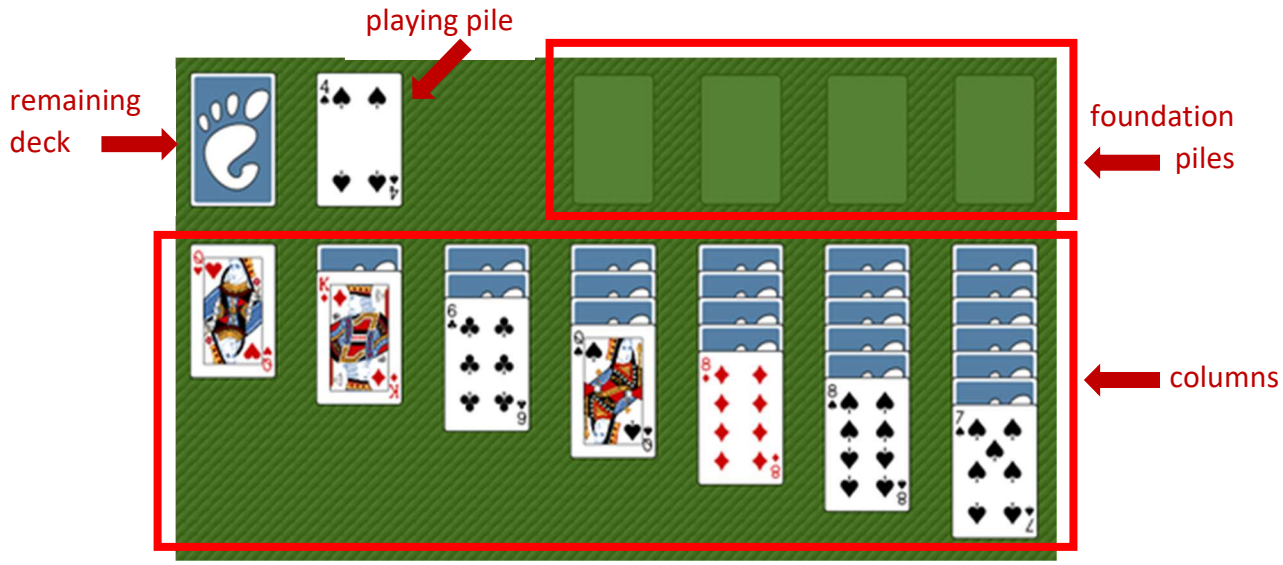
Assignment 3 – (Rigged) Solitaire

Due Date: Monday, December 7th 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 version of the card game Klondike Solitaire, played with a standard deck of 52 playing cards. But your version of this game will have a twist: you will rig it so that you always win.

Solitaire (https://en.wikipedia.org/wiki/Klondike_(solitaire)) is a one player game. You start by dealing cards from your complete deck into seven columns, as shown in the image below. Column 0 has 1 card, column 1 has 2 cards, column 2 has 3 cards, etc., and all cards are dealt face down, except for the card at the bottom of each column. The remaining deck is saved, and will be used to deal cards (3 at a time) face up onto our playing pile.



(Original image taken from Wikipedia)

The object of the game is to move all cards to the foundation piles, where each foundation pile holds cards of one suit only. In our game, the foundation piles will hold (from left to right) the clubs, the hearts, the spades, and the diamonds. In addition, the cards must be placed on the foundation piles in order of increasing rank: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King. Once all 52 cards are on the foundation piles, the game is won!

Rules for moving cards (simplified for our rigged version):

1. Only cards that are face up may be moved.
2. The card on the top of the playing pile may be moved to its suit's foundation pile, if it is next in rank.

3. The card at the very bottom of a column in the board may be moved to its suit's foundation pile, if it is next in rank. If this results in a face down card being revealed at the bottom of the column where the card was moved from, that bottom card will be turned face up.

There are additional rules for moving cards in the full version of Solitaire, but we won't need to know about them in our rigged version of the game.

In order to rig the game, we will sort our deck of cards before dealing them into the columns of the board. From the bottom card of the deck to the top card of the deck, it will be sorted in increasing order, where the order of the ranks is: Ace < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < Jack < Queen < King. The suits also have an order: Diamonds < Spades < Hearts < Clubs. Therefore, after the deck is sorted, the King of Clubs is the top card of our deck (and the first to be dealt), followed by the King of Hearts, then the King of Spades, then the King of Diamonds, then the Queen of Clubs, then the Queen of Hearts, ... and so on until the Ace of Diamonds at the bottom of the deck.

Since the game is rigged, you will be able to implement it so that it looks like the computer is playing the game (and winning!) all by itself. To implement this game, you will need to complete the following four Classes and the main program.

Task 1: CardNode class

Download the Python file called *riggedSolitaire.py* from eClass. Inside this file, you will find a CardNode class that has been partially implemented for you. The CardNode is very much like a Doubly Linked List Node, in that it has a reference to a previous and a next. However, instead of storing general data, it stores information specific to a card: its rank, its suit, and whether it's face up or down. The methods that have been implemented for you should look similar to what we've seen in the lecture notes for a Doubly Linked List Node and the Card class you created in Assignment 2. But there are 3 new methods that you must implement:

`__lt__(anotherCardNode)` – checks to see if the CardNode instance is less than anotherCardNode, based on both its rank and suit. Returns a Boolean value. Note that this is a special method, and will be called when the less than operator (<) is used between two CardNode objects. For examples of the logic, (King of Spades < 10 of Hearts) will return False; but (King of Spades < King of Clubs) will return True.

`__gt__(anotherCardNode)` – checks to see if the CardNode instance is greater than anotherCardNode, based on both its rank and suit. Returns a Boolean value. Note that this is a special method, and will be called when the greater than operator (>) is used between two CardNode objects.

`isPreviousRank(anotherCardNode)` – returns True if the rank of the CardNode instance is one less than the rank of anotherCardNode; False otherwise. For example, if isPreviousRank(King of Hearts) is called on the Queen of Diamonds, it will return True.

Do not change the methods that have already been completed for you. Do not change the method signatures of the three methods that you have to complete (i.e. you must use the input and return values as specified).

Test your CardNode class thoroughly before moving on. You may place these tests in the function `testCardNode()` already started for you, or write additional tests under `if __name__ == "__main__":` for the marker to see.

Task 2: CardList class

Inside *riggedSolitaire.py*, you will also find a CardList class that has been partially implemented for you. The CardList is very much like a Doubly Linked List, except that it contains the specialized CardNodes in its sequence. The getter methods, along with the add and append methods have been completed for you – do not change these. But there are 3 additional methods which you must implement:

`pop()` – removes and returns the last (i.e. tail) CardNode in the CardList's sequence.

`sort()` – sort the CardNodes in the CardList's sequence in increasing order, using INSERTION SORT. Increasing order means that the smallest card (considering both rank and suit) will be at the head and the largest card will be at the tail.

`__str__()` – returns the string representation of all of the CardNodes in the CardList, from the head to the tail. Note that the list starts with the vertical bar character ("|"), is followed by a single space, and then contains the string representation of each CardNode separated by a single space, followed by a single space and a final vertical bar character.

Do not change the method signatures of the three methods that you have to complete (i.e. you must use the input and return values as specified). You may create additional helper methods if you wish, but these cannot be called outside of the CardList class.

Test your CardList class thoroughly before moving on. You may place these tests in the function `testCardList()` already started for you, or write additional tests under `if __name__ == "__main__":` for the marker to see.

Task 3: CardStack class

Inside *riggedSolitaire.py*, you will also find a CardStack class whose `__init__` method has been completed for you. The CardStack is a special kind of Stack, and as you can see, is implemented using a CardList. Keeping that in mind, you must implement the following 5 methods:

`push(card)` – pushes the card onto the top of the CardStack if the CardStack is empty, or if the CardNode currently on the top of the CardStack is one less in rank than the rank of `card`. You must decide which end of your CardList should represent the top of your CardStack, based on which methods are available in the CardList public interface. Returns True if card was pushed onto the CardStack, False if it wasn't.

`pop()` – removes and returns the top CardNode in the CardStack.

`peak()` – returns the top CardNode in the CardStack, but does not modify the contents of the CardStack.

`isEmpty()` – returns True if there are no CardNodes in the CardStack, False otherwise.

`__str__()` – returns the string representation of the CardStack instance. Specifically, if the CardStack is empty, it should return two dashes ("-- "). Otherwise, it should return the string representation of the top CardNode.

Do not change the method signatures of the methods that you have to complete (i.e. you must use the input and return values as specified). You may create additional helper methods if you wish, but these cannot be called outside of the CardStack class.

Test your CardStack class thoroughly before moving on. You may place these tests in the function `testCardStack()` already started for you, or write additional tests under `if __name__ == "__main__":` for the marker to see.

Task 4: Table class

Inside *riggedSolitaire.py*, you will also find a Table class whose `__init__` method has been completed for you. The Table contains a deck (which is populated with CardNodes in the `populateDeck(filename)` already completed for you), a playing pile, four foundation piles, and the 7 columns. You must implement the following 7 methods:

`rigGame()` – rigs the game by sorting the deck. Nothing is returned.

`dealGame()` – deals the columns of cards from the full deck. The deck's top card is dealt face up in column 0, and the next cards are dealt face down in the remaining columns – that completes the first row. The next card is dealt face up in column 1, and the next cards are dealt face down in the remaining columns – that completes the second row. Cards are dealt to complete the remaining rows in a similar fashion, until the columns have the same number of cards as shown in the image on the first page of this assignment specification. Nothing is returned.

`drawThree()` – deal three cards from the top of the deck, and place them face up on the playing pile. Nothing is returned.

`playPileToFoundation()` – attempt to move the card from the top of the playing pile to the appropriate foundation pile (i.e. according to the card's suit). If the card was successfully moved to its foundation pile, return True. If the card cannot be moved to the foundation pile, move it back to the playing pile and return False.

`columnToFoundation(fromIndex)` – attempt to move the card from the bottom of the column specified by `fromIndex` to the appropriate foundation pile (i.e. according to the card's suit). If the

card was successfully moved to its foundation pile, return True. Also check to see if the column should have its new bottom card turned face up. If the card cannot be moved to the foundation pile, move it back to its column and return False.

`displayTable()` – display the top card in the four foundation piles (clubs, then hearts, then spades, then diamonds) on the screen. Display all 7 columns of the board, but lay out the columns as shown in the *sampleOutput.txt* rather than as shown in the picture on the first page of this assignment specification. Display the top card of the playing pile. Nothing is returned.

`gameWon()` – return True if all 52 cards are in the foundation piles; False otherwise.

Do not change the method signatures of the methods that you have to complete (i.e. you must use the input and return values as specified). You may create additional helper methods if you wish, but these cannot be called outside of the Table class.

Test your Table class thoroughly before moving on. You may place these tests in the function `testTable()` already started for you, or write additional tests under `if __name__ == "__main__":` for the marker to see.

Task 5: main program

Create a file *assignment3.py* and import the Table class. Create an instance of your Table class, and prompt the user to enter the filename to populate the deck. If an OSError is raised while trying to populate a deck, re-prompt the user for another filename.

Using your Table object, set up a game of rigged Solitaire. Then have your program automatically execute the following steps in order to win the game (no user input required):

1. Draw 3 cards at a time to place in the playing pile, until there are no cards remaining in the deck.

2. You should now be able to move cards from the playing pile, one at a time, to the appropriate foundation pile.

3. Once the playing pile is empty, you should be able to move cards from the columns to the foundation piles.

When the game has been won, display "WINNER!" and a goodbye message.

Sample Output

Refer to sampleOutput.txt file 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 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 `riggedSolitaire` module. Doing so will result in deductions.

Rubric:

A detailed rubric will be released before the deadline.

Submission Instructions

- All of your code should be contained in **TWO** Python files: **riggedSolitaire.py**, and **assignment3.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 TWO 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 to this problem 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.**