Assignment 1

● Graded

Student

Longpeng Xu

View or edit group

Total Points

8.75 / 10 pts

Autograder Score

6.0 / 6.0

Passed Tests

TestDesign (max_score=0) (0/0)
TestNumHours (max_score=0) (0/0)
TestGenerateInitialPieces (max_score=0.5) (0.5/0.5)
TestInitialState (max_score=0.5) (0.5/0.5)
TestPlacePiece (max_score=0.25) (0.25/0.25)
TestPlacePieceExtra (max_score=0.25) (0.25/0.25)
TestPrintGame (max_score=0.5) (0.5/0.5)
TestPrintGameExtra (max_score=0.5) (0.5/0.5)
TestProcessMove (max_score=0.25) (0.25/0.25)
TestProcessMoveExtra (max_score=0.25) (0.25/0.25)
TestGetPlayerMove (max_score=0.5) (0.5/0.5)
TestCheckMove (max_score=0.25) (0.25/0.25)
TestCheckWin (max_score=0.375) (0.375/0.375)
TestCheckWinExtra (max_score=0.375) (0.375/0.375)
TestCheckStalemate (max_score=0.5) (0.5/0.5)
TestMain (max_score=0.5) (0.5/0.5)
TestMainExtra (max_score=0.5) (0.5/0.5)
TestMain7030 (max_score=0.5) (0.5/0.5)
TestMain7030Extra (max_score=0.5) (0.5/0.5)

Question 2

Readability **1** / 1.25 pts

2.1 **Program Structure** **0.75** / 0.75 pts

✔ **+ 0.75 pts** All of the following criteria has been met:

- Vertical whitespace has been used appropriately to separate logical blocks of code.
- Horizontal whitespace has been used to avoid terse lines of code.
- There are no sections of code which create undue burden on the reader.
- All lines of code conform to PEP8 style rules such as a maximum line length of 80 characters.

+ 0.5 pts Most of the following criteria has been met:

- Vertical whitespace has been used appropriately to separate logical blocks of code.
- Horizontal whitespace has been used to avoid terse lines of code.
- There are no sections of code which create undue burden on the reader.
- All lines of code conform to PEP8 style rules such as a maximum line length of 80 characters.

+ 0 pts At least one of the following criteria has been majorly violated:

- Vertical whitespace has been used appropriately to separate logical blocks of code.
- Horizontal whitespace has been used to avoid terse lines of code.
- There are no sections of code which create undue burden on the reader.
- All lines of code conform to PEP8 style rules such as a maximum line length of 80 characters.

2.2 **Identifier Names** **0.25** / 0.5 pts

+ 0.5 pts All of the following criteria has been met:

- All identifier names conform to the correct casing for python code.
- All non-counter variables have a meaningful name which describes the variable independent of its context.
- Variable types are not included in the name when the type does not describe the variable.
- Constants are used where obviously applicable (e.g. HELP_TEXT)

✔ **+ 0.25 pts** Most of the following criteria has been met:

- All identifier names conform to the correct casing for python code.
- All non-counter variables have a meaningful name which describes the variable independent of its context.
- Variable types are not included in the name when the type does not describe the variable.
- Constants are used where obviously applicable (e.g. HELP_TEXT)

+ 0 pts At least one of the following criteria has been majorly violated.

- All identifier names conform to the correct casing for python code.
- All non-counter variables have a meaningful name which describes the variable independent of its context.
- Variable types are not included in the name when the type does not describe the variable.
- Constants are used where obviously applicable (e.g. HELP_TEXT)

Question 3

Algorithmic Logic **1.25** / 1.25 pts

3.1 **Single Instance of Logic** **0.5** / 0.5 pts

✔ **+ 0.5 pts** Almost no code has been duplicated in your program. You have well designed functions with appropriate parameters to modularise your code.

+ 0.25 pts Some code has been duplicated in your program. You have used some functions to modularise your code.

+ 0 pts Large amounts of code are duplicated in your program. You have made poor use of functions to modularise your code.

3.2 **Variable Scope** **0.25** / 0.25 pts

✔ **+ 0.25 pts** Variables are declared locally in the functions in which they are needed. Global variables have not been used.

+ 0 pts Global variables have been used, reducing the clarify of function logic.

3.3 **Control Structures** **0.5** / 0.5 pts

✔ **+ 0.5 pts** Logic is structured simply and clearly through good use of control structures.

+ 0.25 pts A small number of control structures are unnecessarily complex.

+ 0 pts Many control structures are poorly designed (e.g. excessive nesting, overly complex conditional logic, loops with multiple unnecessary exit points, ...).

Question 4

Documentation **0.5** / 1.5 pts

4.1 **In-line Comment Clarity** **0** / 0.75 pts

+ 0.75 pts Inline comments are used to assist readability in all of the following cases:

- Single lines with complex logic.
- Blocks of code with a singular purpose which should be documented.

Almost all comments enhance the comprehensibility of the code. Comments almost never repeat information already apparent in the code.

+ 0.5 pts Inline comments are used to assist readability in most of the following cases:

- Single lines with complex logic.
- Blocks of code with a singular purpose which should be documented.

Or a few comments are unnecessary to the comprehension of the code. Alternatively, a few comments are overly verbose reducing the ease with which code can be comprehended.

✔ **+ 0 pts** More than one case of missing inline comments in the following situations:

- Single lines with complex logic.
- Blocks of code with a singular purpose which should be documented.

Or many comments are unnecessary to the comprehension of the code. Alternatively, many comments are overly verbose reducing the ease with which code can be comprehended.

Inline comments are verbose, they include individual steps and look more like notes on how to write the code (this is fine while writing the code, but should be removed for final solution). The point of inline comments should be to make code more readable.

4.2 **Informative Docstrings** **0.5** / 0.75 pts

+ 0.75 pts All modules, classes, methods and functions are clearly and concisely described via informative and complete docstrings. Type information is always accurately communicated in docstrings or via type hints.

✔ **+ 0.5 pts** Most modules, classes, methods and functions are clearly and concisely described via informative and complete docstrings. Type information is almost always accurately communicated in docstrings or via type hints.

+ 0 pts Several docstrings are inaccurate or unclear, or absent. Some parameters and return types are unclear.

Docstrings need brief function descriptions, and need to be properly formatted

Autograder Results

Functionality Test Results

TestDesign (max_score=0) (0/0)

1. test_clean_import (weight=1):
   PASSED
2. test_doc_strings (weight=1):
   PASSED
3. test_functions_defined_correctly (weight=1):
   PASSED

TestNumHours (max_score=0) (0/0)

1. test_num_hours_float (weight=1):
   PASSED

TestGenerateInitialPieces (max_score=0.5) (0.5/0.5)

1. test_generate_initial_pieces (weight=1):
   PASSED
2. test_generate_initial_pieces_example (weight=1):
   PASSED
3. test_generate_initial_pieces_6 (weight=1):
   PASSED
4. test_generate_initial_pieces_9 (weight=1):
   PASSED

TestInitialState (max_score=0.5) (0.5/0.5)

1. test_initial_state (weight=1):
   PASSED
2. test_different_lists (weight=1):
   PASSED

TestPlacePiece (max_score=0.25) (0.25/0.25)

1. test_place_piece_board (weight=1):
   PASSED
2. test_place_piece_pieces (weight=1):
   PASSED
3. test_place_piece (weight=1):
   PASSED
4. test_place_piece_2 (weight=1):
   PASSED
5. test_place_piece_alt (weight=1):
   PASSED

TestPlacePieceExtra (max_score=0.25) (0.25/0.25)

1. test_place_piece_extra_1 (weight=1):
   PASSED
2. test_place_piece_extra_2 (weight=1):
   PASSED

TestPrintGame (max_score=0.5) (0.5/0.5)

1. test_print_game_empty (weight=1):
   PASSED
2. test_print_game_one_piece (weight=1):
   PASSED
3. test_print_game_two_pieces (weight=1):
   PASSED
4. test_print_game_multi_pieces (weight=1):
   PASSED

TestPrintGameExtra (max_score=0.5) (0.5/0.5)

1. test_print_game (weight=1):
   PASSED
2. test_print_game_alt (weight=1):
   PASSED

TestProcessMove (max_score=0.25) (0.25/0.25)

1. test_process_move (weight=1):
   PASSED
2. test_process_move_alt (weight=1):
   PASSED
3. test_process_move_row (weight=1):
   PASSED
4. test_process_move_row_alt (weight=1):
   PASSED
5. test_process_move_col (weight=1):
   PASSED
6. test_process_move_piece (weight=1):
   PASSED
7. test_process_move_correct (weight=1):
   PASSED

TestProcessMoveExtra (max_score=0.25) (0.25/0.25)

1. test_process_move_correct (weight=1):
   PASSED
2. test_process_move_corret_alt (weight=1):
   PASSED
3. test_process_move_row (weight=1):
   PASSED
4. test_process_move_row_alt (weight=1):
   PASSED
5. test_process_move_col (weight=1):
   PASSED
6. test_process_move_invalid (weight=1):
   PASSED

TestGetPlayerMove (max_score=0.5) (0.5/0.5)

1. test_get_player_move (weight=1):
   PASSED
2. test_get_player_move_format (weight=1):
   PASSED
3. test_get_player_move_row (weight=1):
   PASSED
4. test_get_player_move_size (weight=1):
   PASSED
5. test_get_player_move_full (weight=1):
   PASSED

TestCheckMove (max_score=0.25) (0.25/0.25)

1. test_check_move_piece_invalid (weight=1):
   PASSED
2. test_check_move_piece_valid (weight=1):
   PASSED
3. test_check_move_piece_number_invalid (weight=1):
   PASSED
4. test_check_move_piece_number_valid (weight=1):
   PASSED

TestCheckWin (max_score=0.375) (0.375/0.375)

1. test_check_win_no_winner (weight=1):
   PASSED
2. test_check_win_naught_winner (weight=1):
   PASSED
3. test_check_win_cross_winner (weight=1):
   PASSED

TestCheckWinExtra (max_score=0.375) (0.375/0.375)

1. test_check_win_no_winner (weight=1):
   PASSED
2. test_check_win_no_winner_alt (weight=1):
   PASSED
3. test_check_win_cross_winner (weight=1):
   PASSED
4. test_check_win_naught_winner (weight=1):
   PASSED

TestCheckStalemate (max_score=0.5) (0.5/0.5)

1. test_check_stalemate (weight=1):
   PASSED
2. test_check_stalemate_not (weight=1):
   PASSED
3. test_check_stalemate_alt (weight=1):
   PASSED

TestMain (max_score=0.5) (0.5/0.5)

1. test_main (weight=1):
   PASSED
2. test_main_play (weight=1):
   PASSED
3. test_main_play_again (weight=1):
   PASSED
4. test_main_stalemate (weight=1):
   PASSED

TestMainExtra (max_score=0.5) (0.5/0.5)

1. test_main (weight=1):
   PASSED

TestMain7030 (max_score=0.5) (0.5/0.5)

1. test_main_grid_5 (weight=1):
   PASSED

TestMain7030Extra (max_score=0.5) (0.5/0.5)

1. test_main (weight=1):
   PASSED
2. test_main_alt (weight=1):
   PASSED

Submitted Files

a1.py 3 Comments Download

```
""" A fancy tic-tac-toe game for CSSE1001/7030 A1. """
from constants import *

Board = list[list[str]]
Pieces = list[int]
Move = tuple[int, int, int] # (row, column, piece size)


def num_hours() -> float:
    '''
    Inputs: No input
    Outputs
        Returns expected hours to finish a1 (type float)
    '''
    return 24.0


def generate_initial_pieces(num_pieces: int) -> Pieces:
    '''
    Inputs
        num_pieces: number of pieces (type int)
    Outputs
        Returns marker sizes from 1 up to and incl num_pieces (type Pieces)
    '''
    return list(range(1, num_pieces+1))


def initial_state() -> Board:
    '''
    Inputs: No input
    Outputs
        Returns a new board where every cell contains EMPTY (type Board)
    '''
    # List comprehension is used to avoid shallow copy
    return [[EMPTY] * GRID_SIZE for _ in range(GRID_SIZE)]


def place_piece(board: Board, player: str, pieces_available: Pieces,
                move: Move) -> None:
    '''
    Inputs
        board: A squared board of cells which may contain pieces (type Board)
        player: A player's name (type string)
        pieces_available: The player's avaiable peices (type Pieces)
        move: Place a piece of (size) on (row, column) on board (type Move)
    Outputs
```

```
47          Returns None
48      '''
49      row, col, size = move
50
51      if player in [NAUGHT, CROSS]:
52          # adding the piece to the position "move" on the board
53          board[row][col] = player + str(size)
54
55          # remove the piece from "pieces_available"
56          pieces_available.remove(size)
57
58      return None
59
60
61  def print_game(board: Board, naught_pieces: Pieces, cross_pieces: Pieces
62                 ) -> None:
63      '''
64      Inputs
65          board: A squared board of cells which may contain pieces (type Board)
66          naught_peices: The available peices of player NAUGHT (type Pieces)
67          cross_peices: The available peices of player CROSS (type Pieces)
68      Outputs
69          Prints something; Returns None
70      '''
71      # Display the players' pieces
72      O_pieces = [str(i) for i in naught_pieces]
73      X_pieces = [str(i) for i in cross_pieces]
```

Instructor | 09/04 at 11:53 pm

variable names should be be uppercase

```
74      print("O has:", ', '.join(O_pieces))
75      print("X has:", ', '.join(X_pieces))
76      print()
77
78      # Prepare board display by flattening a 2D board (type Board) to a 1D list
79      flattened = [item for sublist in board for item in sublist]
80      i = 1
81
82      # Display a well-formatted board which has (2 + 2*GRID_SIZE) printed rows
83      while i <= 2 + 2*GRID_SIZE:
84
85          # The first row displays the column indices
86          if i == 1:
87              i += 1
88              elements = [EMPTY] + [' ' + str(i) + ' ' for i in \
89                                    range(1, GRID_SIZE)]
```

```
            elements += [' ' + str(GRID_SIZE)]
            print(''.join(elements))

        # The even-indexed rows displays the "---"-formated row splitters
        elif i % 2 == 0:
            i += 1
            print(EMPTY + '-'*GRID_SIZE*3)

        # The odd-indexed rows displays the cells where Pieces may be in place
        else:
            i += 1
            row = int((i-1)/2)
            start, end = (row - 1) * GRID_SIZE, row * GRID_SIZE - 1
            print(f"{row}|", end='')
            for j in range(GRID_SIZE - 1):
                print(f"{flattened[start + j]}|", end='')
            print(f"{flattened[end]}|")

    return None


def process_move(move: str) -> Move | None:
    '''
    Inputs
        move: An instruction of a on-board move (type str)
    Outputs
        If move correctly formatted: Returns its (row, column, piece size)
        ... (type Move)
        Otherwise: Prints something; Returns None
    '''
    row, col, size = move[0], move[2], move[4]

    # identify invalid format by length and space/non-space characters
    if len(move) != 5 or not (all(i != ' ' for i in [row, col, size]) and \
                              move[1] == ' ' and move[3] == ' '):
        print(INVALID_FORMAT_MESSAGE)

    # identify invalid rows, columns, or sizes
    elif row not in [str(i) for i in range(1, GRID_SIZE + 1)]:
        print(INVALID_ROW_MESSAGE)
    elif col not in [str(i) for i in range(1, GRID_SIZE + 1)]:
        print(INVALID_COLUMN_MESSAGE)
    elif size not in [str(i) for i in range(1, PIECES_PER_PLAYER + 1)]:
        print(INVALID_SIZE_MESSAGE)

    # return Move of valid format after converting it from str to tuple
    else:
        row, col, size = int(row)-1, int(col)-1, int(size)
        return (row, col, size)
```

```
139
140
141 def get_player_move() -> Move:
142     '''
143     Inputs
144         No input as parameters
145         prompt: User's input indicating a move on-board (type int)
146     Outputs
147         If prompt is "h" or "H": Prints something and Returns function
148         ... get_player_move() (type Move)
149         If prompt is format-validated by function process_move(): Returns the
150         ... move (type Move)
151         Otherwise: Returns function get_player_move() (type Move)
152     '''
153     prompt = input("Enter your move: ")
154
155     if prompt in ["h", "H"]:
156         print(HELP_MESSAGE)
157         return get_player_move()
158     else:
159         # Returns correctly formatted move (type Move) or Prints error message
160         # ... for the incorrectly formatted
161         if process_move(prompt):
162             return process_move(prompt)
163         # Re-prompts user for a correctly formatted if current one is incorrect
164         else:
165             return get_player_move()
166
167
168 def check_move(board: Board, pieces_available: Pieces, move: Move) -> bool:
169     '''
170     Inputs
171         board: A squared board of cells which may contain pieces (type Board)
172         pieces_available: The player's avaiable peices (type Pieces)
173         move: Place a piece of (size) on (row, column) on board (type Move)
174     Outputs
175         Whether the move is valid (type bool)
176     '''
177     row, col, size = move
178
179     # A move is valid only when the size is available, and (row, column) is
180     # ... empty or containing a smaller piece
181     if size in pieces_available and (board[row][col] == EMPTY or
182                                      size > int(board[row][col][-1])):
183         return True
184     else:
185         return False
186
187
```

```
188 def check_win(board: Board) -> str | None:
189     '''
190     Inputs
191         board: A squared board of cells which may contain pieces (type Board)
192     Outputs
193         If there is a winner: Returns winner (type str)
194         Otherwise: Returns None
195     '''
196     # board with cells containing players only, its transpose, its diagnoal,
197     # ... and its reverse diagonal
198     who = [[cell[0] for cell in row] for row in board]
199     who_transpose = [[who[i][j] for i in range(GRID_SIZE)] for j in
```

Instructor | 09/04 at 11:53 pm

non meaningful variable name

```
200                  range(GRID_SIZE)]
201     diag = [who[i][i] for i in range(GRID_SIZE)]
202     rev_diag = [who[i][GRID_SIZE - 1 - i] for i in range(GRID_SIZE)]
203 
204     # Only in the following cases a winner exists
205     winner = None
206 
207     # Detects if a player occupies the diagonal
208     if all(j == diag[0] for j in diag) and diag[0] != ' ':
209         winner = diag[0]
210 
211     # Detects if a player occupies the reverse diagonal
212     elif all(k == rev_diag[0] for k in rev_diag) and rev_diag[0] != ' ':
213         winner = rev_diag[0]
214     else:
215         for i in range(GRID_SIZE):
216 
217             # Detects if a player occupies a row
218             if all(m == who[i][0] for m in who[i]) and who[i][0] != ' ':
219                 winner = who[i][0]
```

Instructor | 09/04 at 11:54 pm

who?

```
220 
221             # Detects if a player occupies a column
222             elif all(n == who_transpose[i][0] for n in who_transpose[i]) and \
223                  who_transpose[i][0] != ' ':
224                 winner = who_transpose[i][0]
225 
```

```
226     return winner
227 
228 
229 def check_stalemate(board: Board, naught_pieces: Pieces, cross_pieces: Pieces
230                     ) -> bool:
231     '''
232     Inputs
233         board: A squared board of cells which may contain pieces (type Board)
234         naught_peices: The available peices of player NAUGHT (type Pieces)
235         cross_peices: The available peices of player CROSS (type Pieces)
236     Outputs
237         Whether stalemate (no more moves can be made) is reached (type bool)
238     '''
239     # Extract the piece size of each cell on board
240     sizes = []
241     for i in range(GRID_SIZE):
242         for j in range(GRID_SIZE):
243             if board[i][j] == EMPTY:
244                 sizes += [0]
245             else:
246                 sizes += [int(board[i][j][-1])]
247 
248     # Scenarios where no more moves can be made
249     # 1. board unfull with min size (0) >= no available pieces (max size = -1)
250     # 2. board full with min size (+) >= no available pieces (max size = -1)
251     # 3. board full with min size (+) >= max available piece size (+)
252     max_naught_peices, max_cross_pieces = -1, -1
253     if naught_pieces:
254         max_naught_peices = max(naught_pieces)
255     if cross_pieces:
256         max_cross_pieces = max(cross_pieces)
257 
258     stalemate = False
259     # All the scenarios can be concluded by the same predicate
260     if min(sizes) >= max(max_naught_peices, max_cross_pieces):
261         stalemate = True
262 
263     return stalemate
264 
265 
266 def main() -> None:
267     '''
268     Inputs: No input
269     Outputs:
270         Returns whole_game() (type None)
271     '''
272     # Helper function covering Step 3, Step 4, Step 5-1
273     def test_move(board: Board, player: str, pieces_player: Pieces) -> None:
274         '''
```

```
275         Inputs
276             board: A squared board of cells which may contain pieces(type Board)
277             player: A player's name (type string)
278             pieces_player: Available pieces of the player (type Pieces)
279         Outputs
280             If move is correctly formatted and valid: Returns
281             ... place_piece(board, player, pieces_player, move) (type None)
282             If move is correctly formatted but invalid: Print something and
283             ... Returns test_move(board, player, pieces_player) (type None)
284             Otherwise: Returns test_move(...) (type None)
285         '''
286         # Step 3: The user is prompted for a move
287         move = get_player_move()
288 
289         # Step 4: Check if the move is correctly formatted and valid
290         if move and check_move(board, pieces_player, move):
291             # Step 5-1: Update the board
292             return place_piece(board, player, pieces_player, move)
293         elif move and not check_move(board, pieces_player, move):
294             print(INVALID_MOVE_MESSAGE)
295             print(f"\n{player} turn to move\n")
296             return test_move(board, player, pieces_player)
297         else:
298             return test_move(board, player, pieces_player)
299 
300     # Helper function covering Step 7
301     def game_over() -> None:
302         '''
303         Inputs: No input
304         Outputs
305             If agree to play again: Returns whole_game() (type: None)
306             Otherwise: Returns None
307         '''
308         # Step 7: Prompt them for whether to play again
309         prompt = input("Play again? ")
310         if prompt in ["y", "Y"]:
311             return whole_game()
312         else:
313             return None
314 
315     # Helper function covering all steps
316     def whole_game() -> None:
317         '''
318         Inputs: No input
319         Outputs: Returns None
320         '''
321         # Initialization of the game
322         board = initial_state()
323         naught_pieces = generate_initial_pieces(PIECES_PER_PLAYER)
```

```
        cross_pieces = generate_initial_pieces(PIECES_PER_PLAYER)
        players = ["O", "X"]
        i = 0

        # Step 1: The current game is displayed
        print_game(board, naught_pieces, cross_pieces)

        # Iterate over Step 2 to Step 6
        while check_stalemate(board, naught_pieces, cross_pieces) == False and \
              check_win(board) == None:
            player = players[i % 2]
            if player == "O":
                pieces_player = naught_pieces
            else:
                pieces_player = cross_pieces
            # Step 2: The user is informed whose turn it is to move.
            print(f"\n{player} turn to move\n")

            # Step 3, Step 4, Step 5-1
            test_move(board, player, pieces_player)
            i += 1

            # Step 5-2: Display the new game state
            print_game(board, naught_pieces, cross_pieces)

            # Step 6: Check if the game is over: stalemate or won by someone
            if check_stalemate(board, naught_pieces, cross_pieces) == True:
                print("Stalemate!")
                # Step 7
                game_over()
            elif check_win(board):
                print(check_win(board), "wins!")
                game_over()

    return whole_game()

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