CSCI160 Computer Science I
Final Project

This project deals with the following topics

- lists
- being creative in creating a game strategy (aka having fun)

Restrictions: You may only use concepts and techniques that were presented in this class. This includes: loops, lists, strings, slicing, decision structures and functions. You may not use comprehensive “black box” Python functions that we have not explicitly covered, or list comprehensions. I am interested in you demonstrating that you understand the basic concepts of looping and processing an indexed structure, and Python has many functions that hide crucial logic details. This is not the point of a CS class. If you are unsure of what is allowed and what is not, **it is your responsibility to ask**. I will be strictly grading this.

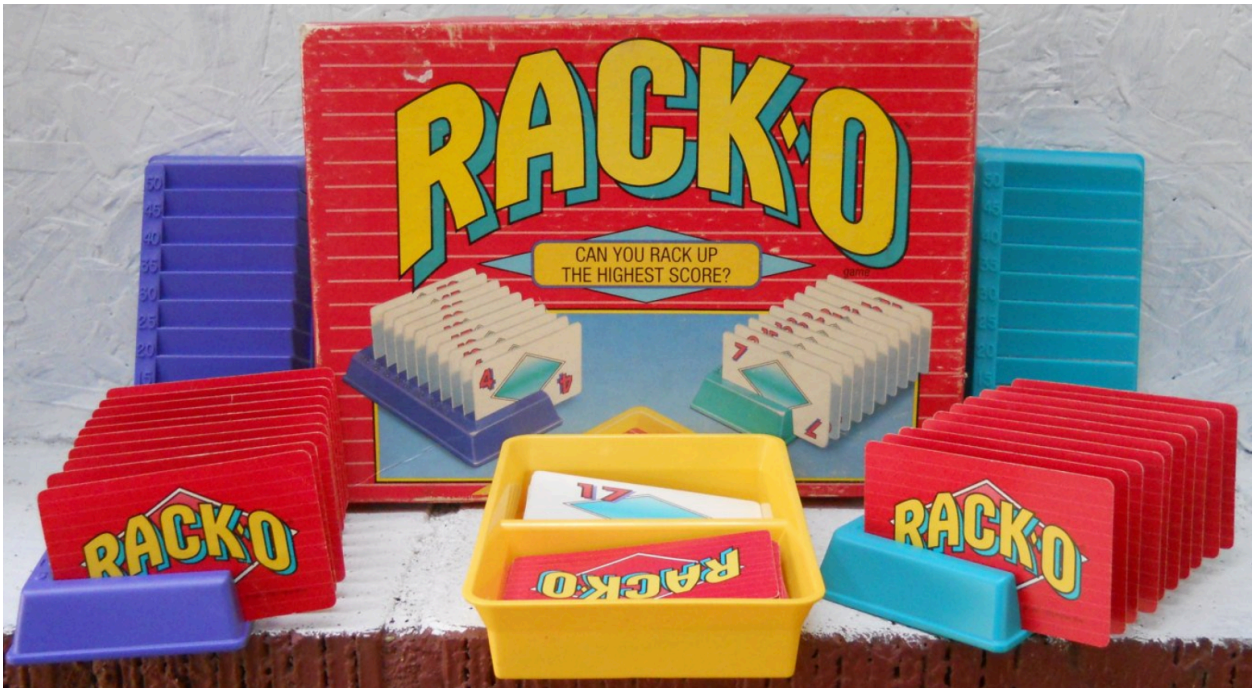
Examples of things not allowed

- sorted(), shuffle(), any(), all(), min(), max(), sort(), map(), list comprehensions

Things you may use

- list methods: append(), insert(), pop(), loop with range()

General Idea : This assignment is mainly intended to get you to practice explicit list modifications. We will be implementing the old card game Rack-O . This game involves



rearranging your hand of cards in order to have an increasing (some people go decreasing) sequence. While Rack-O is typically played with 2 to 4 players, we will keep this simple and just play the user versus the computer.

- The user's moves are decided by the user by having the program ask for input.
- The computer's moves are decided by the rules you program into the logic. No input is required.
- This means that there is NO right answer for the computer's strategy. Just come up with some reasonable enough strategy that ensures that a human user does not consistently beat the computer.

Rules of the game

- A Rack-O deck is composed of 60 cards, each numbered 1 to 60. The objective is to be the first player to arrange all of the cards in your rack from **lowest to highest**.
- To start the game, **shuffle the deck**, both the user and the computer pick a card from the deck. Let the computer pick first. Whomever draws the highest card goes first.
- The two drawn cards are added back onto the deck and the cards get **shuffled again.** Both the user and the computer gets dealt 10 cards. As a player receives each card, they **must** place it in the highest available slot in his rack, starting at slot 10, **without rearranging any of them**.
- The goal of each hand is to **create a sequence of numbers in ascending order, starting at slot 1.**

Game Play

- Shuffle the deck and determine who goes first
- Shuffle the deck again and deal two 10 card **racks**
- The top card of the deck is turned over to start the **discard pile**.
- The player who drew the highest card from the previous step takes a turn by taking the top card from **either the deck or the discard pile**, then discarding one from his rack and inserting the new card in its place.
    - If the player draws a card from the deck, they may immediately discard it
    - if they take the top card from the discard pile they **must put it into their rack**.
- The first player to get 10 cards in ascending order calls ”Rack-O!” and wins the hand.
- The user should be asked if they would like to play again. If so, start a new game. If not, say Good Bye and exit the game.

Note that we will make heavy use of lists in the assignment. Since a rack looks more like a vertically oriented structure, we need to agree upon a convention. For this assignment, the following approach is required.

lst = [3, 17, 11, 30, 33, 38, 49, 46, 25, 53]

is actually the following rack from slots 10 down to 1. It is important to notice the relationship between the slot # and the list index

RACK

Slot	Card
10	3
9	17
8	11
7	30
6	33
5	38
4	49
3	46
2	25
1	53

The **deck and discard pile** are also going to be represented as lists. Note that in both the deck and the discard pile, you only have access to the top.

- If you take a card from the pile or deck, you need to call the **pop()** method.
- If you add a card to a pile or deck you call the **append()** method.

Required function definitions: I expect to see these functions with these names and these signatures exactly as specified.

- **build_deck()** - create a “deck” of “cards”. This will be a list of integers from 1 - 60. **This function will return the list**

- **shuffle(deck)** - shuffle the deck to start the game. Also shuffle the discard pile if you ever run out of cards and need to ‘restart’ the game. This function does not return anything. You need to write the shuffle logic yourself . . . do not rely on Python to shuffle the deck for you. One approach would be to create 2 random numbers

between 1 and 60 and swap the cards at those locations. If you do this many times the deck will be “shuffled”. This function will not return anything as it operates on the list **in-place**

- **check_racko(rack)** - given a rack (this will be either the user’s or the computer’s) determine if Rack-O has been achieved. Remember that Rack-O means that the cards are in ascending order. **This function will return a boolean value representing the Rack-O status**
- **deal_card(deck)** - get the top card from any deck of cards. Use this function anytime a card needs to be dealt. **This function must return an integer.**
- **deal_hands(deck)** - start the game off by dealing two hands of 10 cards each. **This function returns a SINGLE list with TWO sub-lists of 10 cards representing a single hand.** The first returned list will be the user’s, the second list will be the computer’s. To reiterate . . . this function will return a single list containing both hands as sub lists**. Below is an example**

```
hands[
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    [12, 11, 13, 14, 3, 2, 1, 8, 9, 10]
]
hands[0] = user’s hand
hands[1] = computer’s hand
```

  - Make sure that you follow normal card game conventions of dealing. So you have to deal one card to the user, one to the computer, one to the user, one to the computer and so on.
  - The computer is always the first player to be dealt to.
  - Remember that the rules of our version of Rack-O will be that you have to place your cards in the order of top most slot of the rack first, then the next slot and so on. **This is required**
  - In the example above, this would mean that someone was dealt the cards 3, 17, 11, 30, 33, . . . in that order.
- **print_rack(rack)** - given a rack(represented as a list) print it out from top to bottom in a manner that looks more akin to the game. See the example above for the exact specification. Please stick to that specification in terms of the representation of the rack. **The display should be a vertical, two column display that shows the slot # and the card at that slot.**
- **find_and_replace(new_card, card_replaced, hand, discard)** - find the card to be replaced (represented by a number) in the hand and replace it with **new_card**. The replaced card then gets put on top of the discard. Check and make sure that the

card_replaced is truly a card in the hand. If the user accidentally typed a card number incorrectly, just politely remind them to try again and leave the hand unchanged.

- **discard(card, discard)** - add the card to the top of the discard pile.
- **computer_turn(hand, deck, discard_pile)** - This function will contain the computer's strategy. You are supposed to be somewhat creative here, but I do want your code to be deterministic. That is, given a **particular rack** and a **particular card** (either from the discard pile or as the top card of the deck), you either always take the card or always reject it.

  Here are some potential decisions the computer has to make

  1. Given the computer's current hand, do you want to take the top card on the discard or do you want to take the top card from the deck and see if that card is useful to you.
  2. How do you evaluate the usefulness of a card and in which position it should be placed?
  3. There might be some simple rules you can give the computer. For instance, it is disastrous to put something like a 5 in the top slot. You want big numbers toward that end of the rack.

  You are allowed to do pretty much anything in this function except make a random decision or make a decision that is obviously incorrect. For instance, making your bottom card a 60 is a recipe for disaster. Also, the computer CANNOT CHEAT. What does that mean? The computer cannot peek at the top card of the deck and then make a decision. Its decision making should be something that a human should be able to make as well.

  This function has to return the new rack for the computer.

- **main() -** a function that puts it all together. This function will contain all the game logic concerning switching turns and announcing a winner. You continually play Rack-O until the user states they are done. **All game level variables will be created inside this function. No module level variables are allowed. You need to show me that you can pass arguments into function and process the returns.**

There is a **basic structure** for main() below. The intent is that you fill in the required pieces of code once you understand our comments. Some details are intentionally left out.

You do not have to adhere to this structure of main(). You can use your own design as long as you stick to the rules of the game.

```
def main():

        #create all game variables here

         as long as the user wants to play
                #create the deck
                #create an empty discard pile
                #shuffle the deck
                #determine who goes first
                #deal the hands

                #print human rack

                #reveal one card from the deck to begin the discard pile

                while neither the computer nor the user has racko:
                        #Assuming computer goes first . . . computer turn
                        #does the computer have Rack-O?

                        #human turn . . .
                        #show rack and discard pile
                        #ask what they would like to do
                        #if they choose pull from deck
                                #print this card to show the user what they got
                                #ask the user if they want this card
                                if user chooses this card :
                                        #ask for the card(number) they want to kick out
                                        #modify the user's hand and the discard pile
                        #else they chose discard pile
                                #ask for the card(number) they want to kick out
                                #modify the user's hand and the discard pile
                        #print the user's hand
                        #does user have Rack-O?

                  #ask the user if they'd like to play again
```

Your computer should have enough intelligence to beat the 'stupid and lazy' user.

Also, remember the user has no idea what you are doing internally in your functions and does not want to be shown a large amount of print statements. At any given point in the

game, they should know only 3 things - their entire hand printed in the rack form, the top card of the discard and if they choose to dive into the deck they get to know the top card of the deck.

Include the following statement to run your main function when the program is executed.

```
if __name__ == '__main__':
    main()
```

The game must start immediately upon the program being executed. If I have to modify your code in any fashion points will be taken off.

Start this early and ask questions

Submit Racko.py and any other files you created. I will run this project by typing the following

>python Racko.py

If your game is not runnable in this fashion points will be taken off.