INF131 : Project 2021 - Blackjack game

This project must be done in pairs. You must send your code and report to your TP teacher before the last TP (week of 13 December), when you will demo / defend your project.

1 Introduction

In this project we propose to create a card game: Blackjack. Some functions are guided, but you can (and should) also let your imagination run wild to add more functionality, a GUI, etc. After the end of section B, it is not necessary to follow the order of the sections exactly (for example it is possible to code a graphical interface before having integrated smart players).

1.1 Game rules

We focus on a simplified version of the Blackjack game, that is played with several decks of 52 cards. Each deck of 52 cards is divided in 4 colours (spades, hearts, clubs, diamonds), with 13 cards per colour numbered from 1 (ace) to 10, then jack, queen, king.

The goal of a Blackjack game is to get as close as possible of 21 points, but without going past this total. The numerical values of the cards are as follows:

- Cards numbered from 2 to 10 are valued as many points as their number (so between 2 and 10 points)
- The ace is valued either 1 or 11 points, at the player's choice
- Figures (jack, queen, king) are valued 10 points each

The players start the game with 2 cards. At their turn, each player can choose to stop with their current total, or to draw a new card (face down) to add it to their total. If the new total then exceeds 21 points, the player loses immediately. If the total equals exactly 21 points, the player wins immediately. In other cases, the player remains in the game and may on his next turn again choose to continue or stop. When no player is left in game (all have stopped or lost) then the winner is the stopped player who is closest to 21.

1.2 Data structures

We want to be able to play several games in a row. We are therefore going to handle several scores for each player, which will therefore have to be stored in different lists or dictionaries.

- The total of points of the current game, reset to 0 at the start of each game, and equal to the sum of all cards drawn by the player so far during this game. If this total goes over 21, the player loses the game.
- The total of victories or money gained by the player after all the games played. At first, we will count 1 point per game won, and 0 point per game lost. Later, the players will get an initial amount of money that they can bet on games, and the money they gain (or lose) for a game will be proportional to their bet.

1. Option 1: scores, number of victories, and money gains of each player are stored in lists, so they can be accessed from the player number (index in the corresponding list).
2. Option 2: scores, number of victories, and money gains of players are stored in dictionaries whose keys are the players' names, so they can be accessed with the player's name instead of the player's number.

2 Part A: initialisations

2.1 A1 - cards deck

1. Write a function `deck()` that initialises and returns a deck of 52 cards. This function has no argument. It returns a list of string names of the 52 cards, for instance "ace of spades", "2 of diamonds", "king of hearts".

2. Write a function `valueCard(card)` that receives a card name (string at the format above) and returns its numerical value (see rules above). **Warning:** the ace is a special case that needs interacting with the player to ask to choose the value wanted (either 1 or 11, the value must be filtered). *Clue*: the card's colour does not count in the value, so you should start by extracting the card type (which number or figure it is) from its name.

3. Write a function `initStack(n)` that receives as argument the number n of players, and return the drawing stack composed of n decks of 52 cards. This function must use the function `deck()` above to generate each deck, then shuffle them all together, and return the randomly reordered list. Reminder: use module `random`.

4. Write a function `drawCard(p,x)` that receives a list p of cards (the drawing pile) and an optional argument x (the number of cards to draw, default value 1). This function draws x cards at the top of pile p (which **modifies** p), and returns the list of cards drawn (if $x = 1$ it returns a singleton).

2.2 A2 - players and scores

1. Write a function `initPlayers(n)` that receives the number of players n, loops to ask the user for the name of each player, stores all names in a list of names, which it returns.

2. Write a function `initScores(joueurs,v)` that receives the list of players' names and an initial value v (optional starting money, defaulted to 0). This function builds a dictionary whose keys are the players' names, and whose values are their initial score v. The function returns this dictionary.

3. Write a function `firstTurn(players)` that receives the list of players, builds the initial scores dictionary with null scores (by calling function `initScores`), draws 2 cards per player (with function `drawCard`), and updates their scores with the values of their 2 cards. This function returns the scores dictionary thus built and initialised.

4. Write a function `winner(scores)` that receives a scores dictionary, and returns the name and score of the winning player. *Reminder*: the winner is the player whose score is closest to (but smaller or equal to) 21.

3 Part B: game management

3.1 B1 - Player's turn

- Write a boolean function `continue()` that interacts with the player to know if they want to continue or stop. This function must read the answer, filter it until it has one of the allowed values (of your choice: yes/no, go/stop...). This function returns True if the player wants to continue, False if they chose to stop.

- Write a function `playerTurn(j)` that receives the name of a player, and manages their game turn (warning: you need to pass as arguments **all** the needed variables; also take care about side effects) with the following steps:

  – Displays: turn number, name of player, current total of points for this current game (smaller than 21);

  – Interacts: proposes to continue or stop and reads answer;

  – If the player continues, draws first card of the pile, computes its value, adds it to the toal, and checks for victory or defeat;

  – If the player stops or loses, they must be removed from the list of players still in the game.

3.2 B2 - A complete game

- Write a function `gameTurn` that gives one game turn to each player still in game. This function should therefore call the previous `playerTurn` function for each of them. Warning: this function might modify the list of players still in game, since players who lose or stop must be removed from this list.

- Write a boolean function `gameOver` that checks if the current game is finished (returns `True`) or not yet (returns `False`)
- Write a function `completeGame` that repeatedly calls function `gameTurn` until the game is over and the winner is known. This function must then update the players' numbers of victories (in the corresponding dictionary). For the moment, the winner gets 1 point for the victory, and the other players get 0.

3.3 B3 - main program

Write a main program that calls the functions defined above to:

- Asks the user for the number of players;
- Initialises all necessary variables (cards deck, list of players, dictionaries...);
- Plays one complete game;
- Offers to play a new game. if yes: restarts a new game with the same list of players; if no: finishes with a message (option: indicate the total number of games played, and the overall winner).

Need to finish the victories mechanism. The approach is changed from what is proposed above (everything explained on .docx)

Warning: your main program must not repeat any code, but just call the functions already defined.

3.4 B4 - bets

We now want to let players choose at the start of the game how much they wish to bet, after seeing their first 2 cards. The program asks for the bet amount, and filters it to be strictly positive and smaller or equal to their total amount of money. At the end of the game, the winner collects the bets of all other players and adds them to his/her own money total.

At the start of the program, each player starts not with 0 points, but with 100 kopecs that can be gambled. Players who have no more kopecs are eliminated for future games. When there is only one player left, the program does not offer to replay anymore, but finishes and displays the winner's name.

- Write a new version of your program that allows to play with bets (warning: you must also keep the first version without bets).
- Option: the program can let players leave the table between games (so they leave with the money they earned so far).

4 Part C: some intelligence

4.1 C1 - the croupier (to draw or not to draw, that is the question)

We now add the role of the croupier, that is not played by a human but by the program, with an "artificial intelligence". The goal of the other players is to beat the croupier, by obtaining more points than him, or by pushing the game until the croupier goes over 21 before the other players. In this first version, we consider that the croupier always bets the same value (for instance 10 kopecs), but we want to define several functions representing different ways of deciding wether to continue drawing or not.

- Write a first version of the croupier that randomly decides wether to continue drawing or to stop.
- Write a parameterised random croupier, that is more or less "playful", by receiving as argument the probability to keep drawing. With a received probability of 1, the croupier will take a maximum of risks and always continue playing (so he will most probably lose by going over 21); with a received probability of 0, the croupier will take no risk at all and stop immediately with his first 2 cards (even with 0 points, which has no interest); with a received probability of 0.5, the behaviour is the same as the random function above.
- Write a smarter croupier whose probability to stop is based on how far he is from 21. The closer he is to 21, the more risk there is to go over, so the higher the probability to stop should be.

- Write other versions that use the strategies of your choice to optimise the behaviour of the croupier. You can for instance consider the totals of the other players, the cards already known or left on the deck, etc. Your strategies must be explained in the comments and in your project report.

Write one function per strategy, and give them expressive names (for instance `stupid`, `riskall`, `randomp`, etc. Test all your strategies by playing several games with your croupier.

4.2 C2 - the croupier (choice of bets)

Write different functions allowing the smart croupier to use different strategies to choose his initial bet (based on his first 2 cards). For instance you can use the following criteria:

- amount of money left (compared to the initial total of 100 that can have evolved over the previous games), for instance bet a given percentage of the money left;
- a risk-aversion value (always bet the smallest amount, or the maximum amount, or some intermediate value);
- the total value of the first 2 cards (and the resulting probability to get close to 21);
- the total values of the other players' cards (that are visible to all);
- or a combination of these criteria and any other criteria that you can think ok.

Again, your strategies must be explained in comments of your code, and in your project report.

4.3 C3 - the other players

Similar to the croupier, we now want to add the possibility at the start of the game to select for each player if they are played by a human of by the computer. We also want to be able to choose for automated player their type of strategy (among those defined above): totally random, risk-all, measured risk, etc. Based on the strategy selected by the user, the program must choose the initial bet and make the decision to stop or continue at each turn. You should call the functions defined in C1 and modified in C2.

4.4 C4 - automated tournament and comparison of strategies

Play several completetly automated games with players using your various strategies, in order to compare their efficacy. In order to smooth the randomness of the drawing pile, you must play enough games. Then compare the resulting average score (over all games played) of the different strategies (incarnated by different players).

BONUS: by using `matplotlib` you can draw a curve of the average score in function of the probability to replay for the "probabilistic random" player. For instance the average score with a probability to replay of 0 is close to 0 (the player never continues and never wins, unless his initial draw is already valued at 21), as is the average score with a probability to replay of 1 (the player never stops in time so mostly loses as well).

5 Part D: graphical interface

Different approach - DONE

For now, the game is played in the text console. You can create a graphical interface that displays the cards and lets the player interacts by clicking in the window, etc. You can use `turtle` or tkinter for instance.