A Home moderator for Othello

AI – Gabor – Dec, 2021

The below code, properly configured, will use your own Othello script to play a game against Random where you pass in a token on the command line to indicate whether you wish your code to play as ‘x’ or ‘o’. Here is the code:

```
import sys; args = sys.argv[1:]
import random
from moveFastD import findMoves, makeMove, \
     getDefaultMove as findBestMove


def show2D(brd, tkn, mv, findMovesFunc):
  # Display a snapshot:
  # Move played, 2D board, 1D board w. score, psbl moves
  # brd is a string, tkn just moved to mv
  next = 'xo'[tkn=="x"]
  psblMv = findMovesFunc(brd, next)      # Possible moves
  if not psblMv:         # If one side must pass
    psblMv = findMovesFunc(brd, (next:=tkn))
  brdL = [*brd]          # Listify brd to show asterisks
  for m in psblMv: brdL[m] = "*"
  brdL[mv] = brdL[mv].upper()            # Show most recent move
  b2 = "".join(brdL)
  print(f"'{tkn}' played to {mv}")
  print("\n".join([b2[rs:rs+8] for rs in range(0,len(b2),8)]))
  print(f"\n{brd} {brd.count('x')}/{brd.count('o')}")
  if psblMv:             # If game not over, show possible moves
    print(f"Possible moves for '{next}': {sorted([*psblMv])}\n")
```

```
def playGame(findBestMove, findMoves, makeMove, token):
  # plays a game between findBestMove and Random
  # findMove(brd, tkn)
  # findBestMove(brd, tkn, psblMoves)
  # makeMove(brd, tkn, mv, psblMoves)
  # Csaba Gabor, 10 Dec 2021

  brd = '.'*27+'ox......xo'+'.'*27      # Starting board
  tknToPlay = 'x'
  transcript = []                       # Transcript of the game

  while True:
    if not (moves:=findMoves(brd, tknToPlay)):
      tknToPlay = 'xo'[tknToPlay=='x']     # Swap players if pass
      if not (moves:=findMoves(brd, tknToPlay)): break
      transcript.append(-1)             # Note the pass
    if tknToPlay != token:              # if it's Random's turn:
      transcript.append(random.choice([*moves]))
      brd = makeMove(brd, tknToPlay, transcript[-1], moves)
      show2D(brd, tknToPlay, transcript[-1], findMoves)
    else:                               # else it's Our turn
      transcript.append(findBestMove(brd, tknToPlay, moves))
      brd = makeMove(brd, tknToPlay, transcript[-1], moves)
      show2D(brd, tknToPlay, transcript[-1], findMoves)
    brd = brd.lower()                   # Just in case
    tknToPlay = 'xo'[tknToPlay=='x']    # Switch to other side

  # Game is over:
  tknCt = brd.count(token)
  enemy = len(brd) - tknCt - brd.count('.')
  print(f"\nScore: Me as {token=}: {tknCt} vs Enemy: {enemy}\n")
  xscript = [f"_{mv}"[-2:] for mv in transcript]
  print(f"Game transcript: {''.join(xscript)}")

playGame(findBestMove, findMoves, makeMove, args[0])
```

I named my script MiniMod.py. There are a few assumptions, which you will have to customize for your own code. First, the part of your Othello script that runs in response to input (even if there is no input), should be put in to a routine named main(). That is, you should have a

```
def main():
  …
```

Below that, you should have a line that says:

```
if __name__ == '__main__': main()
```

This odd looking code is so that the bulk of your code does not run when it is imported, but rather, only the functions should be defined. The above line differentiates between the script being called directly vs. being imported.

Now, there is a second consideration that applies. If you are defining globals (such as lookup tables), then that code should run. So if you have a call such as `setGlobals()` in your code, do not include that within `main()`, but rather prior to it, such as:

```
setGlobals()
if __name__ == '__main__': main()
```

Now, MiniMod.py assumes that you have three functions defined:

`findMoves(brd, tkn)`	: returns a psblMoves structure
`makeMove(brd, tkn, mv, psblMoves)`	: returns resultant brd as a string
`findBestMove(brd, tkn, mv, psblMoves)`	: returns an integer (move position)

Your code does not need to have these same function names, nor even the same arguments. For example, your makeMove and findBestMove may not take a psblMoves structure. That is OK. You will reflect the relevant arguments and update the code in MiniMod.py by following the purple.

The first thing to do is to import the 3 functions corresponding to the ones above – see the 3rd line of code starting `from MoveFastD import …` This lets python know that MiniMod.py wants to use the three relevant functions for its own purposes. Of course, the first thing to do is to change `MoveFastD` to the name of your Othello script. Now, if you already named the three functions as above, then you are good to go. In the example, within MoveFastD.py, `findBestMove()` is not defined, but `getDefaultMove()` is defined, so rather than renaming it, you let python know to treat `getDefaultMove()` as if it were named `findBestMove()` by appending `as findBestMove` to the import statement.

Now, the next thing to do is to fix up the arguments to the three functions involved. If you happen to have exactly the same arguments, then you are good to go, but chances are that you will have to make a little alteration. For example, your `makeMove()` and `findBestMove()` may not be interested in making use of the possible moves that `findMoves()` found. That is OK – just update the arguments and relevant calls (look to the purple in the `while True` loop within `playGame()`). In the scenario just outlined, you would eliminate the last call to `makeMove()` and `findBestMove()`.

At this point, you should be ready to run your code from the command line using:
`python MiniMod.py x` or `python MiniMod.py o`

If your script is used to printing out debugging information, it will continue to do so over the course of the game. You could potentially alter the way this behaviour works by setting a flag prior to `main()` that causes it not to print, but then override the flag in `main()` to print if the script is directly invoked.

There is one more thing to fix up in order to make this scheme even more useful. What you will get as a result of running the code is a series of snapshots showing you the progress of the game until the game terminates. As you review the game, you may encounter some moves where you may be dissatisfied with what your Othello script is doing. In that case, you could copy / paste in the relevant board to your script to examine what it does.

However, there is another way to go, and that is that at the end of the game, a condensed transcript of the game is printed. It is condensed because it is already so long – why tote around the extra spaces separating the arguments? Therefore, each move is encoded as exactly two characters. If the move is given by a single digit, then it is prefixed by an underscore. Here's an example condensed transcript. Notice that in addition to the underscores, there is also a few -1s present. These indicate a pass.

```
3743182150462919451726384454l1_463625520615147493925223052144260565 3_7
_65731_533_3_2_1345948401032_9_0_85816241241-113-12315
```

In order to use this, your script should augment the arguments that it can handle. One standard way to do this is to loop through each argument and decide what it is meant to do. For example:

```
brd, tkn, moves = ('.'*27) + 'ox...... xo' + ('.'*27), "", []
for arg in args:
  if len(arg)<2: tkn = arg
  elif: len(arg)==64 and not {*arg} - {*"xXoO."}: brd = arg
  elif: len(arg)==2: moves += int(arg)
  elif: len(arg)>2 and not {*arg} - {*"0123456789_-"}:
    moves += [int(arg[k:k+2].replace("_", ""))
                for k in range(0,len(arg),2)]
```

Remember to ignore negative moves!