Project 2

Introduction

For all function calls, you can assume the following:

- all arguments will be valid inputs (e.g. cards will always take the form of a 2-character string `'VS'` where `'V'` is the value — one of `'A234567890JQK'` and `'X'` for the bonus version of the game — and `'S'` is the suit — one of `'SHDC'` and `'X'` for the bonus game);
- the game state at the time your function is called will be valid, i.e. all players will have played legally; in the tournament context, every time a player attempts to play, we check to make sure that the play is valid, and if not, that player forfeits the game (and is the loser, with the other three players being combined winners), meaning that your functions won't ever be called over game states generated by invalid plays;
- the game will be played with a combined (and shuffled!) set of two card decks, meaning there will be exactly two instances of every card in play (and two Jokers in the bonus game).

Another thing to bear in mind is that the state space for the game is very large, making it very difficult to carry out anything more than cursory testing of most of the functions. As such, fully expect the tournament to expose bugs in your code that the standard green-diamond-based testing does not expose. Note that you always have access to a full dump of all games your player is involved in for the tournament, including games where a run-time or logical error occurred, and that these should form part of your medium-term testing (and contribute to you being more confident of doing well over the "hidden hidden" test cases).

And one last thing: as always with coding, make sure you are well versed in the rules of the game (and have ideally played it with others a number of times to have got your head around its ins and outs, and possibly to have started to come up with laying strategies) before attempting any of the questions. A full description of the rules of the game can be found on the Projects page of the LMS, and should be your starting point for the project. All function definitions here assume familiarity with the game.

Question 1

Write a function `comp10001huxxy_score()` which takes a single argument:

- `cards`, a list of cards held by a player, each in the format of a 2-letter string

Your function should return a non-negative integer indicating the score for the combined cards, based on the value of each card (`'A'` = 1, `'2'` = 2, ..., `'K'` = 13).

Question 2

Write a function `comp10001huxxy_valid_table()` which takes a single argument:

- `groups`, a list of lists of cards (each a 2-element string, where the first letter is the card value and the second letter is the card suit, e.g. `'3H'` for the 3 of Hearts), where each list of cards represents a single group on the table, and the combined list of lists represents the combined groups played to the table.

Your function should return a `bool`, which evaluates whether the table state is valid or not. Recall from the rules of the game that the table is valid if all groups are valid, where a group can take one of the following two forms:

- an **N-of-a-kind** (i.e. three or more cards of the same value), noting that in the case of a 3-of-a-kind, each card must have a unique suit (e.g. `['2S', '2S', '2C']` is *not* a valid 3-of-a-kind, as the Two of Spades has been played twice), and if there are 4 or more cards, all suits must be present.
- a **run** (i.e. a group of 3 or more cards, starting from the lowest-valued card, and ending with the highest-valued card, forming a continuous sequence in terms of value, and alternating in colour; note that the specific ordering of cards in the list is not significant, i.e. `['2C', '3D', '4S']` and `['4S', '2C', '3D']` both make up the same run.

Question 3

Write a function `comp10001huxxy_valid_play()` which takes five arguments:

- `play`, a 3-tuple representing the play that is being attempted; see below for details;
- `play_history`, a list of 3-tuples representing all plays that have taken place in the game so far (in chronological order); each 3-tuple is based on the same structure as for `play`;
- `active_player`, an integer between 0 and 3 inclusive which represents the player number of the player whose turn it is to play;

- `hand`, a list of the cards (each in the form of a 2-character string, as for Q1) held by the player attempting the play;
- `table`, a list of list of cards representing the table (in the same format as for Q2).

Your function should return a Boolean indicating whether the play is valid or not given the current game state (i.e. the combination of the plays made to date, the content of the player's hand, and the groups on the table). In this, you only need to validate the state of the table (using `comp10001huxxy_valid_table` from Q2, which you are provided with a reference implementation of) if the play ends the player's turn and they have played to the table. Note that `play_history`, `hand`, and `table` all represent the respective states prior to the proposed play being made (e.g. `play_history` will not contain `play`).

The composition of the 3-tuple used to represent each play is `(player_turn, play_type, play_details)`, where `player_turn` is an integer (between 0 and 3 inclusive) indicating which player is attempting to play, and `play_type` and `play_details` are structured as follows, based on the play type:

- pick up a card from stock (and thereby end the turn): `play_type = 0`, `play_details = None`;
- play a card from the hand to the table: `play_type = 1`, `play_details = (card, to_group)` where `card` is the card from the hand that is to be played, and `to_group` is the (zero-offset) index of group in `table` to play to; in the instance that the card is to start a new group, `to_group`should be set to the one more than the index of the last group on the table (i.e. if there are three groups, the last group will be index 2, so 3 would represent that the card is to be used to start a new group);
- play a card from one group on the table to another: `play_type = 2`, `play_details = (card, from_group,to_group)` where `card` is the card to be played from the group, `from_group` is the (zero-offset) index of the group in `table` to play `card` from, and `to_group` is the index of the group in `table` to play `card` to (and, similarly to above, a value of one more than the index of the last group indicates that a new group is to be formed)
- end the turn, after playing from the hand or play between groups on the table: `play_type = 3`, `play_details = None`.

Note that picking up a card (`play_type = 0`) implicitly ends the turn, whereas if plays are made from the hand/between groups on the table, an explicit "end of turn" play (`play_type = 3`) must be used to confirm that the player is ending their turn.

Question 4

The fourth question requires that you implement the function that is called in the tournament `comp10001huxxy_play()`, within each of your turns. The function takes four arguments (all of which are identical in detail with the arguments of the same name to `comp10001huxxy_valid_play`):

- `play_history`, a list of 3-tuples representing all plays that have taken place in the game so far (in chronological order); each 3-tuple is based on the same structure as for `play` in Q3;
- `active_player`, an integer between 0 and 3 inclusive which represents which the player number of the player whose turn it is to play;
- `hand`, a list of the cards (each in the form of a 2-character string, as for Q1) held by the player attempting the play;
- `table`, a list of list of cards representing the table (in the same format as for Q1).

Your function should return a 3-tuple based on the same structure as `play` in Q3.