Project 10: Nine Men's Morris

Overview:

This project is worth 55 points (5.5% of your course grade). It requires knowledge of both classes and exceptions, and is due by TUESDAY November 28 at 11:59PM. This project will take time; I recommend starting it early.

Nine Men's Morris is a mill-forming board game for two people that plays like a cross between Tic-Tac-Toe, Checkers, and Go. We will be writing a program that allows for the game to be played.

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 through Mimir before the project deadline.

Understanding the Game:

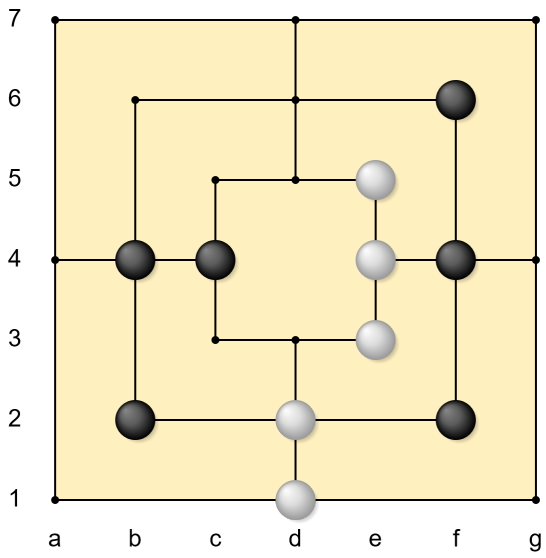
The best way to understand any game is to play it. You can play Nine Men's Morris online here:

http://www.smartlittlegames.com/ninemensmorris

We suggest that you play around with this game for a little while so that you can understand what it is you need to implement. The game is fairly straightforward to play, however, there are a few subtle rules, so be sure to pay attention to this project specification and the example above.

The Board:

Observe the image below from an in-progress game:



Each of the 24 intersections on the grid is called a point. The two players get 9 pieces each to place on these points. The goal of the game is to reduce your opponent to less than 3 pieces remaining by forming mills (three pieces in a row). If a player forms a mill, they get to remove one of their opponent's pieces from play—only one opponent's piece can be removed during a turn, even if placing a piece creates more than one mill. A removed piece is removed from the game and will never be played again.

Phase 1: Placing pieces

The game begins with an empty board. The players determine who plays first, then take turns placing their pieces one per play on empty points. If a player is able to place three of his pieces on contiguous points in a straight line, vertically or horizontally (i.e., not around a corner), he has formed a mill and may remove one of his opponent's pieces from the board and the game, with the caveat that a piece in an opponent's mill can only be removed if no other pieces are available. After all men have been placed, phase two begins.

Phase 2: Moving pieces

Players continue to alternate moves, this time moving a piece to an adjacent point. A piece may not "jump" another piece. Players continue to try to form mills and remove their opponent's pieces as in phase one. A player can "break" a mill by moving one of his pieces out of an existing mill, then moving it back to form the same mill a second time (or any number of times), each time removing one of his opponent's men.

Victory:

The game is won when a player reduces their opponent to fewer than 3 pieces in play, as they can no longer form mills and progress to a win.

Detailed Program Specifications:

You will develop a program that allows the user to play Nine Men's Morris according to the rules outlined above. The program will use the instructor supplied NMM.py module to model the board and assist with adjacency and mills. To help clarify the specifications, we provide sample output of a correctly implemented Nine Men's Morris python program below in the Sample Output section. We also provide a proj10.py file that contains stubs of the required functions. The program will run as is, but will not do anything useful. You are encouraged to create additional functions as you see fit. Your program must use the `import` statement for NMM.py; you are not allowed to copy the contents of NMM.py into your proj10.py implementation.

Note that you are **not** developing the Artificial Intelligence for the computer to play the game as your opponent. Instead you are creating a program that enforces the rules so that two human players can play the game.

Program commands:

The program will recognize the following commands regardless of case (i.e. a1, A1, H, and h are all valid). When specifying a point, the first character is a letter, the second is a digit.

xx	Place piece at point xx (only valid during the placing phase)
xx yy	Move piece from point xx to point yy
R	Restart the game from the beginning
H	Display this menu of commands
Q	Quit the game

Here, xx and yy denote valid points on the board in the form <letter><number> (e.g. a1 or d5).

The program will repeatedly display the current state of the game and prompt the user to enter a command, alternating players until someone wins the game or enters q, whichever comes first.

The program will detect, report, and recover from invalid commands. None of the data structures representing the board will be altered by an invalid command (and only the points member variable should ever be modified).

NMM.py Description:

The file NMM.py contains the class Board. This class is an example that does not have private attributes (the next project will have a class with private attributes). It has two constants that will be useful: ADJACENCY and MILLS. If you create an instance of Board named `board`, you refer to a constant as `board.MILLS` There are also two methods for manipulating the state named `assign_piece` and `clear_place`. They can be called as `board.clear_place(argument)`. Finally, there is a `__str__` method so you can print a board, e.g. `print(board)`.

Function Descriptions:

We have already supplied stubs for functions that we require (along with a large portion of the main function), but your implementation can include more functions if desired.

1. `count_mills(board, player) -> count`

`count_mills` takes in the current state of the board and one player, counts how many of the mills are held by the player, and returns the count. This can be used to determine if a player has formed a new mill by calling it before and after a placement is done.

3. `place_piece_and_remove_opponents(board, player, destination) -> None`

This function is used place a piece for "player" and if a mill is created, it removes an opponent's piece (by calling remove_piece specified below). It takes three parameters: board, player, and destination. It needs to raise a RuntimeError, if a placement is invalid (cannot place a piece in an occupied spot). If the placement is valid, it needs to place the piece at the destination (calling the board assign_piece method in the Board class) and if a new mill is created, remove an opponent's piece (calling the remove_piece function specified below). Mills created can be determined by calling count_mills before and after the move.

4. `move_piece(board, player, origin, destination) -> None`

This function is used to move a piece. It takes in board, player, origin, and destination. `It` needs to raise a RuntimeError if a movement is invalid. If a movement is valid, it needs to check adjacency (if necessary), remove the player's piece from the origin point, and call the appropriate functions to place the piece for player at the destination point and remove any opponent pieces if new mills were formed (by calling `place_piece_and_remove_opponents`).

5. `points_not_in_mills(board, player) -> collection_of_points`

This function will find all points belonging to player that are not in mills, and return them as an iterable data structure of your choice—a list or a set. This is used by the `remove_piece` function. Use the list "mills" in the Board class. Be careful because one piece can be part of horizontal mill, but not in a complete mill that is vertical. (I found this small function to be the most difficult to get correct.)

6. `placed(board,player) -> collection_of_points`

Return points where player's pieces have been placed. As with the previous function, return a set or a list, your choice.

7. `remove_piece(board, player) -> None`

This function will remove a piece belonging to player from board. It needs to determine which points are valid to remove (functions `points_not_in_mills` and `placed` are helpful), loop and get input until a valid piece is removed, and handle the removal of the piece from the board (by calling the board clear_place method in the Board class).

8. `is_winner(board, player) -> Boolean`

This function will be used to decide if a game was won. A game has been won if the opposing player has been reduced to fewer than three pieces.

9. `get_other_player(player) -> other_player`

This function is already implemented for you. It gets the other player and returns it. You will want to use this function any time you need to use the other player (change of turn, removal of pieces, etc.).

10. `main() -> None`

We have provided most of the framework necessary for the main function. I would advise against removing anything provided here. You only need to fill in the logic necessary where the pass statements currently are. You should raise RuntimeErrors for invalid commands and call the appropriate functions for valid commands.

Some Mandatory Stuff:

1. You must use the try-except block to handle errors (the only try-except block necessary is included in the sample, but it would not be incorrect to use one in remove_piece), each error message must explain exactly what type of errors a user is making. There are many errors to consider and we will not list them for you. You should be able to think of them when analyzing gameplay.

2. You should use the RuntimeError's custom message option to throw and handle errors for all errors. The error message is placed in quotes and when the exception is caught in the main your message in quotes is passed as error_message in the except block and is then printed.

Example:

```
if everything_looks_fine:
    do_whatever_necessary()
else:
    raise RuntimeError("Error: invalid command because of …")
```

Assignment Notes and Tips:

1. Before you begin to write any code, play with the game at the link provided above and look over the sample interaction below to be sure you understand the rules of the game and how you will simulate the game.

2. We provide a module called NMM.py that contains a Board class. Your program must use this module (by calling `import NMM`). You should understand each line of this file and should not modify it (it's not very long. Just look through it to understand what information you have access to).

3. We have provided a framework named proj10.py to get you started. Using this framework is mandatory. It runs as is, but does not do anything useful. Gradually replace the "stub" code (marked with pass and comments) with your own code. Modify and test functions one at a time—do not move on to a new function until existing functions are correct.

4. It might go without saying, but start with just getting placement working correctly before you move on to movement.

7. The coding standard for CSE 231 is posted on the course website: https://www.cse.msu.edu/~cse231/Online/General/coding.standard.html. Items 1-9 of the Coding Standard will be enforced for this project.

8. Your program should not use any global variables inside of functions. That is, all variables used in a function body must belong to the function's local name space. The only global references will be to functions and constants.

9. Your program must contain the functions listed in the previous sections. You may develop additional functions as you feel is appropriate.

Sample Output:

Because the interactions are lengthy, three have been saved in separate files named output?.txt (where ? represents a digit 1, 2, or 3) available in the project folder. Each output file corresponds to an input?.txt file.

Grading Rubrics

```
General Requirements:
  __0__   (5 pts) Coding Standard 1-9
Implementation:
  __0__ (4 pts) count_mills function
  __0__ (3 pts) placed function
  __0__ (4 pts) is_winner function
  __0__ (4 pts) points_not_in_mills function
  __0__ (14 pts) Pass Test1
  __0__ (8 pts) Pass Test2
  __0__ (13 pts) Pass Test3
```