Project 10: The Game of Reversi

Edit1 4/9: added “or diagonal” to Game Rules and Objective.

This assignment is worth 55 points (5.5% of the course grade) and must be completed and turned in before 11:59pm on 4/16/2018.

Assignment Overview:
This assignment will give you more experience on the use of custom built Python classes. In this project you are going to write a Python program that enforces the rules of the game of **Reversi**.

Assignment Background:
You will implement a classical board game called **Reversi** in Python using classes. **Reversi** is a strategy board game for two players, played on an 8×8 uncheckered board. There are sixty-four identical game pieces called ***Pieces*** (often spelled "discs"), which are light on one side and dark on the other.

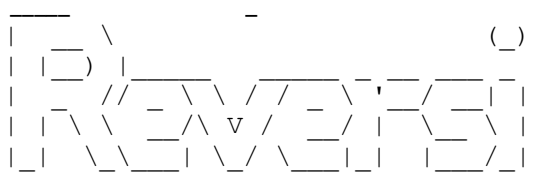
Game Rules and Objective:
Players take turns placing disks on the board with their assigned color facing up. During a play, any disks of the opponent's color that are in a straight line (horizontal, vertical or diagonal) and bounded by the disk just placed and another disk of the current player's color are turned over to the current player's color. The objective of the game is to have the majority of disks turned to display your color when the last playable empty square is filled. More details on this game can be found here:

https://en.wikipedia.org/wiki/Reversi

In fact, the best way to learn the rules is to play the game first hand, you can play the game here https://www.coolmath-games.com/0-reversi

The reversi.py file, Board and Piece class:
The Board and Piece classes are defined in the reversi.py file. So you have all the tools already to be used in a game. You will use this class to write your game logic. **You must not modify this class.** The first thing you need to do is to **read that file and understand each function and variables in both of the classes**. There is also a main function that tests different functionalities of Board and Piece objects. Read, run and see the main function outputs to understand how the classes work.

The gameplay:
Unfortunately we don't have any nice user interface for the game. What you are going to implement is a text based command line interface. When you start/run the game, it will look like this:



```
Developed by The Students Inc.
CSE231 Spring Semester 2018
Michigan State University
East Lansing, MI 48824, USA.

Pick a color:
```

First the game will ask for which color you are going to choose, then you type black or white. You and your opponent's colors are fixed. **In our case, white will always play the first move.** The game will start as follows --

```
 ______                          _
|  __  \                        (_)
| |__)  |______   ______ _ __ ___ _
|  _   // _ \ \ / / _ \ '__/ __| |
| | \ \  __/\ V /  __/ |  \__ \ |
|_|  \_\___| \_/ \___|_|  |___/_|

Developed by The Students Inc.
CSE231 Spring Semester 2018
Michigan State University
East Lansing, MI 48824, USA.

Pick a color: black
You are 'black' and your opponent is 'white'.
Current board:
    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
a |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
b |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
c |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
d |   |   |   | ▒ | ▓ |   |   |   |
  +---+---+---+---+---+---+---+---+
e |   |   |   | ▓ | ▒ |   |   |   |
  +---+---+---+---+---+---+---+---+
f |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
g |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
h |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+

  Black: 2, White: 2
[white's turn] :>
```

The board is composed of 8x8 “cells” and initialized with 2 black and 2 white pieces placed diagonally in the center (the blacks are on right diagonal and the whites are on the left diagonal). The rows on the board is numbered as a, b, c, ..., h and the columns as 1, 2, 3, ..., 8 etc. The game

waits on a prompt that reads [white's turn] :> . Then a player will type a board position to place a piece, like this:

```
[white's turn] :> d6
     white played d6.
Current board:
    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
a |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
b |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
c |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
d |   |   |   | ▒ | ▒ | ▒ |   |   |
  +---+---+---+---+---+---+---+---+
e |   |   |   | ▓ | ▒ |   |   |   |
  +---+---+---+---+---+---+---+---+
f |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
g |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
h |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+

  Black: 1, White: 4
[black's turn] :>
```

Here you can see the white has captured the black piece at d5 and the board is updated accordingly. The turn of the player has also been changed to black. The game will continue like this until the entire board is filled or there is no valid move left for both players. The user can also type hint to get the list of positions that capture opponent's pieces, like this:

```
[black's turn] :> hint
     Hint: e6, c6, c4
Current board:
    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
a |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
b |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
c |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
d |   |   |   | ▒ | ▒ | ▒ |   |   |
  +---+---+---+---+---+---+---+---+
e |   |   |   | ▓ | ▒ |   |   |   |
  +---+---+---+---+---+---+---+---+
f |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
g |   |   |   |   |   |   |   |   |
```

```
  +---+---+---+---+---+---+---+---+
h |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+

  Black: 1, White: 4
[black's turn] :>
```

The hint command displays all possible moves for black, in this case, they are e6 (to horizontally capture the white piece at e5), c6 (to diagonally capture the white piece at d5) and c4 (to vertically capture the white piece at d4).

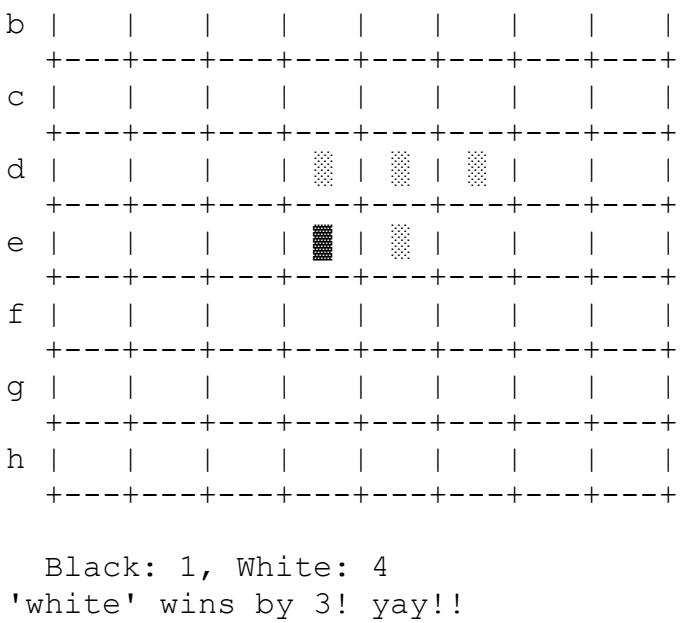
In some situations, a player might not have any valid move. In that case the player can pass the board to the opponent by typing pass on the prompt, like this:

```
[black's turn] :> pass
     Can't hand over to opponent, you have moves, type 'hint'.
Current board:
    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
a |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
b |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
c |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
d |   |   |   | ▒ | ▒ | ▒ |   |   |
  +---+---+---+---+---+---+---+---+
e |   |   |   | ▓ | ▒ |   |   |   |
  +---+---+---+---+---+---+---+---+
f |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
g |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
h |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+

  Black: 1, White: 4
[black's turn] :>
```

Since the black still has valid capturing moves, the pass did not happen. Instead, it prints a message saying that a pass is not possible in this case, otherwise the prompt would have changed to [white's turn] :>. A player can leave the game anytime by typing exit on the prompt, like this:

```
[black's turn] :> exit
Current board:
    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
a |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
```



```
  Black: 1, White: 4
'white' wins by 3! yay!!
```

The game exits by showing who is the winner with the winning difference of the piece counts.

Project Specifications:
You need to complete the functions in the proj10.py file (along with code and function headers and comments), where all the gameplay logic and mechanisms will emulate the steps described previously. The last function, game_play_human() will use all the functions to implement the gameplay. The functions that you need to complete are as follows:

1. def indexify(position):
This function converts the letter-number position to row-column indices. The input position is a string like a1, h8, z23 etc. For example, for a 8x8 board:

if the input is a1 then it will return a tuple (0,0)
if the input is a2 then it will return a tuple (0,1)
.
.
if the input is z23 then it will return a tuple (25,22)
and so on ...

Hint: This function does not check if the input is a valid position. Note that all numeric positions start from 0, not 1. I used a dictionary to map from letters to numbers (for a challenge create the dictionary in one line using comprehension; enumerate was useful).

2. def deindexify(row, col):
This function does the exact opposite of indexify(). In this case, the input is row and column values and it returns strings like a1, h8 etc. For example, for a 8x8 board:

if the input is 0, 0 then it will return a string a1
if the input is 0, 1 then it will return a string a2
.
.
if the input is 25, 22 then it will return a string z23

and so on ...

Hint: This function does not check if the input is a valid position. Note that all numeric position starts from 0, not 1. I used a dictionary to map from numbers (for a challenge create the dictionary in one line using comprehension; enumerate was useful).

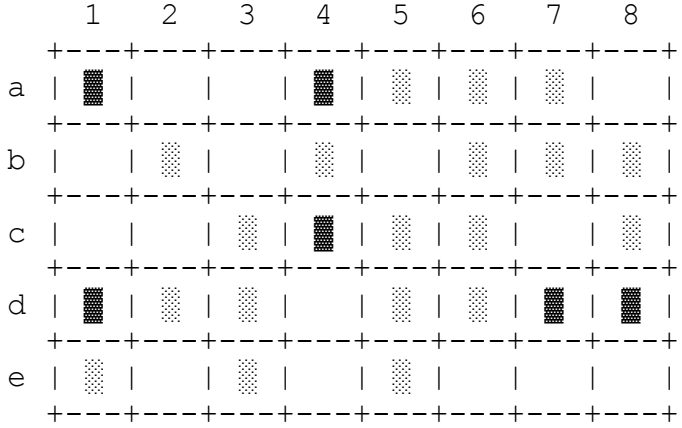
3. def initialize(board):
This function puts 2 black and 2 white pieces in the middle of the board. The black pieces will be placed diagonally right and the white pieces will be places diagonally left, so that the black and the white pieces are diagonal with respect to each other. **This function must work for any board size; be sure to handle odd and even sides.** There is only one way to place the 4 pieces for even board sides (for 8x8 white at 'd4' and 'e5'; black at 'd5', and 'e4'); for odd-sized boards place smaller than the middle, e.g. for 9x9 place at the same positions as in 8x8. Note that the board exists before this function is called—the function simply places initial pieces on an existing board.

4. def count_pieces(board):
Counts the total number of black and white pieces currently on the board. The counts will be returned as a tuple where the count of the black will come first. For example, if black count is 10 and white count is 15 then it will return a tuple (10,15). Hint: you can find the color of a piece and you can find the pieces on the board—check every square of the board.

5. def get_all_streaks(board, row, col, piece):
This is the most complex function in this project and we are providing it. You should understand how it works. This function finds all capturing streaks if a piece is placed on the row, col position. This function returns a dictionary of all capturing streaks in all 8 directions keyed as east, west, north, south, ..., south-east, south-west etc. **These directions should be understood as the typical cartographic/compass directions on a map starting with *up* being *north*.** An empty dictionary called streaks is initialized in the beginning of the function, and the rest of the function populates each of the None entries with capturing streaks and return it:

```
streaks = {'e': None, 'w': None, 'n': None, 's': None, \
           'ne': None, 'nw': None, 'se': None, 'sw': None}
```

Here the key *e* stands for *east*, *w* stands for *west*, *sw* stands for *south-west* etc. For example, if we consider this board:



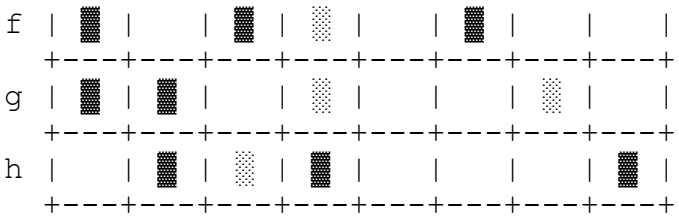


Figure 1: Example Board

and if we place a **black** piece at position `d4` (i.e. `(3,3)`) then this function builds up the following dictionary and returns it:

```
streaks = {'e': [(3, 4), (3, 5)], 'w': [(3, 1), (3, 2)], 'n': [], 's': [],\
           'ne': [], 'nw': [(1, 1), (2, 2)], 'se': [(4, 4)], 'sw': []}
```

Examine the board in Figure 1 carefully, and try to understand why there are no valid capturing streaks along the *n, s, ne* and *sw* directions.

The function sorts each list before it adds them to the `streaks` dictionary.

6. def get_all_capturing_cells(board, piece):
Given a board and a piece, this function returns a list of all capturing streaks found from all empty positions on the board in a dictionary. Where the key of the dictionary is the (row, col) position tuple and their corresponding values are the list of capturing streaks. Basically, this function calls the `get_all_streaks()` function on each empty cell on the current board, and builds a dictionary by using the `(row, col)` tuple as the key and the sorted list of cells that it can capture from all the directions returned by the `get_all_streaks()` function. For example, if we consider the board in the Figure 1 above, this function will return this dictionary:

```
{(1, 2): [(2, 2), (3, 2), (4, 2)], (7, 4): [(6, 3)], (5, 4): [(5, 3)],
(2, 6): [(2, 4), (2, 5)], (1, 4): [(2, 5)], (3, 3): [(1, 1), (2, 2),
(3, 1), (3, 2), (3, 4), (3, 5), (4, 4)], (4, 3): [(5, 3), (6, 3)], (4,
5): [(3, 4)], (4, 1): [(3, 2)], (0, 7): [(0, 4), (0, 5), (0, 6), (1,
7), (2, 7)], (2, 1): [(2, 2)]}
```

If you look at the entry with key `(3,3)`, the corresponding value is a list of cells: `[(1, 1), (2, 2), (3, 1), (3, 2), (3, 4), (3, 5), (4, 4)]` which is basically a list of all the cells found from all directions that is returned by calling `get_all_streaks(board, 3, 3, piece)`. That function returns a dictionary that needs to be converted to a list of cells—then sort the resulting list (for consistent testing on Mimir). Note that a valid move (capturing move) is defined as a move that captures at least one piece from opponent. This function should not save invalid or non-capturing cells in the dictionary.

7. def get_hint(board, piece):
This function returns the list of board positions like `['a1', 'g5', ..., 'f7']` etc. This function calls the `get_all_capturing_cells()` function and gets the streaks for each empty position on the board. Then it sorts the list of positions with respect to the total length of the capturing streaks and returns it. That is, count the total cells in the list of lists that is each value in

the dictionary—sort on that value from highest to lowest. Then, for consistency in Mimir testing, if two have the same number of cells also order from highest to lowest, e.g. if positions 'e4' and 'd3' have the same number of cells then 'e4' will come before 'd3'. For the example board given the Figure 1 above, this function will return this list:

```
['d4', 'a8', 'b3', 'e4', 'c7', 'h5', 'f5', 'e6', 'e2', 'c2', 'b5']
```

8. def place_and_flip(board, row, col, piece):
This function calls the `get_all_streaks()` function to get all the capturing streaks from a position row, col. Then places the piece to that position and flips all the pieces along the streak positions found from the `get_all_streaks()` function. This function throws a `ValueError`: i) If the `row,col` position does not yield any capture or, ii) If the row, col position is already occupied or, iii) The position is outside of the board.

9. def is_game_finished(board):
A game is finished when black or white is left with no possible move or the board is full. This function returns a Boolean True/False.

10. def get_winner(board):
Gets the current winner. Counts the number of black and white pieces and decides which player is the winner. Returns a string, which is either 'black', 'white' or 'draw'. Use the `count_pieces()` function here.

11. def choose_color():
This function asks for a color inside a loop until a valid color name i.e. 'black'/'white' is entered. If a wrong color or an arbitrary string is entered, it will print an error message. Once it receives a correct color assignment, it will store those color values in to two variables called my_color and opponent_color and return them as a tuple (my_color, opponent_color).

12. def game_play_human():
This function implements the main gameplay mechanism by stitching up all the functions that have written so far. **This function is already completed** but you need to follow the code to understand the structure of the code and the gameplay.

Deliverables:
The deliverable for this assignment is the following file:

proj10.py – the source code for your Python program

Be sure to use the specified file name and to submit it for grading via the Mimir system before the project deadline.

Sample gameplay outputs and Mimir tests:
The Mimir system will test all the functions specified in this project description. Mimir will also test gameplay examples specified in play01.in by comparing them with the play01.out. There is also a hidden test whose input/output files will not be provided to the students. **You are strongly recommended to read the example gameplay outputs in play01.out file. They can be opened with notepad/wordpad or text-editor on Mac.**

Grading Rubric:
The most part of the project will be graded by Mimir's automated tests and the score allocations are done as follows:

General Requirements:

(5 pts)	Coding standard 1-9, see the cse231 course web page.

Implementations:

(4 pts)	indexify() and deindexify() tests
(3 pts)	initialize() test, must work for any board size
(3 pts)	count_pieces() test
(6 pts)	get_all_capturing_cells() test
(6 pts)	get_hint() test
(6 pts)	place_and_flip() function, must throw ValueError
(2 pts)	is_game_finished() test
(2 pts)	get_winner() test
(3 pts)	choose_color() no Mimir test
(7 pts)	Test 1 (play01.in/.out test, files are on the project page)
(8 pts)	Hidden Test