COMP1005/1405 – Winter 2022
Assignment # 3
Due on Friday, March 11th by 11:59 pm.

Submission Guidelines:

Submit a single zip file called A3.zip on Brightspace.

Did you zip it correctly? You must always create the zip files using the built-in, default, archiving program available with your operating system. If we cannot easily open your zip file and extract the python files from them, we cannot grade your assignment. Other file formats, such as rar, 7zip, etc., will not be accepted.

Windows: Highlight (select with ctrl-click) all of your files for submission. Right-click and select "Send to" and then "Compressed (zipped) folder". Change the name of the new folder "A3.zip".

MacOS: Highlight (select with shift-click) all of your files for submission in Finder. Right-click on one of the files and select "compress N items…" where N is the number of files you have selected. Rename the "Archive.zip" file "A3.zip".

Linux: use the zip program.

Did you submit the correct files? You are responsible for ensuring that you have submitted the correct file(s), and the submission is completed successfully. After submitting your A3.zip file to Brightspace, be sure that you download it and then unzip it to make sure that you submitted the correct files. This also checks that your zip file is not corrupted and can be unzipped.

You can submit as many times as you wish and Brightspace will save your latest submission. I would suggest that you submit as soon as you have one question done and keep re-submitting each time you add another problem (or partial problem).

We will not accept excuses like "I accidentally submitted the wrong files" or "I didn't know that the zip file was corrupt" or "My Internet was down" or "My computer was not working" after the due date.

If you are having issues with submission, contact the instructor before the due date.

Late Policy

The assignment is due on Friday, March 11th, 2022 by 11:59 PM (Ottawa time). Late submissions are allowed until Sunday, March 13th, 2022, 11:59 PM (Ottawa time) without any penalty.

Multiple submissions are allowed and Brightspace will only keep your LAST submission. The submission link will close after Sunday, March 13th, 2022.

Here is a summary of the penalties based on the submission time.

Last submission Day/Time	**Penalty applied**
Friday, March 11th, 2022 by 11:59 PM (Ottawa time)	No penalty
Sunday, March 13th, 2022, 11:59 PM (Ottawa time)	No penalty
Monday, March 14th or later	100%

Regret Clause

If you think you have committed an academic violation (plagiarism), you have 12 hours after the final allowable submission time (11 am on Monday, March 14th, for this assignment) to invoke this regret clause.

If you do, you will receive ZERO for the assignment (or portions of it that you invoke the clause for) but will not receive any further sanctions from the Dean's office. Your assignment will still be used when we are checking for plagiarism, but you would not face any further sanctions if it was decided that you had committed an academic violation.

For example, if you collaborated too closely with another student and this is discovered, you would not be further sanctioned. The other student, if they did not also invoke this clause, may still be further sanctioned by the Dean's office.

This is not a group project, so collaboration is not allowed on this assignment.

Self Evaluation Quiz Criteria

SE Quiz 3 is a follow-up quiz for Assignment-3. After you submit this assignment, you will be able to take the quiz on Brightspace.

You will be asked some leading questions to help you assess the problem-solving skills that you might have achieved by doing this assignment, such as:

- Do you feel more comfortable with programming terminologies?
- Did you draw a flowchart or write a pseudocode to help understand an algorithm?
- Did you try to break your problem into smaller sub-problems and solve each sub-problem separately?
- Did you use examples to help understand a problem?
- Could you explain the problems in your own words to someone else in this class?
- Did you find similar or related statements that we've talked about in lecture or tutorial, and did you try to understand how the problems were similar and how they were different?
- Did you ask a question on discord that required you to communicate what you understood and what was still confusing?
- Did you answer a peer's question on discord?
- Do you feel more comfortable using Python data structures like lists and dictionaries?
- Do you feel more comfortable using Functions, scopes, local/global variables in Python?
- Did you trace your code/algorithm and make sure it follows correctly and produce correct results?
- Did you try to identify the stages of the programming life cycle while you were solving a problem for this assignment?
- Do you feel more comfortable with identifying the errors in your code and handling those properly?
- For the given assignment, which letter grade would you give yourself (A+, …, F)?
- In 2-3 sentences, justify your letter grade selection.

Note: only the last two questions are mandatory for this quiz.

Topics: Functions, Data Structures: Lists

In this assignment, you will use Python's data structures to model a board game and write several functions to design and create this (text-based) game. Download the Python file called **a3.py** from brightspace. You are expected to add several functions to this file that are required to play this game.

Please be sure to read through the entire assignment before you begin to design your algorithm. There will be plenty of opportunities in this assignment to apply the decomposition and pattern recognition techniques. If you apply these techniques correctly, the complexity of this assignment problem will reduce significantly. So be sure to make a plan and design your algorithm before you begin to add code to the python file. You are required to present your plan/algorithm if you seek any help from the TAs or the instructor.

The game:

This game is a fusion between two very popular games; the tic-tac-toe (a board game) and the rummy (a card game). Tic-tac-toe is a well-known game that requires two players to enter some symbols in the cells of a board/table/two-dimensional list and match them either horizontally, vertically, or diagonally. Rummy is a card game that can also be modeled using a two-dimensional list and two or more players must match suits and/or values of their cards to win this game. You can check these links to find more information about these games; https://en.wikipedia.org/wiki/Tic-tac-toe and https://bicyclecards.com/how-to-play/rummy-rum/ for more information. Ours will be a single-player game. Let us see how our fusion game works.

1. The deck: We will use a deck of 32 cards, where each card will be simulated with a pair of values in this format [suit, face]. A suit is a string taken from this list ["SP", "CL", "HR", "DM"]. A value is an integer in the range [2, 9], inclusive. Examples of some valid cards are ["SP", 5], ["CL", 9], ["HR", 2], and ["DM", 7]. Any other combination of these values and values outside the given ranges are invalid, for example, ["SP", "DM"], ["SP", 5, 6], [7, "DM"] and ["CLUBS", 2] are all invalid. You'll find a function called create_deck() in a3.py that creates this deck of cards.

2. The board: This game will be played using a 3 x 3 table. I'll call it the *board* in this assignment specification. You can use any variable name for this list in your code. Note that '3 x 3' indicates that the board (table) always has 3 rows and 3 columns. Each cell on this board stores one card. The board will be displayed to the user in the following format, see examples below.

Three sample boards		
| DM,2 | SP,7 | CL,6 | | HR,2 | CL,2 | SP,3 | | DM,3 | SP,5 | DM,8 |	| SP,5 | HR,5 | DM,7 | | SP,4 | SP,3 | SP,2 | | DM,8 | HR,7 | HR,3 |	| CL,4 | DM,4 | CL,3 | | HR,5 | CL,6 | SP,6 | | DM,5 | CL,7 | HR,4 |

You will use a two-dimensional list (2D-list) to represent this board. The additional '|' symbols are not part of this 2D-list but they must be printed in this format whenever the board is displayed to the terminal. Note that, the square brackets '[' and ']' have not been printed.

3. The objective of the game: the user wants to finish the game by scoring the maximum points possible. To score points in this game the user needs to match the cards on the board in at least one of the following four categories.

- **Simple set, 10 points**: all cards on a row/column/diagonal on the board must have the same suit (can have different face values). Examples of two simple sets are ["CL", 8], ["CL", 5], ["CL", 7] and ["HR", 8], ["HR", 3], ["HR", 6].
- **Set, 15 points**: all cards on a row/column/diagonal on the board must have the same face value (can have different suits). Two sample sets can be as follows ["HR", 3], ["DM", 3], ["SP", 3] and ["DM", 8], ["SP", 8], ["CL", 8].
- **Simple run, 20 points**: all cards on a row/column/diagonal on the board must have consecutive face values either in the increasing or decreasing order (can have different suits). For example ["SP", 2], ["DM", 3], ["CL", 4] and ["CL", 6], ["CL", 5], ["DM", 4] are simple runs.
- **Run, 25 points**: Cards on a row/column/diagonal on the board must have the same suit and consecutive face values either in the increasing or decreasing order. Examples of runs are ["SP", 2], ["SP", 3], ["SP", 4] and ["CL", 6], ["CL", 5], ["CL, 4].

4. Rules of the game: this game starts by creating a deck of cards, shuffling the deck (randomly rearranging the order of the cards), dealing the first nine (9) cards to the user (these cards will be stored on the board), and displaying the board to the terminal so that the user can see their cards. Refer to the following example that shows a sample deck, a shuffled deck, the sequence of the dealt cards on the board. You will see the index values that can be used to access these elements in the 2D-list.

```
Deck : [['SP', 2], ['SP', 3], ['SP', 4], ['SP', 5], ['SP', 6], ['SP', 7], ['SP', 8],
['SP', 9], ['CL', 2], ['CL', 3], ['CL', 4], ['CL', 5], ['CL', 6], ['CL', 7], ['CL',
8], ['CL', 9], ['HR', 2], ['HR', 3], ['HR', 4], ['HR', 5], ['HR', 6], ['HR', 7], ['HR',
```

```
8], ['HR', 9], ['DM', 2], ['DM', 3], ['DM', 4], ['DM', 5], ['DM', 6], ['DM', 7], ['DM',
8], ['DM', 9]]
```

Note that the deck has 32 cards. The shuffled deck (shown below) also has the same 32 cards arranged in some random order.

```
Deck (shuffled): [['CL', 3], ['CL', 6], ['SP', 7], ['HR', 6], ['DM', 3], ['DM', 2],
['DM', 5], ['SP', 2], ['HR', 8], ['HR', 2], ['CL', 7], ['SP', 5], ['HR', 5], ['HR',
7], ['CL', 5], ['SP', 4], ['SP', 6], ['HR', 3], ['CL', 8], ['DM', 7], ['DM', 8], ['SP',
9], ['DM', 6], ['HR', 4], ['SP', 8], ['CL', 4], ['CL', 2], ['DM', 9], ['HR', 9], ['CL',
9], ['DM', 4], ['SP', 3]]
```

After your program deals the first nine cards to the player, these cards will be removed from the shuffled deck and stored on the board as follows.

The order in which cards will be inserted in the board	This is the board with the cards taken from the shuffled deck	Index values to access each element (card) in this 2D-list (board)
| 1 | 2 | 3 | | 4 | 5 | 6 | | 7 | 8 | 9 |	|['CL',3]|['CL',6]|['SP',7]| |['HR',6]|['DM',3]|['DM',2]| |['DM',5]|['SP',2]|['HR',8]|	| 0,0 | 0,1 | 0,2 | | 1,0 | 1,1 | 1,2 | | 2,0 | 2,1 | 2,2 |

```
Rest of the cards on the shuffled deck: [['HR', 2], ['CL', 7], ['SP', 5], ['HR', 5],
['HR', 7], ['CL', 5], ['SP', 4], ['SP', 6], ['HR', 3], ['CL', 8], ['DM', 7], ['DM',
8], ['SP', 9], ['DM', 6], ['HR', 4], ['SP', 8], ['CL', 4], ['CL', 2], ['DM', 9], ['HR',
9], ['CL', 9], ['DM', 4], ['SP', 3]]

board[0][0] returns ['CL', 3]
board[1][2] returns ['DM', 2]
board[2][1] returns ['SP', 2]
```

- Your program's next step would be to repeatedly show two options to the user; 'Deal' and 'Done'. The user chooses 'Done' (case insensitive) to end the game. Your program must identify if the cards on the board are matched using at least one of the categories and then display if the user won and how many points they scored.
- If the user enters 'Deal' (case insensitive), they'll have a chance to take the next card from the deck and use it to replace one of the existing cards on the board to get one step closer to a match. To replace a card, the user must enter the valid index values (i.e., the row and column numbers) that represent a card on the board (see how the index values are used to access cards above). These two numbers can be entered either on the same line separated

by a space (see the sample output) or separately on two different lines, whichever option you prefer. Invalid inputs must be identified and the prompts must be repeated.

Note that after each iteration your program must show the updated board, the current score, and the number of cards left in the deck until the user decides to end the game (by entering 'Done') or the deck becomes empty. Invalid user inputs should always be met with suitable error messages and repeated prompts.

5. How to keep scores: the game starts with 0 points. With every 'Deal', the user loses 1 point. The user gets points if their cards get matched using one of the four categories mentioned in section **3 (The objective of the game)**. Their final score will be calculated by 'the points awarded by a matching category' minus 'the total points lost'. If more than one category can be matched (e.g., 'set' and 'simple run') then the category with the highest achievable points (i.e., 'simple run' in this case) will be considered as the matching category and that points will be used to calculate the final score.

Note that the player must end the game to be able to win. If the board has matching cards but the user enters 'Deal', they will not be declared as the winner.

6. When the game ends: if the player wins, your program must display a congratulatory message and the name of the matching category to the terminal. The final score will always be displayed.

7. Sample outputs: The following sample outputs show how this game can be played in different situations. User input is highlighted in blue for emphasis. The matching cards are highlighted in yellow on the final board. Note that the final board may satisfy multiple matching categories, only the category awarding the highest points is highlighted. Section 8 has information about the required functions that you will need to add to a3.py.

Output-1:

```
Welcome to the game!
|  HR,8 |  CL,2 |  HR,3 |
|  CL,4 |  DM,5 |  SP,6 |
|  CL,6 |  HR,7 |  SP,2 |
Score: 0, Deal or Done? >> deal
New card: ['DM', 4], enter location to replace card <row col>: 2 1
|  HR,8 |  CL,2 |  HR,3 |
|  CL,4 |  DM,5 |  SP,6 |
|  CL,6 |  DM,4 |  SP,2 |
Cards left to play: 19
Score: -1, Deal or Done? >> deal
New card: ['SP', 5], enter location to replace card <row col>: 0 2
```

```
|  HR,8 |  CL,2 |  SP,5 |
|  CL,4 |  DM,5 |  SP,6 |
|  CL,6 |  DM,4 |  SP,2 |
Cards left to play: 18
Score: -2, Deal or Done? >> done
Congrats!! You've got a Simple Run on the board, Score: 18
```

Output-2:

```
Welcome to the game!
|  CL,3 |  SP,9 |  DM,6 |
|  DM,7 |  CL,8 |  DM,8 |
|  CL,7 |  CL,2 |  DM,4 |
Score: 0, Deal or Done? >> done
Congrats!! You've got a Simple Set on the board, Score: 10
```

Output-3:

```
Welcome to the game!
|  DM,3 |  SP,6 |  SP,4 |
|  HR,5 |  DM,7 |  HR,2 |
|  CL,4 |  HR,3 |  CL,6 |
Score: 0, Deal or Done? >> none
Invalid choice. Score: 0, Deal or Done? >> deal
New card: ['DM', 9], enter location to replace card <row col>: -2 -6
New card: ['DM', 9], enter location to replace card <row col>: 4 9
New card: ['DM', 9], enter location to replace card <row col>: 1 2
|  DM,3 |  SP,6 |  SP,4 |
|  HR,5 |  DM,7 |  DM,9 |
|  CL,4 |  HR,3 |  CL,6 |
Cards left to play: 19
Score: -1, Deal or Done? >> done
Sorry, no match on the board. Score: -1
```

Output-4:

```
Welcome to the game!
|  HR,7 |  SP,5 |  HR,5 |
|  CL,3 |  HR,9 |  SP,7 |
|  DM,8 |  SP,6 |  HR,8 |
Score: 0, Deal or Done? >> deal
New card: ['DM', 6], enter location to replace card <row col>: 1 1
|  HR,7 |  SP,5 |  HR,5 |
|  CL,3 |  DM,6 |  SP,7 |
|  DM,8 |  SP,6 |  HR,8 |
Cards left to play: 19
Score: -1, Deal or Done? >> deal
New card: ['DM', 5], enter location to replace card <row col>: 0 0
|  DM,5 |  SP,5 |  HR,5 |
|  CL,3 |  DM,6 |  SP,7 |
|  DM,8 |  SP,6 |  HR,8 |
Cards left to play: 18
Score: -2, Deal or Done? >> done
Congrats!! You've got a Set on the board, Score: 13
```

Output-5:

```
Welcome to the game!
|  CL,9 |  SP,4 |  HR,9 |
|  DM,4 |  CL,2 |  DM,6 |
|  HR,8 |  HR,7 |  CL,8 |
New card: ['DM', 3], enter location to replace card <row col>: 0 0
|  DM,3 |  SP,4 |  HR,9 |
|  DM,4 |  CL,2 |  DM,6 |
|  HR,8 |  HR,7 |  CL,8 |
Cards left to play: 19
Score: -1, Deal or Done? >> deal
New card: ['CL', 7], enter location to replace card <row col>: 1 1
|  DM,3 |  SP,4 |  HR,9 |
|  DM,4 |  CL,7 |  DM,6 |
|  HR,8 |  HR,7 |  CL,8 |
Cards left to play: 18
Score: -2, Deal or Done? >> deal
New card: ['SP', 2], enter location to replace card <row col>: 0 1
|  DM,3 |  SP,2 |  HR,9 |
|  DM,4 |  CL,7 |  DM,6 |
|  HR,8 |  HR,7 |  CL,8 |
Cards left to play: 17
Score: -3, Deal or Done? >> deal
New card: ['CL', 6], enter location to replace card <row col>: 0 0
|  CL,6 |  SP,2 |  HR,9 |
|  DM,4 |  CL,7 |  DM,6 |
|  HR,8 |  HR,7 |  CL,8 |
Cards left to play: 16
Score: -4, Deal or Done? >> done
Congrats!! You've got a Run on the board, Score: 21
```

Note, I did not include one more option in the sample output to show that this game can also end when there are no more cards left in the deck to play.

8. Required Functions that you must add to the program

The python file **a3.py** contains the main function that must act as the starting point of your program. That means the main function should be used to call all other functions to start and execute all the required steps before ending the game. In addition, you'll find the following functions in a3.py.

- `create_deck()` does not take any input parameter. This function creates and returns a 2D-list (representation of a deck of cards), where each element in this list is a one-dimensional list [s, f] representing a card from the 's' suit that has an integer face value 'f'. (See section **1. The deck** for reference).

- `shuffle_deck()` takes a 2D-list (deck of cards) as the parameter and returns the shuffled deck. This function uses the shuffle() function from the random module to re-arrange the cards in the original deck in a random fashion. You can re-write this function, if you wish, to use a different technique to shuffle the cards.

You must write the following functions (not necessarily in this order) as they are the required components for this assignment. You are encouraged to create additional helper functions to organize your code better.

- `deal()` takes the shuffled deck of cards as the parameter and deals the first nine (9) cards to the player. That means you'll remove the first nine cards from the shuffled deck and store them in a 2D-list (board). Make sure to follow the order shown in section **4. Rules of the game** to store these cards.

- `print_board()` takes a 2D-list (similar to the board) and prints the elements of the board to the terminal in the format specified in section **2. The board**.

- `update_board()` takes the board, the new card, and the index (row, column) values of the old card that needs to be replaced as input parameters. This function replaces the old card with the new card and returns the updated board.

- `verify_matching()` takes the 2D-list representing the board and an integer representing the current score as the input parameters. This function checks if the current board satisfies at least one matching category and returns three values; (i) a boolean value that indicates if there was a match, (ii) a string representing the matching category that was satisfied by this board (hint: you can use a category for 'no match'), and (iii) the updated score (int) as described in section **5. How to keep scores**.

  Important hint: you may want to divide this problem into multiple smaller problems to check conditions for each matching category. You can write helper functions for those smaller problems and call them inside the verify_matching() function to check all winning conditions.

- `game()` takes a shuffled deck of cards, continues the steps of the game, and returns a boolean indicating if the user wins the game (returns True) or loses it (returns False), an integer representing the final score of the player, and a string representing the winning category.

- `main()`: complete this function so that the entire game is handled within it. Note that your program can have global constants but **NO global variables**. That means points will be deducted if you use any global variable in this assignment.

Hints:

- Recall variable scopes and how local variables are treated inside a function. While writing helper functions, make sure to know which parameters to pass to a function and what values to return from it so that your program does not need to use global variables to store any information.
- Apply decomposition and pattern recognition. A large portion of the game logic can be checked using very similar techniques. First, try to write the entire program using only the simplest matching category i.e., 'Simple Set'. Once you have a perfectly working code for this version of the game, you will find it a lot easier to incorporate all other matching categories in your program.

The marking scheme and rubric will be posted in the #assignment-faqs on Discord.

Some considerations

- Your output should be neatly formatted and all required information must be present in the output. As before, you can customize the format of the output including the greetings and error messages.
- Again, you cannot assume that the user will enter good/valid input. All invalid inputs must be handled by printing appropriate error messages and repeating the menu until a proper value is entered.
- Although this assignment gives you plenty of opportunities to practice working with data structures, the current version does not have much room to generalize the game conditions. If you are interested, you can add additional functionality to this game that allows a full/double deck(s) of 52 cards, multiple players, a board with variable size (some larger size boards will not work for all matching categories though), the face values of the cards to calculate the final score of the winner, etc.

A3.zip Submission Recap

You just need to submit a3.py with A3.zip

Invalid submission and penalty

- Submissions with an incorrect file name or format will receive a 10% penalty, with no exceptions. You are responsible for following the guidelines outlined in this assignment and the course outline.
- Add your name and id number on top of each file. If this info is not present, a 10% penalty will be applied.
- Submissions that crash (i.e., terminate with an error) on execution may receive a mark of zero (0).
- Your program must not use any python module that has not been discussed in class or this specification.
- Your program can use any string method for user input validation and any tool or concept that is discussed in class.
- Submissions that crash (i.e., terminate with an error) on execution may receive a mark of zero (0).