise, return false
```

Part 7: Giving the player an easy hint

You should now be able to play the game successfully. The hint method has not yet been implemented. We will implement a simple hint option that game code can use to give the user a location to play to either win on their next play or stop the opponent from winning.

Complete the method named *hint* (notice the use of a lowercase h).

The method will take 1 parameter:

1. the **piece** type of one of the players

The method returns a row and a column. Currently, this is a default value of -1,-1, which indicates there is no hint. We will add code before this default value that will attempt to find a hint of a spot that wins the game for the indicated player.

The following pseudo-code is an algorithm, which gives such a hint. It relies on the concept of checking every location on the board. The algorithm attempts to play the player's piece type temporarily. Then it checks if the player has won. If the player has won, then the algorithm reverts the board to what it was and returns the location. If the player has not won, then the algorithm reverts the board and moves to the next location.

```
For every row board
        For every column in the board
                If we can play at this row and column
                        Play the player's piece
                        If the player has won the game
                                Remove the player's piece from the last played location
                                Return the row and column
                        Otherwise,
                                Remove the player's piece from the last player location
Return -1 ,- 1
```

Once this method is implemented, you can enter -h as an argument. If there is a play that wins the board or blocks the computer, then that location will be highlighted on the board, and the row and column reported in the shell/command line window output for the user to see. My implementation of hint is around 10-12 lines of code.

There is an already completed hint "-a" that uses minmax tree AI to give a perfect hint. It should only be used for 3x3 boards. The Computer AI opponent that uses it is only available when playing 3x3 boards, as well.

Additional Requirements

- You must **not** modify any of the code that I have provided in *tictactoe.py*. (unless doing bonus)
- You should modify *Board.py* by:
    - adding your name and student number to the top of the file (and updating the comment at the top, if you feel inclined to do so),
    - adding the __init__, rows, cols, canPlay, play, winInRow, winInCol, and winInDiag and adding the comment that goes before each method, and
    - updating the bodies of full, won, and hint methods.
- You may assume that the methods that you are writing will be called with "reasonable" values. For example, it is *not* necessary to protect against scenarios such as __init__ being called with a negative or 0 number of columns.
- All lines of code that you write must be inside methods inside the class Board (except for the method definitions themselves).
- You must create and use the methods described previously in this document.
- Do not define one method inside of another method

- You must make appropriate use of loops. In particular, your program should work for game boards of various sizes (3x3, 3x4, 4x3, 4x4, 3x5, 5x3, 4x5, 5x4, 5x5). Various game board sizes can be tested by choosing different row and column combinations at the beginning of the game.
- **Include appropriate comments for each of your *Board.py* methods and for mine.** Each of your methods should begin with a comment that briefly describes the method's purpose, parameter, and return value. Methods that do not return values should be explicitly marked as such.
- **Your program must not use global variables** (except for constant values that are never changed, and in this assignment, you may not even want any global constants beyond the ones that I have already defined).
- Your program must use proper programming practices, which include appropriate variable names, detailed comments, minimizing the use of magic numbers, etc**. Your program should begin with a comment that includes your name, student number, and a brief description of the program.**
- Break and continue are generally considered bad form. As a result, you are **NOT** allowed to use them for this assignment. In general, their use can be avoided by using a combination of if statements and writing better conditions for loops.
- You should not import any other source code files than what has already been imported.

Hints

- Some functions require the number of rows or the number of columns of the board. These values can be determined by using the built-in *len()* function. In particular, the number of rows in the board is $\boldsymbol{len(board)}$**,** and the number of columns in the board is $\boldsymbol{len(board[0])}$.
- While my automated tests are reasonably thorough for boards with 3 or 4 rows and columns, they don't consider every possible case. You need to implement methods that provide the functionality described in this document, not just methods that pass the collection of provided tests.
- You can turn off tests for a specific part of the assignment by changing the constant at the top of the file. For example:

  TESTPART4 = False
  TESTPART5 = False
  TESTPART6 = False
  TESTPART7 = False

  will stop tests for your sub-methods of **won** and test for the **won** method itself. Turning these off may help with the speed of running your program on a slower laptop after you have a function implemented correctly already.

For an A+:

The current game can only be played for 3 in a row as a win condition. You may notice that games with large boards end rather quickly. To make the game more interesting, modify the assignment (for the bonus you will have to change *tictactoe.py* and *Board.py*) to take another argument from the user for how many pieces in a row should be required for a win. This value should be either 3, 4, or 5 and a valid length, i.e. Asking for 4 in a row on a 3x3 board should not be accepted. If the input is invalid, the program should exit as it does for other invalid arguments. If the argument is valid, modify the code of both *tictactoe.py* and *Board.py* so you can pass this value into all the required functions and change the methods in the Board class to use this new parameter correctly.

You can delete the test parts of the *tictactoe.py* file. There are no provided test for the bonus part.

This bonus will be particularly challenging, as you will have to change winInRow, winInCol, winInDiag, won, hint, and game over in *Board.py* to take an additional length parameter. As a result, each call to these functions made in *tictactoe.py* will have to be changed, as well. My recommendation is to keep a copy of your regular working assignment to submit in case you cannot finish the bonus. If your bonus does not work and you do not submit a regular version, your TA will not give your marks for non-functional code. You can submit both a bonus and the regular copy; your TA will mark the regular copy and then mark the bonus copy for the bonus parts.

My recommendation is to create a copy of *tictactoe.py* called *tictactoebonus.py*. You will have to disable (or simply delete) all the tests for the code, as they will no longer apply. Then create a new *Board.py* file called *BoardBonus.py* and change the import statement in *tictactoebonus.py* to import the Board class from this new file containing the new Board class. You will find that you have to change *tictactoe.py's* existing checkArgs, minmax1, minmax2, AI, and main functions to work with your new Board class methods that now take an additional length parameter. Every call to a changed function must also be changed to accommodate the new parameter.

This bonus is much more challenging than the previous ones. It should also demonstrate the challenges of major design changes after you have completed code with an initial design. Please do not attempt the bonus until you have completed the regular assignment. Your TA will grade the bonus by running and playing games of different sizes and different win lengths.

If you complete the A+ portion of the assignment, please submit both the original Board.by and the *BoardBonus.py/tictactoebonus.py*

Grading:

This assignment will be graded on a combination of functionality and style. A base grade will be determined from the general level of functionality of the program (Does it create the board correctly? Does it correctly implement canPlay, play, and full? Can it determine if the player has won? Does it provide correct hints? etc.). The base grade will be recorded as a mark out of 12.

Programming style will be graded on a subtractive scale from 0 to -3. For example, an assignment which receives a base grade of 12 (A), but has several stylistic problems (such as magic numbers, missing comments, etc.) resulting in a -2 adjustment will receive an overall grade of 10 (B+). Fractional marks will be rounded to the closest integer.

Mark	Letter Grade
12	A
11	A-
10	B+
9	B
8	B-
7	C+
6	C
5	C-
4	D+
3	D
0-2	F