CSCI 1133, Fall 2018 Due: 5:00 pm, Friday, Nov 16

Programming Examination 2

PLEASE NOTE: This is a graded assessment of individual programming understanding and ability, and is ***not*** a collaborative assignment; you must design, implement and test the solution(s) completely on your own without outside assistance from anyone. You may not consult or discuss the solution with anyone. In addition, you may not include solutions or portions of solutions obtained from any source other than those provided in class. Note that *providing* a solution or assisting another student in any way on this examination is also considered academic misconduct. Failure to heed these directives will result in a failing grade for the course and an incident report filed with the Office for Student Conduct and Academic Integrity for further sanction.

A. (100 points) **Othello**

This examination consists of designing and writing a single well-structured Python program that must be committed and pushed to your remote GitHub examination repository *prior* to the deadline. Note that your program *must* be named `othello.py` (all lower-case).

Late submissions will not be accepted and will result in a zero score for the exam.

TA help for this examination will not be provided. If you have clarification questions, they must be addressed to the graduate TAs for the class.

The total point value will be awarded for solutions that are *complete*, *correct*, and *well structured*. A *"well structured"* program entails good design that employs meaningful functional decomposition, appropriate comments and general readability (descriptive names for variables and procedures, appropriate use of blank space, etc.) If you are not sure what this means, review the "well-structured" program requirements provided in Lab2.

Note that your work will be graded using, and must function correctly with, the current version of Python 3 on CSE Labs UNIX machines. If you complete this programming exam using a different system, it is *your* responsibility to ensure it works on CSELabs machines prior to submitting it.

The rubric includes but is not limited to the following specific (accumulative) point deductions:

- Missing academic integrity pledge -100 points
- Syntax errors -50 to -100 points
- Misnamed source file or incorrect repository -25 points
- Use of global variables -25 points
- Missing main program function -25 points

Examination Repository

Examination files must be submitted to GitHub using your remote exam repository. Exam repositories have already been created for each registered student and are named using the string `exam-` followed by your X500 userID (e.g., `exam-smit1234`). You must first clone your exam repository in your local home directory before submitting the materials for this exam. If you are having difficulty, consult the second Lab from earlier in the semester or the GitHub tutorial on the class webpage. If your exam repository is missing or something is amiss, please contact the graduate TA. Do not create any additional folders or directories in your exam repository. DO NOT SUBMIT YOUR EXAM FILES TO YOUR LAB/EXERCISE REPOSITORY!

Background

The modern board game, *Othello,* is a commercially produced version of a game that originated in the 19th century called 'Reversi'. It's an interesting 2-player game of strategy played on an 8x8 grid in which each player places tokens on the board in order to cover more grid cells than their opponent. Each token is a two-sided disk with white on one side and black on the other. The basic premise of the game is that as each player plays a token of his/her color (white or black), they 'flip' over all the intermediate tokens of their opponent. The game ends when all the grid cells have been covered with tokens, or no allowable move can be made.

Read the following Wikipedia description of the game:

www.wikipedia.org/wiki/Reversi

The following website has an interactive version that you can play. You should try it to make sure you are comfortable with how the game is played. Note that there are rules for how the game is played on this website as well, read them thoroughly as they will help you understand what your program should do.

www.othelloonline.org

Program Requirements

Using Turtle graphics, write a well-structured Python program named `othello.py` that will play an interactive game of *Othello*. Your program will pit a human player against the computer.

Your program must do the following:

- Include the academic integrity pledge in the program source code (details below)
- Use turtle graphics to implement the graphical interface (set the turtle speed to 0 in order to speed things up).
- Include numeric row and column number "headers" in the graphical display to assist the human player in determining what to enter.
- Maintain the board state using a 2-dimensional list (matrix) with 8 rows of 8 columns. Each board location must indicate if it has a black token, a white token or is unoccupied. The upper-left cell is board location [0][0], and the lower right cell is board location [7][7]. Careful: this is not oriented like a graph!
- The "human" player will play black and the computer will play white. The human player will always go first. Your program must include a loop that will alternate between soliciting the human player's move (row, column) and making the next "computer" play. The game continues until the entire board is occupied or until either player can no longer make a valid move. The game can also be terminated prematurely by entering a null-string as input.
- A message should appear when the game ends indicating which player is the winner, along with the final score.
- Avoid using global variables (global *constants* are perfectly fine). You will need to pass the "board" matrix as an argument to the various functions that need it.
- Solicit input using the turtle `.textinput()` function. The input should consist of the row and column for the next (human) play.
- Include a pure function: `selectNextPlay(board)` that will be called to get the computer's next play. You can be as creative as you wish, however it is acceptable to simply make a random choice from any of the remaining *valid* moves.
- Include a pure Boolean function: `isValidMove(board,row,col,color)` that will return `True` if the specified row,column is a "valid" move for the given color and `False` otherwise. Note that for a move to be valid, it must satisfy three conditions:
    1. It must be unoccupied
    2. At least one of its neighbors must be occupied by an opponent's token (opposite of `color`)

3. At least one of the straight lines starting from row,col and continuing horizontally, vertically or diagonally in any direction must start with an opponent's token and *end* with a token matching the `color` argument.

- Include a pure function: `getValidMoves(board,color)` that will scan every cell of the board and return a list of (row,column) tuples that are *valid* plays for the indicated token color.
- You can find descriptions of the `.textinput()` and `.write()` turtle functions (along with all the Turtle graphics module functions) here: *docs.python.org/3/library/turtle.html* .

Constraints:

- You may use any *built-in* Python object class methods (string, list, etc.)
- You may use *imported* functions and class methods from the `turtle, random` and `math` modules only

Academic Integrity Pledge

The first lines of your program *must* contain the following text (exactly as it appears), replacing the first line with your full name and X500 ID. If this statement does not appear in your program source, you will receive a score of zero for the exam.

```
# <replace this line with your name and x500 ID (e.g., John Smith smit1234>

# I understand this is a graded, individual examination that may not be
# discussed with anyone.  I also understand that obtaining solutions or
# partial solutions from outside sources, or discussing
# any aspect of the examination with anyone will result in failing the course.
# I further certify that this program represents my own work and that none of
# it was obtained from any source other than material presented as part of the
# course.
```

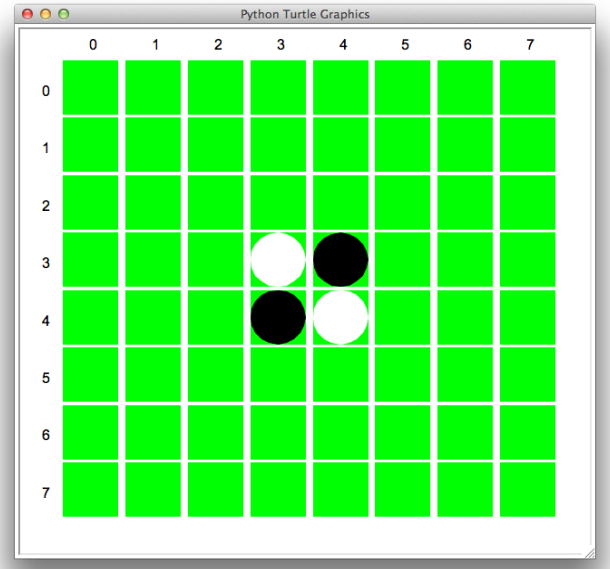
Suggestions and Hints:

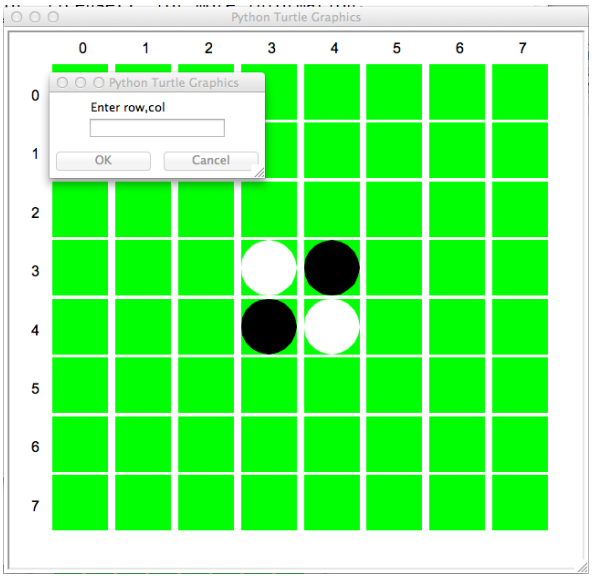
- You will find this project much easier if you employ the principles of "top-down, functional decomposition", as discussed in lecture, and implement your program as a collection of short, simple functions. Think about the tasks your program needs to complete, and write simple functions to do each of those tasks.
- Tackle problems one at a time. First, for example, try to draw the board to make sure you understand how the coordinate system should work. The graphical display is fairly uncomplicated if you set the world coordinates to match the board (leaving some room for the border) and use the turtle.stamp() method. You simply change the turtle shape to "square" and "stamp" it at each grid cell location.
- Once you've created the board, you will need to think about the problem of getting input from a user and deciding if this input is valid. Plan to spend time considering the logic behind when a move is valid. We recommend drawing this problem out on a piece of paper and drawing out different scenarios. How should you decide if a move should be allowed? You will need to reference the rules of the game. It may be more helpful to you to think about when a move should be "illegal."
- To play tokens, you can change the turtle shape to "circle" and "stamp" the token on the board (in the appropriate color). To "flip" tokens, you need only stamp over the existing token in a new color. For example, if a token needs to be "flipped" from black to white, you can simply stamp a white token directly over the existing black token.
- You will find it useful to construct (and use!) a "helper" function that will convert "row, column" board coordinates to "x, y" turtle coordinates! Remember that the board is oriented with [0][0] in the upper left corner! You can look at the examples below to see how the board coordinates should be oriented.
- The most complicated aspect of this program is determining the lines of opponent tokens to "flip" when a move is made (hint: this is also needed to determine if a proposed move is "valid"). One way to do this is to subtract the corresponding row and column coordinates of the first two cells in a "line" (horizontal, vertical, diagonal) to obtain a row and column "delta value." This delta value can be

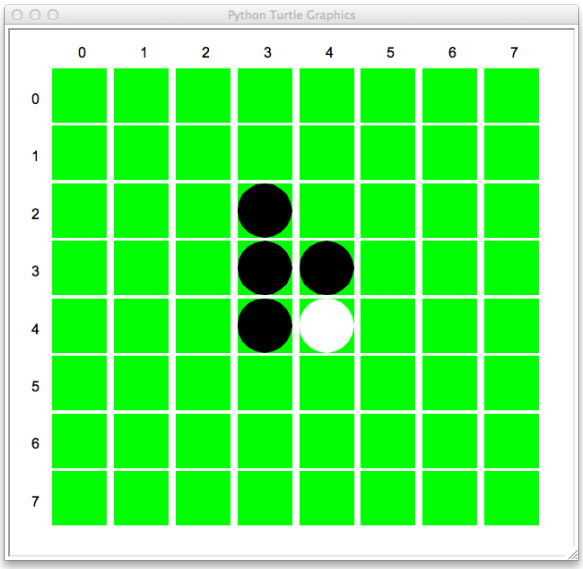
used in an iteration loop to identify all the cells in that line (draw this out on a piece of paper and make sure you understand why this is true!). For example, suppose a player wishes to play a black token at position (3,3). If you examine the neighbors of (3,3) and find that at position (4,4) there is a white token, you may observe that this indicates the diagonal line of tokens to the lower right of (3,3) is a "candidate line" of tokens to be flipped. In order to determine if the white tokens should be flipped, you would need to examine all tokens in the "candidate line." The criteria for flipping the white tokens is that they must have a black token at both ends of the line and white tokens in the middle. So, you would want to keep moving your token by 1 in the x direction (your row delta value) and 1 in the y direction (your column delta value) until you either reach a space with no tokens (NOTE - this would mean no tokens should be flipped!) or until you reach a black token.

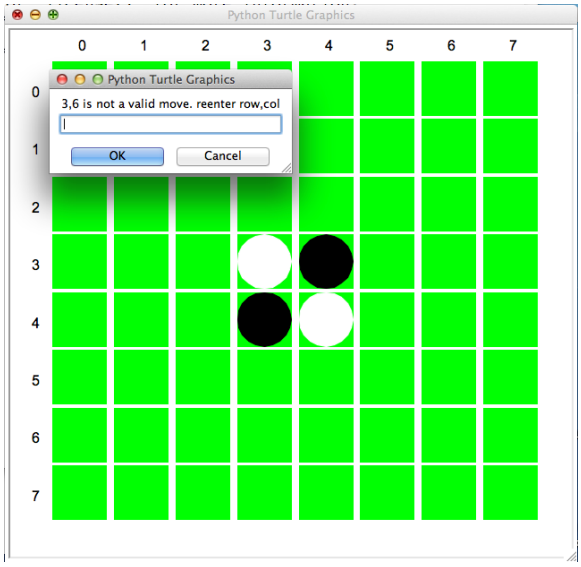
- Break large problems into smaller problems. For example, before you try to write the function `getValidMoves(board,color),` make sure you understand how to determine the "candidate lines" described above.
- Another challenge in this problem is dealing with the grid border. For example, the cell at [0][0] does not have 8 neighbors, it has 3. You might find it useful to construct a helper function that returns a list of neighbors, given a cell's row and column. Another useful helper function would return a Boolean indicating if a particular row, column pair is "inGrid". Note that an "inGrid" cell should have eight neighbors - don't forget about the diagonal neighbors! You will want to think through the mathematics of "neighbors" - it is VERY helpful to draw out the grid on a piece of paper to help figure out the mathematical expressions. If a cell is "inGrid" at location (i, j), note that it has, for example, a diagonal neighbor at location (i-1, j+1). You will want to figure out the possible locations of all 8 neighbors. Note also that if a cell is not "inGrid," you need to figure out which border the cell is on in order to determine which neighbors it has.
- Start early.

Example Program Output






Python Turtle Graphics
0 1 2 3 4 5 6 7
0
1
2
3
4
5
6
7


Python Turtle Graphics
0 1 2 3 4 5 6 7
Python Turtle Graphics
3,6 is not a valid move. reenter row,col
OK
Cancel
0
1
2
3
4
5
6
7