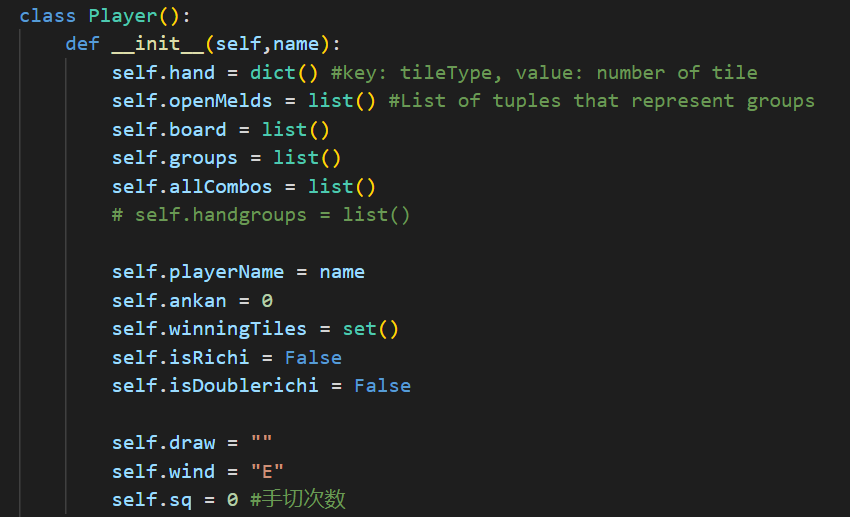
How the program is supposed to be used:

I play a Mahjong game, and I also run this program alongside the game. While I am playing, I also input every action in the game, whether it’s draw&discard or chi/pon/kan. At any time, I can enter analysis mode and the program can help me to make decisions.

What each file is doing:



Simulate each player. Note that the player representing the user and the opponents are all Player class, but when the game simulates, it only checks tile availability/drawn tile of the user because only the user’s hand should be accessible.

**self.Wind**

Key Properties:



**openMelds**

**self.Hand**

**self.Board**

Hand: a dictionary containing the hand {‘p5’:2,’p6’:2,’p7’:2, ...}

OpenMelds: the displayed tiles when players do chi/pon/kan

FullHand: get Hand+OpenMelds to determine some yakus and calculate score

Key Functions:

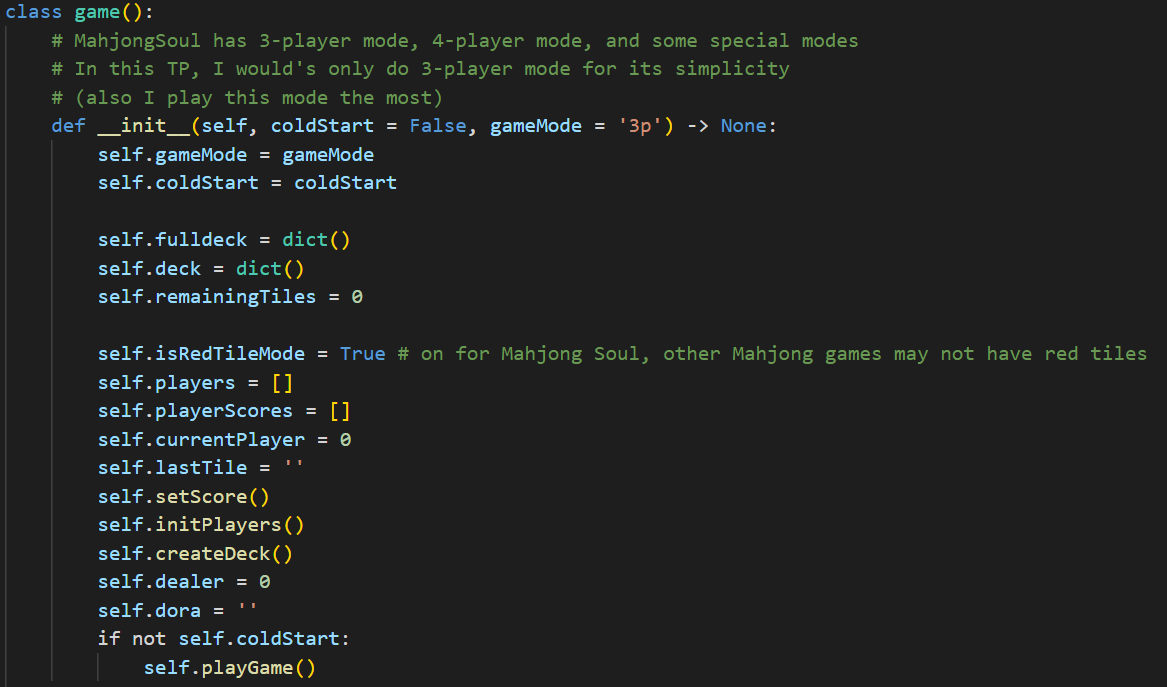
getTileGroup(tile): get the tiles that can form a group with the tile given. For example, for s2, it outputs[(s2,s2),(s1,s3),(s3,s4)] because three s2 make up a triplet, and (s1,s3) or (s3,s4) make up a sequence.

SearchGroup(hand): pick a pair out of the hand, then dfsSearchGroup(). Kind of wrapper function for dfsSearchGroup

dfsSearchGroup(hand): the hand size should be a factor of 3, and this function try to form groups. For each kind of tile in hand, the program try to grab tiles from getTileGroup(tile). If those tiles are available, grab the entire group from the hand and call itself while passing the group. Whether those tile are available or not, the hand revert to the previous status, because **there could be more than one way to form groups.** This is why the search don’t return at the end but only record a legal search for later purpose. For example, for a hand like s2 s2 s3 s3 s4 s4 s5 s5, it can be either (s2,s2),(s3,s4,s5)(s3,s4,s5) or (s2,s3,s4),(s2,s3,s4),(s5,s5).

Some Note on Algorithm Complexity: At this point, SearchGroup() and dfsSearchGroup() is already on par with previous Mahjong TPs, and this would be how those TPs check win condition. But my program goes on to check other rules under “Richi” rule(all previous Mahjong TPs are under a simpler rule), and the program also calculate the score with score.py. And this entire process should be more complex.

checkWin(): this is abandoned because I move this to isWin() in game class. This is for testing purpose.



In game.py, this simulate an entire game. The game class should reflect everything in the actual Mahjong game, and the class can perform all actions in the actual game.

Key Functions:

test(): to manually input everything on the board, for testing purpose only

getStart(): enter the initial stats if the game is not in test mode

playGame(): simulate the game process, each play take turns

isWin(): does the player has a basic winning pattern

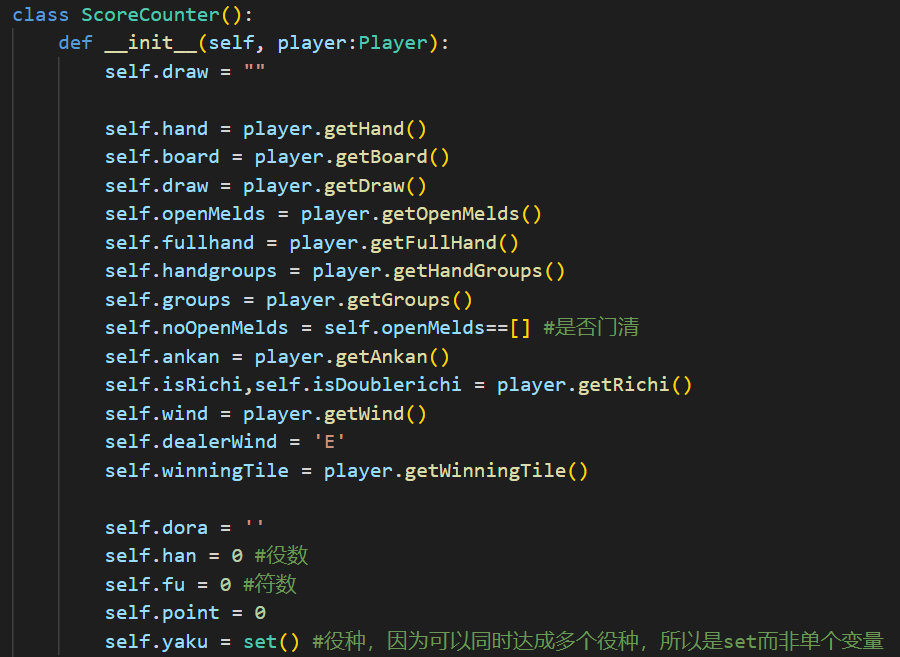
isTenpai(): if given a desired tile, can isWin() be true?

getWinningTiles(): if isTenpai(), get all desired tiles

getTenpai(): get every tile that can be discarded in order to reach isTenpai()

doAction(): simulate an action, if it’s an analysis function, doAnalyze

doAnalyze(): automatically output a brief analysis, then await user input to either do specific analysis or quit the analysis mode, this would return to doAction since doing analysis isn’t an actual move in a game.



ScoreCounter takes a player and get most properties from the player class. Then, go through each yaku(“wining pattern”) and then calculate the score. Because of the ‘Richi’ rule, some yakus check the fullHand, some only check the openMelds. Note that when using ScoreCounter, it is assumed that the basic pattern is already formed( 4xABC/AAA + DD)