Chess

Assignment 1
Semester 2, 2021
CSSE1001/CSSE7030

Due date: 16:00, 10th of September, 2021 GMT+10

1 Introduction

For this assignment you will implement a text-based version of chess. We are using the rules found in the Gameplay section of chess' Wikipedia page.[1] The specific details for how your program should run are described in Section 3.

You are required to implement a number of functions as per Section 4 of this document.

As with Assignment 0, your program must operate *exactly* as specified. In particular, your program's output must match *exactly* with the expected output. Your program will be marked by automatically so minor differences in output (such as whitespace or casing) *will* cause tests to fail resulting in a *zero mark* for that test.

2 Getting Started

Download `a1.zip` from Blackboard — this archive contains the necessary files to start this assignment.

Once extracted, the `a1.zip` archive will provide the following files/directories:

`a1.py` *This is the only file you will submit* and is where you write your code. *Do not* make changes to any other files.

`a1_support.py` *Do not modify or submit this file*, it contains functions to help you implement your tasks. You do not need to read or understand the code itself, but you should read the docstrings in order to get an idea of how the functions work.

`gameplay` This folder contains example gameplay which you should use to test that your code is working correctly.

3 Gameplay

When your program runs, the initial board state should be displayed, and the user should be prompted for white's first move. The standard prompt for white's move should read '`White's move:` ', and the standard prompt for black's move should read '`Black's move:` '. When a user is prompted for a move, they may do one of four things:

Enter '`h`' or '`H`': Print help text and re-prompt for move (without switching turns). See `gameplay/help-and-quit.txt` for the expected output.

[1] `https://en.wikipedia.org/wiki/Rules_of_chess`

Enter `'q'` or `'Q'`: Prompt the user to confirm they want to quit the game. Terminate the game on `'y'` or `'Y'` and re-commence play (prompt for move without switching turns) on any other input. See `gameplay/help-and-quit.txt` for the expected output.

Enter a valid move: The move should be performed, the updated board should be displayed, and the player whose turn it is should switch. The user whose turn it becomes should be prompted for their move. A valid move will be of the form `'from_square to_square'`. Users must enter the positions as a letter-number (column-row) pair. For example, `'e2 e4'` is a valid move for white from the initial board state, and is shown at the beginning of the `gameplay/checkmate.txt` file.

Enter an invalid move: The user should be informed that their move is invalid before being re-prompted for another move. An invalid move may be text in any form that does not conform to one of the above categories, or it may be a move in valid form that is not a valid move based on the rules of chess. Example invalid moves are shown in `gameplay/invalid-moves.txt`.

When a *valid* move is made, any of the following situations could arise:

Checkmate: The move puts the other player in checkmate. The players should be informed of the outcome, and the game should terminate; see `gameplay/checkmate.txt`.

Check: The move puts the other player in check but not checkmate (note: a player is not allowed move themselves into check). The other player should be informed that they are in check before being prompted for their move; see, for example, `gameplay/in_check.txt`.

Stalemate: The move causes a stalemate. The players should be informed of the outcome, and the game should terminate; see `gameplay/stalemate.txt`.

Normal: A piece is moved on the board without triggering any special case. Update the board, display it, and prompt the opposite player for their move.

4 Implementation

This section outlines the functions you are required to write. You are awarded marks for the number of tests passed by your functions. Thus an incomplete assignment with working functions may well be awarded more marks than a complete assignment with faulty functions.

4.1 Data Structures and Game Encoding

The chess board is stored as an octuple of strings (a tuple with eight string entries). Each string represents one row of the chess board, and each character of the string represents a chess piece or an empty square. Pieces and empty squares are encoded in the following way:

- `'P'`, `'K'`, `'Q'`, `'B'`, `'N'`, `'R'` represent white's: pawn, king, queen, bishop, knight, and rook, respectively.
- `'p'`, `'k'`, `'q'`, `'b'`, `'n'`, `'r'` represent black's: pawn, king, queen, bishop, knight, and rook, respectively.
- `'.'`: An empty square.

You may also *use the constants* in the `a1_support.py` file *when referring to pieces* which will make you code more readable.

Positions are integer tuples holding `(row, column)` where the `(0,0)` position is the *top-left* square of the board.

In accordance with the above, the function definitions throughout this section include type hints, where the following type aliases exist (and are defined in `a1_support.py`):

- `Board = Tuple[str, str, str, str, str, str, str, str]`
- `Position = Tuple[int, int]`
- `Move = Tuple[Position, Position]`

You are not obligated to include type hints in your assignment, though you must communicate the type information somehow (either through type hints or docstrings).

4.2 `main` function

The `main` function should be called when the file is run, and coordinates the overall gameplay. The `main` function should prompt for and process user input, and update and display the board state until either the game ends or the user quits. The output from your main function (including prompts) must exactly match the expected output. The `gameplay` folder shows the output of a correctly implemented game given example user moves. Running the sample tests will also give you a good idea of whether your prompts and other outputs are correct.

You should develop `main` in parallel with the rest of the functionality; that is, while you won't be able to achieve a fully functioning `main` before completing all other functions, you can incrementally increase functionality each time you complete one of the other functions. The `main` function should be fairly short, and should utilize other functions you have written. In the provided `a1.py`, the function definition for `main` has already been provided, and the `if __name__ == __main__:` block will ensure that the code in the `main` function is run when your `a1.py` file is run. Do *not* call your `main` function (or any other functions) outside of this block.

4.3 User Interaction

This section outlines *functions you need to write* for processing user input and updating board state. Example outputs for each function, as well as an example of using the functions together, are found at the end of the section.

4.3.1 `initial_state() -> Board:`

Return the board state for a new game.

4.3.2 `print_board(board: Board) -> None:`

Print a human-readable board. There are two spaces between a row and its number, and one empty line between the board and the row of letters `abcdefgh`.

4.3.3 `square_to_position(square: str) -> Position:`

Convert chess notation to its `(row, col): Position` equivalent. The top-left corner of the board, `a8`, corresponds to the position `(0, 0)`. The bottom-left corner of the board, `a1`, corresponds to the position `(7, 0)`.

4.3.4 `process_move(user_input: str) -> Move:`

Assume the `user_input` is valid and convert the user input to a move based on `(row, col): Position`.

4.3.5 `change_position(board: Board, position: Position, character: str) -> Board:`

Return a copy of `board` with the character at `position` changed to `character`. Note in the example usage that the second call to `change_position` does not preserve the change from the first call.

4.3.6 `clear_position(board: Board, position: Position) -> Board:`

Clear the piece at `position` (i.e. replace with an empty square) and return the resulting board. The board will remain unchanged when there is no piece at this position.

4.3.7 `update_board(board: Board, move: Move) -> Board:`

Assume the move is valid and return an updated version of the board with the move made.

4.3.8 Example usage

```
>>> board = initial_state()
>>> board
("rnbqkbnr", "pppppppp", "........", "........", "........",
   "........", "PPPPPPPP", "RNBQKBNR")
>>> print_board(board)
rnbqkbnr 8
pppppppp 7
........ 6
........ 5
........ 4
........ 3
PPPPPPPP 2
RNBQKBNR 1

abcdefgh
>>> square_to_position("e4")
(4, 4)
>>> process_move("e2 e4")
((6, 4), (4, 4))
>>> process_move("b1 g3")
((7, 1), (5, 6))
>>> change_position(board, (2, 4), BLACK_PAWN)
("rnbqkbnr", "pppppppp", "....p...", "........", "........",
   "........", "PPPPPPPP", "RNBQKBNR")
>>> change_position(board, (2, 5), WHITE_PAWN)
("rnbqkbnr", "pppppppp", ".....P..", "........", "........",
   "........", "PPPPPPPP", "RNBQKBNR")
>>> clear_position(board, (0, 0))
(".nbqkbnr", "pppppppp", "........", "........", "........",
   "........", "PPPPPPPP", "RNBQKBNR")
>>> clear_position(board, (4, 4))
("rnbqkbnr", "pppppppp", "........", "........", "........",
   "........", "PPPPPPPP", "RNBQKBNR")
>>> update_board(board, process_move("e2 e4"))
```

```
("rnbqkbnr", "pppppppp", "........", "........", "....P...",
   "........", "PPPP.PPP", "RNBQKBNR")
>>> # Because we are returning copies of board, board remains
   unchanged
>>> board
("rnbqkbnr", "pppppppp", "........", "........", "........",
   "........", "PPPPPPPP", "RNBQKBNR")
>>> # By reassigning board to be the result of the function call,
   functions can work together
>>> board = update_board(board, process_move("e2 e4"))
>>> board = update_board(board, process_move("e7 e5"))
>>> # An initial move has been made for each player, and board is
   updated
>>> print_board(board)
rnbqkbnr 8
pppp.ppp 7
........ 6
....p... 5
....P... 4
........ 3
PPPP.PPP 2
RNBQKBNR 1

abcdefgh
>>>
```

4.4 Move Validation

This section outlines *functions you need to write* to handle move validation. Example outputs for each function are displayed at the end of the section.

4.4.1 `is_current_players_piece(piece: str, whites_turn: bool) -> bool:`

Returns true only when `piece` is belongs to the player whose turn it is.

4.4.2 `is_move_valid(move: Move, board: Board, whites_turn: bool) -> bool:`

Returns true only when the move is valid on the current board state for the player whose turn it is. A move is valid if *all* of the following are true:

- Both positions in the move exist on the board;
- The positions in the move are different (i.e. it is not valid to move a piece to the position it is already at);
- The piece being moved belongs to the player who is trying to move it;
- The square the piece is being moved to is empty or contains a piece of the opposite colour;
- The move is a valid for the type of piece being moved, according to the rules of chess (e.g. a diagonal move would be invalid for a rook); and
- The move does not put the player whose turn it is in check.

4.4.3 Example usage

```
>>> is_current_players_piece(WHITE_KING, True)
True
>>> is_current_players_piece(WHITE_KING, False)
False
>>> is_current_players_piece(BLACK_KING, False)
True
>>> is_current_players_piece(EMPTY, True)
False
>>> is_current_players_piece(EMPTY, False)
False
>>>
>>> board = initial_state()
>>> board = update_board(board, process_move("e2 e4"))
>>> board = update_board(board, process_move("e7 e5"))
>>> print_board(board)
rnbqkbnr  8
pppp.ppp  7
........  6
....p...  5
....P...  4
........  3
PPPP.PPP  2
RNBQKBNR  1

abcdefgh
>>> next_move = process_move("g1 f3")
>>> is_move_valid(next_move, board, True)
True
>>> is_move_valid(next_move, board, False)
False
>>> # Because it is whites turn and the move is valid for white,
   we can update the board
>>> board = update_board(board, next_move)
>>> print_board(board)
rnbqkbnr 8
pppp.ppp 7
........ 6
....p... 5
....P... 4
.....N.. 3
PPPP.PPP 2
RNBQKB.R 1

abcdefgh
>>> next_move = process_move("e5 e4")
>>> is_move_valid(next_move, board, False)
False
>>> # Even though the piece in the move belongs to the correct
   player, it is invalid by the rules of chess
```

4.5 End of Game Checks

This section outlines functions used to determine whether a checkmate or stalemate has occurred. Example outputs are shown at the end of the section.
Hint: A player is 'in check' if their king is under attack (can be reached) by one of the other player's pieces. A 'checkmate' occurs when the player is in check and can't make any valid moves to escape check. The `is_in_check` function defined in `a1_support.py` can be used to tell you whether a player is in check or not.

4.5.1 `can_move(board: Board, whites_turn: bool) -> bool:`

Returns true only when the player can make a valid move which does not put them in check.

4.5.2 `is_stalemate(board: Board, whites_turn: bool) -> bool:`

Returns true only when a stalemate has been reached. A stalemate occurs when the player who is about to move isn't currently in check but can't make any moves without putting themselves in check.

4.5.3 `check_game_over(board: Board, whites_turn: bool) -> bool:`

Returns true only when the game is over (either due to checkmate or stalemate). Also prints information about the result if the game is over, or if the player is in check.

4.5.4 Example usage: check and checkmate

```
>>> print_board(board)
....k... 8
........ 7
......R. 6
........ 5
........ 4
........ 3
.....q.. 2
r......K 1

abcdefgh
>>> is_in_check(board, True)
True
>>> can_move(board, True)
True
>>> # Whites king is in check but can escape by blocking with the
    rook
>>> check_game_over(board, True)

White is in check
False
>>> # However, if we deletes whites rook, white cannot move out of
    check (checkmate)
>>> rook_position = square_to_position("g6")
>>> board = clear_position(board, rook_position)
>>> print_board(board)
```

```
....k... 8
........ 7
........ 6
........ 5
........ 4
........ 3
.....q.. 2
r......K 1

abcdefgh
>>> is_in_check(board, True)
True
>>> can_move(board, True)
False
>>> check_game_over(board, True)

Checkmate
True
>>>
```

4.5.5 Example usage: stalemate

```
>>> print_board(board)
k....... 8
..Q..... 7
....K... 6
........ 5
........ 4
........ 3
........ 2
........ 1

abcdefgh
>>> is_in_check(board, False)
False
>>> can_move(board, False)
False
>>> is_stalemate(board, False)
True
>>> # A stalemate has occurred, because black is not in check but
    cannot move any pieces without putting themself in check
>>> check_game_over(board, False)

Stalemate
True
>>> # "\nStalemate" is printed, and True is returned
>>>
```

4.6 CSSE7030 Task

If you are enrolled in CSSE7030, you are required to complete the additional task of adding special moves. There are three special moves that you are required to implement:

Promotion If a pawn reaches the final row at the opposite end of the board, the player must promote it to any non-pawn, non-king piece of its own colour.

Castling If the player has not yet moved their king and a rook during the game, and there are no pieces in between the king and rook, and none of the positions from the kings position to the rooks position (inclusive) are under attack, the player may 'castle'. This move involves moving the king two positions towards the rook, and then placing the rook in the position the king skipped over.

En passant If, on the prior move, the opponent moved a pawn two positions from its starting position, and that pawn could have been captured by one of the player's pawns if it was to have only moved one position, the player may take the opponent's pawn with their own, on the position the opponent's pawn skipped over.

These moves must be implemented via the functions described below, and their functionality must be integrated into the `main` function. Example usage for the functions are shown at the end of this section.

4.6.1 `attempt_promotion(board: Board, whites_turn: bool) -> Board:`

Checks whether there is a pawn on the board that needs to be promoted. If there is, this function should prompt the user and ask which piece should replace the pawn. Update the board if necessary then return the board.

4.6.2 `is_valid_castle_attempt(move: Move, board: Board, whites_turn: bool, castling_info: Tuple[bool, bool, bool]) -> bool:`

Returns true only when the given move is a valid attempt at castling for the current board state. `castling_info` is a tuple of booleans which are true if the player's left rook, king, and right rook have moved this game, respectively.

4.6.3 `perform_castling(move: Move, board: Board) -> Board:`

Given a valid castling move, returns the resulting board state.

4.6.4 `is_valid_en_passant(move: Move, board: Board, whites_turn: bool, en_passant_position: Optional[Position]) -> bool:`

Returns true only when the supplied move constitutes a valid en passant move. `en_passant_position` is the position skipped by the opponent's pawn, or `None` if a pawn did not move forward two positions on the last move.

4.6.5 `perform_en_passant(move: Move, board: Board, whites_turn: bool) -> Board`

Given a valid en passant move, returns the resulting board state.

4.6.6 `update_castling_info(move: Move, whites_turn: bool, castling_info: Tuple[bool, bool, bool]) -> Tuple[bool, bool, bool]:`

Returns the updated castling information for the player whose turn it is, after performing the given, valid move.

4.6.7 update_en_passant_position(move: Move, board: Board, whites_turn: bool) -> Optional

If the current player's pawn just moved forward two squares, returns the position that an opponent pawn could take to perform a valid en passant move. If no en passant move should be possible, returns None.

4.6.8 Example usage: promotion

```
>>> print_board(board)
k...P... 8
........ 7
........ 6
........ 5
........ 4
........ 3
........ 2
.......K 1

abcdefgh
>>> board = attempt_promotion(board, True)
What piece would you like (q, n, r, b)? q
>>> print_board(board)
k...Q... 8
........ 7
........ 6
........ 5
........ 4
........ 3
........ 2
.......K 1

abcdefgh
>>>
```

4.6.9 Example usage: en passant

```
>>> print_board(board)
rnbqkbnr 8
.ppppppp 7
........ 6
p...P... 5
........ 4
........ 3
PPPP.PPP 2
RNBQKBNR 1

abcdefgh
>>> move = process_move("d7 d5")
>>> board = update_board(board, move)
>>> en_passant_position = update_en_passant_position(move, board,
    False)
```

```
>>> en_passant_position
(2, 3)
>>> print_board(board)
rnbqkbnr 8
.pp.pppp 7
........ 6
p..pP... 5
........ 4
........ 3
PPPP.PPP 2
RNBQKBNR 1

abcdefgh
>>> # Try to have white en passant to take the pawn black just
    moved
>>> move = process_move("e5 d6")
>>> is_valid_en_passant(move, board, True, en_passant_position)
True
>>> # Valid to en passant, so perform the move
>>> board = perform_en_passant(move, board, True)
>>> print_board(board)
rnbqkbnr 8
.pp.pppp 7
...P.... 6
p....... 5
........ 4
........ 3
PPPP.PPP 2
RNBQKBNR 1

abcdefgh
>>>
```

4.6.10 Example usage: castling

```
>>> print_board(board)
rnbqkb.r 8
pppp.ppp 7
........ 6
.B..p... 5
....n... 4
.....N.. 3
PPPP.PPP 2
RNBQK..R 1

abcdefgh
>>> move = process_move("e1 g1") # Phrased as the move for the
    king (moving 2 squares right)
>>> is_valid_castle_attempt(move, board, True, (False, False,
    False))
True
```

```
>>> board = perform_castling(move, board)
>>> print_board(board)
rnbqkb.r 8
pppp.ppp 7
........ 6
.B..p... 5
....n... 4
.....N.. 3
PPPP.PPP 2
RNBQ.RK. 1

abcdefgh
>>>
```

5 Assessment and Marking Criteria

This assignment assesses course learning objectives:

1. apply program constructs such as variables, selection, iteration and sub-routines,
2. read and analyse code written by others,
3. read and analyse a design and be able to translate the design into a working program, and
4. apply techniques for testing and debugging.

5.1 Functionality

Your program's functionality will be marked out of a total of 6 marks. As in assignment 0, your assignment will be put through a series of tests and your functionality mark will be proportional to the number of tests you pass. You will be given a *subset* of the functionality tests before the due date for the assignment.

Task	*CSSE1001 Weight*	*CSSE7030 Weight*
User Interaction	35%	30%
Move Validation	20%	15%
End of Game Checks	35%	30%
`main` function	10%	10%
Special Moves	0%	15%

Table 1: Percentage of functionality mark per task.

You may receive partial marks within each section for partially working functions, or for implementing only a few functions.

You need to perform your *own* testing of your program to make sure that it meets *all* specifications given in the assignment. Only relying on the provided tests is likely to result in your program failing in some cases and you losing some functionality marks. Note: Functionality tests are automated, so string outputs need to match *exactly* what is expected.

Your program must run in the Python interpreter (the IDLE environment). Partial solutions will be marked but if there are errors in your code that cause the interpreter to fail to execute your program, you will get zero for functionality marks. If there is a part of your code that causes the interpreter to fail, comment out the code so that the remainder can run. Your program must run using the Python 3.9 interpreter. If it runs in another environment (e.g. Python 3.8 or PyCharm) but not in the Python 3.9 interpreter, you will get zero for the functionality mark.

5.2 Code Style

The style of your assignment will be assessed by a tutor. Style will be marked according to the style rubric provided with the assignment. The style mark will be out of 4.

The key consideration in marking your code style is whether the code is easy to understand. There are several aspects of code style that contribute to how easy it is to understand code. In this assignment, your code style will be assessed against the following criteria.

- Readability
  - Program Structure: Layout of code makes it easy to read and follow its logic. This includes using whitespace to highlight blocks of logic.
  - Descriptive Identifier Names: Variable, constant, and function names clearly describe what they represent in the program's logic. Do not use Hungarian Notation for identifiers. In short, this means do not include the identifier's type in its name, rather make the name meaningful (e.g. employee identifier).
  - Named Constants: Any non-trivial fixed value (literal constant) in the code is represented by a descriptive named constant (identifier).
- Algorithmic Logic
  - Single Instance of Logic: Blocks of code should not be duplicated in your program. Any code that needs to be used multiple times should be implemented as a function.
  - Variable Scope: Variables should be declared locally in the function in which they are needed. Global variables should not be used.
  - Control Structures: Logic is structured simply and clearly through good use of control structures (e.g. loops and conditional statements).
- Documentation:
  - Comment Clarity: Comments provide meaningful descriptions of the code. They should not repeat what is already obvious by reading the code (e.g. `# Setting variable to 0`). Comments should not be verbose or excessive, as this can make it difficult to follow the code.
  - Informative Docstrings: Every function should have a docstring that summarises its purpose. This includes describing parameters and return values (including type information) so that others can understand how to use the function correctly.
  - Description of Logic: All significant blocks of code should have a comment to explain how the logic works. For a small function, this would usually be the docstring. For long or complex functions, there may be different blocks of code in the function. Each of these should have an in-line comment describing the logic.

5.3 Assignment Submission

This assignment follows the assignment submission policy as assignment 0. Please refer to the assignment 0 task sheet.
You must submit your assignment as a single Python file called `a1.py` (use this name – all lower case), and *nothing else*. Your submission will be automatically run to determine the functionality mark. If you submit a file with a *different name*, the tests will *fail* and you will get *zero* for functionality. Do *not* submit the `a1_support.py` file, or any other files. Do *not* submit any sort of archive file (e.g. zip, rar, 7z, etc.).

5.4 Plagiarism

This assignment follows the same plagiarism policy is as per assignment 0. Please refer to the assignment 0 task sheet.

6 Updates

6.1 v2.0

#266 Typo of `sqaure_to_position` in supplied `a1.py` corrected to `square_to_position`.

#270 Gameplay section updated to call positions (column-row) pairs rather than (row-column) pairs.

#280 Gameplay examples in `gameplay` directory updated to be consistent with the specifcation. Specifically, to have two spaces separating the board and the row numbers.

#282 The `out_of_bounds` docstring in `a1_support.py` was updated to reflect it's functionality.