ICS 32 Fall 2016
Project #4: *The Width of a Circle (Part 1)*

Due date and time: *Monday, November 21, 11:59pm*

This project is to be done individually

Background

My first exposure to computers, as a kid in school, was in the context of computer games; some were educational games (it was school, after all), though many were not. The first time I remember sitting behind a computer — a Radio Shack TRS-80 Model I — I played a game called FASTMATH, which pitted two players against one another, trying to alternately solve arithemtic problems and type in the answers as quickly as possible. Sure, it was just a boring educational game, one that was ridiculously simple by today's standards, but at the time I was captivated, and I still remember it to this day. (I especially loved winning, though I didn't always win.)

Thanks to the wisdom and generosity of my parents, it wasn't long before I had my own computer at home (a Commodore 64), complete with its own collection of games. None of the games I played on my own computer could be classified as educational in a direct sense, though those games were sneaky: They taught me a surprising collection of lessons and motivated me to ask many interesting questions about computing, as I endeavored first to win them outright, then to modify them (to cheat or to change how a game was played to make it more fun), and finally to write them from scratch. Games in those days, of course, didn't have the same photorealistic, three-dimensional, surround-sound appeal that they have today, but they were nonetheless fun and exciting; their simplicity made writing one's own game seem more possible with limited skills than it does today, in an era of tremendously complex games built by gigantic teams of programmers, designers, and artists. (In truth, it's easier to build simple games now than it was then, because the computers have become so much more powerful and the tools have gotten better. It's just harder to compete with the large-scale, professionally-developed games.) Unfortunately, my skills didn't develop quickly enough — I always aimed too high, relative to what I knew how to do, but it was tougher when there was no Internet to search when you got stuck on something — and I never realized the goal of writing my own games before I became interested in other things, though I certainly learned a lot trying.

This project is the first of a two-part sequence that offers you to opportunity to build your own game. The first of the two projects focuses on developing a clean set of game logic and a console-mode "outer shell" that you'll use to test it. The second one pivots into building a graphical user interface atop the same game logic, focusing on drawing graphics and handling a variety of user input. Along the way, we'll focus on finding a design that serves both purposes, on finding ways to simplify our code by eliminating duplication of boilerplate, and continuing our journey into understanding the mechanics and the benefits of classes and object-oriented programming in Python.

Games may seem frivolous to some of you — I know that not everyone likes to play them — but they provide a fascinating combination of problems to be solved: software engineering, human-computer interface, computer networks, psychology and cognition, and even (in multiplayer online games) economics and sociology. Game developers push the envelope — in some cases further than just about any other kind of software developers —

and many of these lessons can be applied in more seemingly serious contexts. Even if you're not that interested in games, you'll be surprised what building games can teach you about software.

The game of Othello

This project and the subsequent one will ask you to implement a game called Othello. Othello (also known as Reversi) is a well-known two-player strategy game. The game is played on a rectangular board divided into a grid — usually 8x8, though the size of the grid can vary. Players alternately place *discs* on the game board; one player's discs are black and the other player's are white. When discs are placed on the game board, other discs already on the board are *flipped* (i.e., a black disc becomes a white disc or vice versa). The game concludes when every square on the grid contains a disc, or when neither player is able to make a legal move; the winning player is generally the one who has more discs on the board at the end of the game, though there are alternate ways to determine a winner.

The rules of the game, along with some notion of strategy, are described in the Wikipedia entry on Reversi. If you haven't played Othello before, or have seen it previously but don't remember how it works, you should at least read the sections of the Wikipedia entry that cover the rules of the game; knowing how the game is played before proceeding with this project is vital. If you want to try playing the game, a web-based version of it is available.

The program

This project asks you to implement the game logic of Othello, along with a console-mode program with a very spartan user interface that you'll use to test it — and that we'll use to *automatically* test it, making it crucial that you get the format of the program's input and output right according to the specification below. Spacing, capitalization, and other seemingly-minor details are critical.

Using your test program, you'll be capable of playing a single game of Othello on a single computer. There are a handful of options that allow you to specify, before the game begins, how the game will be played. The program begins by reading input that selects these options; the game is then played by requiring input directly into the console that specifies a player's next move, continuing until the game is complete, at which point the program ends.

When you're finished, you'll have the game logic that will form the basis of your completed version of Othello in the next project.

A detailed look at how your program should behave

Your program will take its input via the console, *printing no prompts to a user*. The intent here is not to write a user-friendly user interface; what you're actually doing is building a tool for testing your game logic, which we'll then be using to *automatically* test your game logic. So it is vital that your program reads inputs and writes outputs precisely as specified below. You can freely assume that the input will match the specification described; we will not be testing your program on any inputs that don't match the specification.

- Your program begins by printing a line of output specifying which of the two sets of Othello rules are supported by your program, by printing either **FULL** or **SIMPLE**. (See the section titled *Simplified Othello rules* below for a description of the simplified set of rules, which you can implement for partial credit if you're unable to complete the full rules.)
- Your program then reads five lines of input, specifying the options that will be used to determine how the game will be played.

  - The first line specifies the number of rows on the board, which will be an *even integer* between 4 and 16 (inclusive).
  - The second line specifies the number of columns on the board, which will be an *even integer* between 4 and 16 (inclusive) and *does not* have to be the same as the number of rows.
  - The third line specifies which of the players will move first: **B** for black, **W** for white.
  - The fourth line specifies the arrangement of the cells on the board when the game starts. According to the rules, the game begins with four discs on the board: two white and two black, arranged on the four center cells of the grid, with the two white discs separated diagonally and the two black discs separated diagonally. This line selects which color disc will be in the top-left position of these four center cells: **B** for black, **W** for white.
  - The fifth line selects how the game is won: **>** means that the player with the most discs on the board at the end of the game wins, **<** means the player with the fewest.
- After specifying the options, the game begins, proceeding one move a time until it's complete.
  - Before each turn, the following information is printed to the console:
    - The character **B**, followed by a colon and a space, followed by the number of discs on the board that are black. This is followed by two spaces, the character **W**, a colon and a space, and the number of discs on the board that are white.
    - The contents of the board, with each row occupying one line of output, and one space separating each pair of cells. Each cell is written either as a **B** (if a black tile occupies that cell), a **W** (if a white tile occupies that cell), or a **.** (if no tile occupies that cell). For example:

```
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . B W . . .
. . . W B . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
```

    - The word **TURN** (in all-caps), followed by a colon and a space, followed by **B** if it's the black player's turn, **W** if it's white's turn.
  - After printing the information above, the program reads a line of input specifying the current player's move. The move is specified as two integers on a line, separated by a space. The first integer specifies the row number (with 1 being the top row, 2 being the row below that, and so on) and the second integer specifies the column number (with 1 being the leftmost column, 2 being the one to the right of that, and so on).
    - If the move is valid, print the word **VALID** alone on a line, apply that move (i.e., make the appropriate changes to the state of the game) and then continue the game..
    - If the move is invalid (see the rules if you're curious what makes a move invalid), print the word **INVALID** alone on a line and wait for the user to specify another move.
- At the conclusion of the game (i.e., after the last move is made and there are no more legal moves on the board), the final state of the game is printed to the console, quite similarly to how it's printed before each turn:
  - The character **B**, followed by a colon and a space, followed by the number of discs on the board that are black. This is followed by two spaces, the character **W**, a colon and a space, and the number of discs on the board that are white.
  - The contents of the board, in the same format described above.
  - The word **WINNER** (in all-caps), followed by a colon and a space, followed by **B** if the black player has won, **W** if the white player has won, and **NONE** if no player has won (i.e., the final score is tied).

The program ends when the game is over.

A complete example of program execution

The following is an example of the program's execution, if implemented as it should be, including the full Othello rules. Boldfaced, italicized text indicates input, while normal text indicates output.

```
FULL
4
4
B
B
>
B: 2  W: 2
. . . .
. B W .
. W B .
. . . .
TURN: B
2 4
VALID
B: 4  W: 1
. . . .
. B B B
. W B .
. . . .
TURN: W
1 2
VALID
B: 3  W: 3
. W . .
. W B B
. W B .
. . . .
TURN: B
1 4
INVALID
1 1
VALID
B: 5  W: 2
B W . .
. B B B
. W B .
. . . .
TURN: W
1 4
VALID
B: 4  W: 4
B W . W
. B W B
. W B .
. . . .
```

```
TURN: B
1 3
VALID
B: 7  W: 2
B B B W
. B B B
. W B .
. . . .
TURN: W
3 4
VALID
B: 5  W: 5
B B B W
. B B W
. W W W
. . . .
TURN: B
4 2
VALID
B: 7  W: 4
B B B W
. B B W
. B W W
. B . .
TURN: W
2 1
VALID
B: 5  W: 7
B B B W
W W W W
. B W W
. B . .
TURN: B
4 4
VALID
B: 8  W: 5
B B B W
W B W W
. B B W
. B . B
TURN: W
4 1
VALID
B: 7  W: 7
B B B W
W B W W
. W B W
W B . B
TURN: B
3 1
VALID
```

```
B: 10  W: 5
B B B W
B B W W
B B B W
W B . B
TURN: W
4 3
VALID
B: 8  W: 8
B B B W
B B W W
B B W W
W W W B
WINNER: NONE
```

A couple of "gotchas" to be aware of in the game logic

For the most part, Othello games proceed with players moving alternately, and continue until all cells on the grid contain a disc. However, there are a couple of wrinkles that you'll need to be sure you handle:

- Sometimes, a player will make a move and, as a result, the opposite player will have no valid moves available (i.e., there is no cell in the grid in which the opposite player can move afterward). In that case, the turn reverts back to the player who just moved, and that player moves again.
- Occasionally, neither player will have a valid move on the board, even though there are still empty cells in the grid. (One example is when all of the discs on the board belong to one player, though there are others, as well.) In this case, the game immediately ends and the winner is determined based on the number of discs each player has on the board.

Simplified Othello rules

Implementing Othello can be a challenge, so we're providing you with the option of implementing a simplified set of Othello rules instead of the full rules described in the Wikipedia entry on Reversi. Implementing the simplified rules will leave you ineligible for full credit on the project, by capping your *Correctness and robustness* score at 12 points out of a possible 20, but a finished version of the simplified rules will almost certainly score higher than an incomplete version of the full rules, since we'll at least be able to test it successfully. Note, too, that your score on Project #5 will not be affected; either set of rules is allowed on that project, with no penalty for the simplified version, since the focus of that project is building a graphical user interface atop your existing game logic from this one.

The simplified Othello rules are as follows:

- The same configuration options need to be specified in the same order (i.e., number of rows and columns, who moves first, how the tiles are arranged at the start of the game, and whether more or fewer tiles wins the game) and they have the same meanings as in the full rules.
- A player can make a valid move anywhere on the board that is adjacent to a tile (i.e., one cell away in any of the eight directions) occupied by a tile belonging to the other player.
- When a player makes a move, any adjacent cell on the board (i.e., one cell away in any of the eight directions from where the move was made) containing a tile of the opponent's color is flipped.
- The game ends when neither player can make a legal move — when the board is completely filled with tiles, or when no open cells on the board are legal for either player.

The format of the input and output are also identical to what's described in the section titled *A detailed look at how your program should behave* above; the only difference is the first line of the output (**SIMPLE** instead of

FULL) and the logic for determining whether a move is legal and which tiles are flipped.

Be aware that you do have to specify the first line of the output correctly, so we'll know which tests to run against your program. You can implement *either* the full Othello rules *or* the simplified ones, and you'll need to begin by outputting either **FULL** or **SIMPLE** so we'll know which one you've chosen, and make sure that your first line of output matches what you actually intended to submit.

Thinking through your design

Module design

You are required to keep the code that implements the game logic entirely separate from the code that implements your console-mode user interface. To that end, *you will be required* to submit at least two modules: one that implements your game logic and another that implements your user interface. You're welcome to break these two modules up further if you find it beneficial, but the requirement is that you keep these two parts of your program — the logic and the user interface — separate.

Note that this requirement is motivated partly by a desire to build good design habits, but also by the practical reality that maintaining that separation properly will give you a much better chance of being able to reuse your game logic, as-is and without modification, in the next project, when you'll be asked to build a graphical user interface for your game. In a big-picture sense, you can think of the user interface in this project as being a "throwaway"; while we'll be using it to grade your game logic, the true goal here is the complete version of Othello with a graphical user interface. So keeping this "throwaway" code completely separate from your game logic means that it will be easy to leave it out of your completed version, without causing anything else to break.

Module naming

Exactly one of your modules must be executable (i.e., should contain an **if __name__ == '__main__':** block), namely the one that you would execute if you wanted to launch your console user interface and play your game. No other modules are permitted to have an **if __name__ == '__main__':** block. (This is so our test automation tools can automatically determine which of your modules to execute.)

Using classes and exceptions to implement your game logic

Your game logic must consist of at least one class whose objects represent the current "state" of an Othello game, with methods that manipulate that state; you can feel free to implement additional classes, if you'd like. Note that this is in stark contrast to the approach used in **connectfour.py** in Project #2, where we used a namedtuple and a set of functions that returned new states. Classes offer us the ability to mix data together with the operations that safely manipulate that data; they allow us to create kinds of objects that don't just know how to *store* things, but also to *do* things.

Some of the methods I found useful in my own implementation of the Othello game state are listed below; this is not an exhaustive list, and you'll probably find a need for additional methods beyond these.

- Get the number of rows and/or columns on the board.
- Find out whose turn it is.
- Determine whether the game is over.
- Determine whether a disc is in some cell in the grid; if so, determine its color.
- Make a move.

Even if your console user interface does error checking, your game logic must not assume the presence of a particular user interface, so it must check any parameters it's given and raise an exception if the parameters are problematic (e.g., a non-existent row or column, an attempt to make an invalid move, an attempt to make a move after the game is over). Create your own exception class(es) to represent these error conditions.

Testing

One issue that comes up in the implementation of a program like this one is that it's difficult to test some of the corner cases that come up in the game logic by playing your game using your console interface. It can be difficult to duplicate games that end in a tie, situations where the turn skips back to the player who just moved, situations where the game ends with empty cells still on the board, and so on. And yet you need to be sure that these issues, and others like them, are handled correctly by your game logic.

The best way to handle problems like this is to test interesting, small-picture scenarios separately. One way to do that is to load your game logic into the Python interpreter and type Python expressions and statements into the interpreter manually to verify that the behavior is as you expect; if you aren't doing that already, you're missing out on one of the more valuable tools Python offers for testing and understanding the programs you write.

Another approach is to figure out some interesting scenarios and write program input that covers these scenarios, saving each one into a file using your favorite text editor. You can then copy and paste these into your program to test and re-test interesting cases as you work.

Thinking about the future in addition to the present

The next project will revisit the Othello game that you're building here, but will ask you instead to build a graphical user interface for your game using the **tkinter** library. We'll be talking a lot about **tkinter** and event-based programming in lecture; as we learn more about it, be sure you consider how your design for this project, particularly your game logic, can be done in a way that allows you to reuse code in the subsequent project, as you will not want to have to start over from scratch. This means you'll need to be cognizant of how you can separate code that handles console input and output from code that implements underlying game logic. It also means you'll want to start thinking about how your graphical user interface might need to use your game logic, as we learn more about graphical user interfaces and event-based programming in lecture over the next week or so.

A word about the use of outside resources

I am aware that there are existing versions of Othello written in Python that are available online. It should go without saying that you are not permitted to download these and submit them as your own, in whole or in part, and that you are not permitted to use them as any kind of basis for your own work, but prior experience has taught me otherwise. In a recent quarter, nearly 10% of my students submitted other people's work, mostly taken from online sources. I generally like to keep a pretty open policy about outside resources when it's pedagogically wise, but other implementations of Othello are strictly off-limits in your work on this project. Don't look for them, don't download them, and don't refer to them; the goal here is to write this code on your own. Be aware that a variety of existing implementations found online will be included in the plagiarism detection that we do after your work is submitted, something that has been very effective in finding and penalizing this kind of thing in the past.

Deliverables

Put your name and student ID in a comment at the top of each of your **.py** files, then submit all of the files to Checkmate. Take a moment to be sure that you've submitted all of your files.

Follow this link for a discussion of how to submit your project via Checkmate. Be aware that I'll be holding you to all of the rules specified in that document, including the one that says that you're responsible for submitting the version of the project that you want graded. We won't regrade a project simply because you submitted the wrong version accidentally.

Can I submit after the deadline?

Yes, it is possible, subject to the late work policy for this course, which is described in the section titled *Late work* at this link.

Simplified Othello rules incorporated by Alex Thornton, Fall 2015.
User interface requirements tightened up to allow test automation by Alex Thornton, Spring 2015.
Originally written by Alex Thornton, Winter 2013, with some influence from *Games Without Frontiers* and *Black and White*, also written by Alex Thornton.