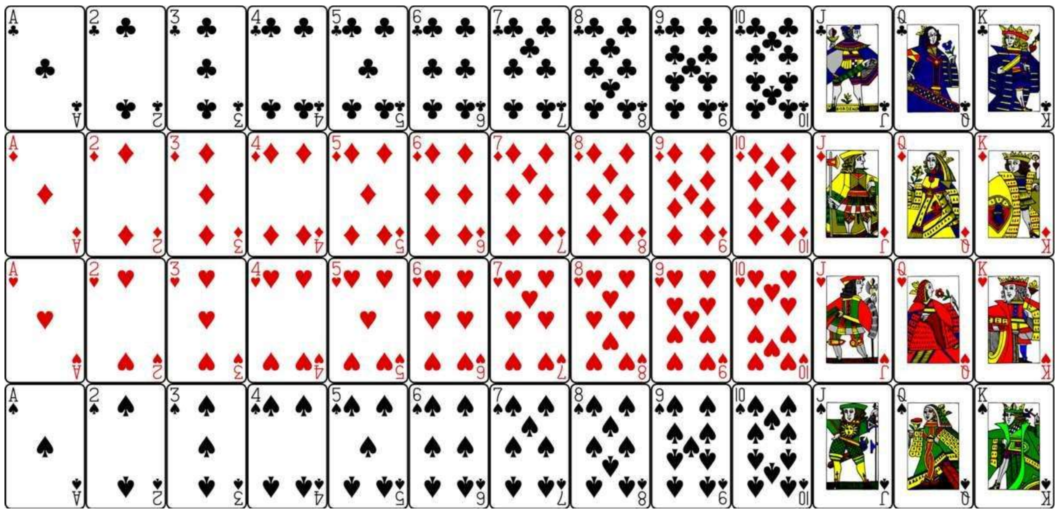
Assignment 2 – Playing Card BINGO

Due Date: Friday, November 20th 2020 at 23:55 (Edmonton time)
Percentage overall grade: 7%
Penalties: No late assignments allowed
Maximum marks: 100

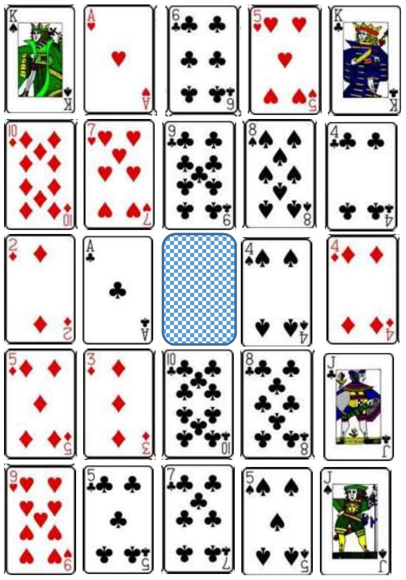
Assignment Specification

You are tasked with creating a two-player version of BINGO, played with two standard decks of 52 playing cards. A deck of 52 cards has four suits: club, diamond, heart, and spade. Each suit has 13 ranks: 2 through 10, Jack, Queen, King, and Ace, as shown below.



The first deck of cards will be used to create two BINGO grids, where each is a 5 by 5 grid of cards, as shown below. All of these cards will be dealt face up, except for the card in the very middle of the grid which will be face down. This will use up 50 cards, and the remaining 2 will be placed in a discard pile.

An example of a BINGO grid:



The second deck of cards will be used to draw calling cards from. Every time a calling card is drawn off the top of this deck, Players A and B check their BINGO grid for the matching card. If they find a match, they turn the card over in place (so that it is face down). Each player then checks if their face down cards have formed the winning BINGO pattern – if they have, the player shouts BINGO! to indicate that they have won the round. If neither Player A or B has a BINGO, the calling card is discarded into a second (separate) discard pile, and another calling card is drawn. This continues until one player (or both players, in the case of a tie) has a BINGO!

Task 1: Card class

Create a Python file called *playingCards.py*. Inside this file, create and test a Card class according to the description below. Note that the following methods form the public interface for this class – you must complete them all as specified, and cannot add additional methods to the public interface. However, you can include additional private helper methods that are only called within this class, if you wish.

`Card(rank, suit, faceUp)` – creates a card which is described by its rank and suit, that is either facing up (so that the rank and suit are visible) or facing down (so that only the back of the card can be seen). The `__init__` method should assert that the values of the input parameters are valid, before being used to initialize three private attributes of the same names (i.e. `self.__rank`, `self.__suit`, `self.__faceUp`). A valid rank is one of the following one-character strings: 'A', 'a', '2', '3', '4', '5', '6', '7', '8', '9', 'T', 't', 'J', 'j', 'Q', 'q', 'K', 'k' and a valid suit is one of the following one-character strings: 'S', 's', 'H', 'h', 'D', 'd', 'C', 'c'; but self.__rank and self.suit should be uppercase if it is a letter. A valid faceUp is Boolean, where a True value indicates that the card is facing up and a False value indicates that the card is facing down.

`getRank()` – returns the rank of the Card instance as a one-character string.

`getSuit()` – returns the suit of the Card instance as a one-character string.

`isFaceUp()` – returns the Boolean value indicating whether the Card instance is facing up (True) or down (False).

`turnOver()` – updates the Card instance so that if it was facing up, it will now be facing down. Similarly, if it was facing down, it will not be facing up. Nothing is returned.

`__eq__(anotherCard)` – checks to see if the Card instance and anotherCard are the same (i.e. both rank and suit). Returns a Boolean value. Note that this is a special method, and will be called when the equivalency operator (==) is used between two Card objects.

`__str__()` – returns the string representation of the Card instance. For example, a ten of hearts which is facing up will return the string `'[ TH ]'`, with a single space between the open square bracket and the rank, and a single space after the suit. Any card that is facing down will return the string `'[ ]'`, with one space between the two square brackets.

Test your Card class thoroughly before moving on. You may wish to use assertions to verify expected behaviour, like in Lab 7 (Linked Lists), though you don't have to. Place these tests under `if __name__ == "__main__":` in *playingCards.py* for the marker to see.

Task 2: Deck class

Create and test a Deck class in *playingCards.py*, according to the description below. Note that the following methods form the public interface for this class – you must complete them all as specified, and cannot add additional methods to the public interface. However, you can include additional private helper methods that are only called within this class, if you wish.

`Deck()` – creates an empty deck, capable of holding 52 standard playing cards. Notice that a deck of cards acts very much like a queue, where we deal the top card (front of our queue) and add new cards to the bottom of the deck (back of our queue). In the `__init__` method, create a single private attribute that is the most time-efficient queue with a maximum capacity that we covered in the lectures. You should import the queues.py file provided with Lab 6 to accomplish this. (You do not need to submit the queues.py file since your marker will also have access to that file.)

`addCard(card)` – modifies the deck by adding a new card (provided as input), face down to the bottom of the deck. Nothing is returned. You should assert that the card provided is a Card instance, where a resulting AssertionError has the argument 'Can only add cards to deck'. You should also raise an Exception if the deck is full, with the argument 'Cannot add xx: Deck is full' where xx is replaced with the string representation of the card (face up). This method should have an O(1) time efficiency.

`dealCard()` – modifies the deck by removing the card from the top of the deck, and returns that top card, face up. Raise an Exception if the deck is empty, with the argument 'Cannot deal card from empty deck'. This method should have an O(1) time efficiency.

`deckSize()` – returns the integer number of cards currently in the deck.

`isComplete()` – returns a Boolean value indicating whether the deck is complete (True) or not (False). A deck is considered to be complete if it contains all 52 playing cards, without repeats.

`__str__()` – returns the string representation of all of the cards in the deck. You can decide how this string should be formatted, but make it clear which is the top card and be sure to show the rank and suit of each card in some way (i.e. do not just show all cards as face down)

Test your Deck class thoroughly before moving on. Again, place these tests under `if __name__ == "__main__":` in *playingCards.py* for the marker to see. You now have a playingCards module that you can use for many different card game programs!

Task 3: Bingo class

Create a Python file called *bingo.py*. Inside this file, create a Bingo class according to the description below. Note that the following methods form the public interface for this class – you must complete them all as specified, and cannot add additional methods to the public interface. However, you can include additional private helper methods that are only called within this class, if you wish.

`Bingo()` – creates a new 2D bingo grid that is empty, and will search for winning BINGO lines by default. The empty grid should be stored in a private attribute. You should also create a private attribute to store the play mode and initialize it as 'L'. Nothing is returned.

`populate(cards, playMode)` – adds the `cards` to the bingo grid, starting with the top left corner and moving along the rows (left to right) until the lower right corner. All cards should be placed in the grid face up, except for the card in the middle of the grid. Update the play mode attribute according to the `playMode` input. You should assert that `cards` is a list containing enough Card objects to completely fill the 5 by 5 bingo grid. You should assert that playMode is 'L', 'C', or 'F'. Nothing is returned.

`search(card, rankOnly)` – performs a sequential search though all cards in the bingo grid, looking for a match to the input `card`. If a match is found, the card in the bingo grid must be turned face down and True is returned; if no match is found, the bingo grid is not updated and False is returned. When checking for a match, both the rank and suit must match if `rankOnly` is False; but only the rank must match if `rankOnly` is True. This means that if `rankOnly` is False, at most one card will be a match, but if `rankOnly` is True, there may be as many as four matches in the same bingo grid. You should assert that both `card` and `rankOnly` are the correct types.

`isBingo()` – returns True if the face down cards in the bingo grid match the pattern specified by the play mode; returns False otherwise. If the play mode is 'L', the face down cards must form at least one line of 5 face down cards, where the line can be horizontal, vertical, or diagonal. If the play mode is 'C', there must be a face down card in all four corners of the bingo grid. If the play mode is 'F', all cards in the bingo grid must be face down.

`clear()` – removes all cards from the bingo grid and returns them in a list.

`__str__()` – returns the string representation of the Bingo instance. Specifically, it includes the cards (formatted as shown below, each centered in a field width of 6 places) and 'BINGO!' centered under the grid if the face down cards match the pattern specified by the play mode. For example, the string representation for the sample BINGO grid shown on the first page of this assignment description would look like the following if printed:

```
[ KS ][ AH ][ 6C ][ 5H ][ KC ]
[ TD ][ 7H ][ 9C ][ 8S ][ 4C ]
[ 2D ][ AC ] [ ]  [ 4S ][ 4D ]
[ 5D ][ 3D ][ TC ][ 8C ][ JC ]
[ 9H ][ 5C ][ 7C ][ 5S ][ JS ]
```

If the King of Spades, 7 of Hearts, 8 of Clubs, and Jack of Spades were face down, and the play mode was 'L':

```
 [ ]  [ AH ][ 6C ][ 5H ][ KC ]
[ TD ] [ ]  [ 9C ][ 8S ][ 4C ]
[ 2D ][ AC ] [ ]  [ 4S ][ 4D ]
[ 5D ][ 3D ][ TC ] [ ]  [ JC ]
[ 9H ][ 5C ][ 7C ][ 5S ] [ ]
            BINGO!
```

Be sure to test this Bingo class thoroughly before moving on. Place these tests in a new Python file called *bingoTests.py* so the marker can see your tests.

Task 4: Table class

Create and test a Table class in *bingo.py*, according to the description below. Note that the following methods form the public interface for this class – you must complete them all as specified, and cannot add additional methods to the public interface. However, you can include additional private helper methods that are only called within this class, if you wish.

`Table()` – creates a new empty BINGO table. This table should have two empty decks: one to create bingo grids from, and another to call cards from. The table should also have two empty discard piles: one which holds all of the cards from the bingo deck when no longer in use, and another to hold the calling cards after they have been called. The table should also have two empty bingo grids: one for Player A, and another for Player B. All of these should be stored in private attributes (you should have six in total). Nothing is returned.

`populateDeck(deckNumber, filename)` – uses the contents of `filename` to populate a deck with cards; if `deckNumber` is 0, the bingo deck should be populated, if `deckNumber` is 1, the calling deck should be populated. You can assume that filename is a text file that has a 2-character string on each line. However, you cannot assume that every string represents a valid card – invalid cards should not be added to the deck (and an explanatory message displayed on the screen). After adding all valid cards specified in the text file, you should raise an Exception with the argument 'Deck created from xx is not valid' (where xx is replaced with the filename) if the resulting deck is not complete with 52 unique cards. At the beginning of this method, you should also assert that `deckNumber` is a valid integer, and `filename` is a string. Nothing is returned.

`dealBingo(mode)` – deals cards from the bingo deck to the two bingo grids, starting with player A and alternating between the two players until both bingo grids are full. The remaining cards in the bingo deck should then be placed in the bingo discard pile. Propagate any exceptions that may have been raised by another class. Nothing is returned.

`displayTable()` – displays Player A's bingo grid and Player B's bingo grid on the screen, as shown in the provided sample output. Nothing is returned.

`callCard()` – deals the top card from the calling deck and adds it to the calling discard pile. Returns the top card that was dealt.

`updateTable(card, rankOnly)` – searches both players' bingo grids for the card, and updates both grids as appropriate. You should assert that `card` and `rankOnly` are the correct types. Nothing is returned.

`clearTable()` – removes cards from both players' bingo grids and places the cards in the bingo discard pile. Also removes any cards left in the calling deck and places them in the calling discard pile. Nothing is returned.

`resetDecks()` – shuffles the cards in the bingo discard pile and repopulates the bingo deck. Also shuffles the cards in the calling discard pile and repopulates the calling deck. Nothing is returned.

`isWinner()` – returns True if at least one of the players has a BINGO! Returns False otherwise.

`whoWon()` – returns a string indicating who won. If player A wins, return 'Player A won!' If player B wins, return 'Player B won!'. If both players get a BINGO! at the same time, return 'Tie!' If neither player has won, raise an Exception with the argument 'Neither player has won'.

Be sure to test this Table class thoroughly before moving on. Place these tests in *bingoTests.py* so the marker can see your tests.

Task 5: main program

Create an instance of your Table class, and prompt the user to enter the filenames to populate first the bingo deck and then the calling deck. If an OSError is raised while trying to populate a deck, re-prompt the user for another filename.

Using your Table object, play a round of BINGO according to the rules defined above. At the beginning of the round, prompt the user to enter the play mode for that round: either (L)ine, (C)orners, or (F)ull. Continue to re-prompt until the first character of the user's entry is 'L', 'C', or 'F' (upper or lowercase). Also ask the user if they would like to play a speed round (where only the card ranks are compared). Continue to re-prompt until the first character of the user's entry is 'Y' or 'N' (uppercase or lowercase), meaning yes or no respectively.

During the round, display the round number, deal the bingo grids for the two players and display the starting state of the table. Continue to call a card from the calling deck (displaying the updated bingo grids each time) until one or both players have a BINGO!, and display the winner of the round. If any exceptions are raised during the round, handle them by ending the round and displaying the exception's argument along with the message 'Cannot continue with this round.'

At the end of the round (regardless of why it was ended), clear the table and reset both deck. Then ask the user if they would like to play another round. Continue to re-prompt until the user enters y/Y/n/N as the first character in their entry. Start a new round if the player enters y or Y; display a goodbye message and finish the program if the player enters n or N.

Sample Output

Refer to sampleOutput.txt files for formatting and exact message displays.

Assessment

In addition to making sure that your code runs properly, we will also check that you follow good programming practices. For example, divide the problem into smaller sub-problems, and write functions/methods to solve those sub-problems so that each function/method has a single purpose; use the most appropriate data structures for your algorithm; use concise but descriptive variable names; define constants instead of hardcoding literal values throughout your code; include meaningful comments to document your code, as well as docstrings for all methods and functions; and be sure to acknowledge any collaborators/references in a header comment at the top of your Python files.

Restrictions for this assignment are that you cannot use break/continue, and you cannot import any modules other than the `random` module, the `queues` module (from Lab 6), and your `playingCards` and `bingo` modules. Doing so will result in deductions.

Rubric:

- Code quality and adherence to specifications: 20%
- Card class and tests: 10%
- Deck class and tests: 10%
- Bingo class and tests: 20%
- Table class and tests: 20%
- Main BINGO game: 20%

Submission Instructions

- All of your code should be contained in **THREE** Python files: **playingCards.py**, **bingo.py**, and **bingoTests.py**.
- Make sure that you include your name (as author) in a header comment at the top of all files, along with an acknowledgement of any collaborators/references.
- Please submit your THREE python files via eClass before the due date/time.
- Do not include any other files in your submission.
- Note that late submissions ***will not be accepted***. You can make as many submissions as you would like before the deadline – only your last submission will be marked. So submit early, and submit often.

REMINDER: Plagiarism will be checked for

Just a reminder that, as with all submitted assessments in this course, we use automated tools to search for plagiarism. In case there is any doubt, you **CANNOT** post this assignment (in whole or in part) on a website like Chegg, Coursehero, StackOverflow or something similar and ask for someone else to solve this problem (in whole or in part) for you. Similarly, you cannot search for and copy answers that you find already posted on the Internet. You cannot copy someone else's solution, regardless of whether you found that solution online, or if it was provided to you by a person you know. **YOU MUST SUBMIT YOUR OWN WORK.**