ICS 32 Winter 2014
Project #4: *The Width of a Circle (Part 1)*

Due date and time: *Monday, March 3, 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". The second 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 square 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 build a console-mode, two-player version of Othello that is capable of playing a single game of Othello on a single computer. There are a handful of options that allow users to decide how the game will be played before it begins. The program begins by asking the users to choose these options; the game is then played by asking users to enter their moves directly into the console, continuing until the game is complete, at which point the program ends.

A detailed look at how your program should behave

Users will interact with your program in the following way.

- The program asks the user to specify the following options, in the order specified below. As with previous projects, when users enter erroneous input (e.g., a number of rows with a fractional part, an invalid move), the program should inform the user that the input was erroneous (and how) and ask the user for the input again.
    - The number of rows on the board, which must be an *even integer* between 4 and 16.
    - The number of columns on the board, which must be an *even integer* between 4 and 16 and *does not* have to be the same as the number of rows.
    - Which of the players will move first: black or white. (Generally, the black player moves first in Othello, though we'll allow the user to specify that the white player should move first if preferred.)
    - 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. The user can choose which color disc will be in the top-left position of these four center cells: white (the traditional default) or black.
    - Finally, the user can select what it means to win the game. There are two choices:

        - The player with the most discs on the board at the end of the game is the winner.
        - The player with the fewest discs on the board at the end of the game is the winner, which makes for an interesting and different flavor of the game.
- Until the game is over, the following sequence of events is repeated.
    - Display the score (i.e., how many discs of each color are on the board), the board, and whose turn (black or white) it is to move.
    - The user is asked to make a move, by specifying a cell on the board in which he or she would like to place a disc. It's up to you how a user selects a cell, though you should make it clear in your user interface what the user is required to do.
        - If the move is invalid (see the rules if you're curious what makes a move invalid), inform the user and ask for another one.
    - The move is made, discs are placed and flipped, and we're done with this move, proceeding to the next one.
- When the game is over, display the score, the board, and who the winner is. Be sure to handle the case when there is no winner (i.e., the number of discs on the board for both players is equal at the end of the game).
- At this point, your program can end. If you prefer, you can begin again (by asking the user to specify options again, etc.), but if you do, give the user the opportunity to exit the program if another game isn't desired.

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.
- Occasionally, neither player will have a valid move on the board, even though there are still empty cells in the grid. In this case, the game immediately ends and the winner is determined based on the number of discs each player has on the board.

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 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.

At least one of your modules should be executable (i.e., should contain an **if __name__ == '__main__':** block), namely the one that you would execute if you wanted to launch your user interface and play your game.

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 should 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.

A better approach, however, is to write a *test module*, a separate, executable module that contains code that **assert**s the behavior you want to verify. If you're going to have to type tests into the interpreter anyway, you might as well write them in a module instead, so you can run them repeatedly. As the number of tests you've written begins to increase, there's increasing value in re-running your tests; every time you make a change to your program, you could re-run your test module and see whether something that was working previously is now broken. This is a powerful technique indeed.

In order to do this kind of testing, whether you run your tests manually in the interpreter or write them more explicitly in a test module, you may find yourself needing to include a few functions in your modules and/or methods in your classes that you might not otherwise need, like being able to manually manipulate the cells on the game board without making moves, so you can set up a particular board scenario without having to figure out a sequence of moves that generates it. If the idea of writing code you don't appear to need seems wasteful, consider the flip side of the argument:

spending a little bit of time implementing a few additional functions might very well save you a lot more time in debugging your program further down the line. Automating testing, something we'll talk more about later this quarter, is well worth the effort; in my experience, I quite often save a lot more time than I spend in writing tests.

For this project, I would focus this kind of testing on your game logic, as opposed to your user interface. It's a great way to ensure that your user interface is being built on a solid foundation — and even allows you to write your game logic without writing your user interface initially — and will almost surely reduce the amount of time you spend debugging your program. If you do decide to write tests with **assert**s, be sure to write them in a separate module (or modules); don't mix the **assert**s into your program.

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 interface with 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. 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. 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.

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 greaded. 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.

Originally w ritten by Alex Thornton, Winter 2013, w ith some influence from *Games Without Frontiers* and *Black and White*, also w ritten by Alex Thornton.