Programming Languages and Techniques

Homework 3

Due as per deadline on canvas

This homework deals with the following topics

* lists

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

General Idea

This assignment is mainly intended to get you to practice list modifications

We will be implementing the old game Racko which is a game that involves rearranging your hand of cards in order to have an increasing (some people go decreasing) sequence.

While Racko 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 asking for input, the computer's moves are decided by you the programmers! What that means is that there is NO right answer for the section that asks you to program a strategy for the computer. All we want you to do is come up with some reasonable enough strategy that ensures that a human user does not consistently beat the computer. So unlike your previous assignments, this one has a creative component to it as well.

Rules of the game

A Racko 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. The person who gets to play first is the person who has the higher number.

The cards then get shuffled again and both the user and the computer gets dealt 10 cards. As a player receives each card, he 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.

The top card of the deck is turned over to start the discard pile. A player 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, he may immediately discard it; if he takes the top discard, though, he must put it into his rack.

The first player to get his 10 cards in ascending order calls ”Rack-O!” and wins the hand.

While I do not have a link to an online Racko game, there is a very similar game (with little Vikings!) over here tower blaster

We also have the actual game with us (assuming I did not forget to bring it with me to recitation) and are more than happy to pass it around in class for you to get a feel for what the game is like.

Finally here is just a random picture of the game to remind yourself what it looks like (you can also stop by my office to take another look).

The actual program

Once again we provide you with stubs of functions. But this time around there is less guidance as to what exactly you need to put in them.

Here are the functions that need to be written. We are expecting to see these functions with these names and these method signatures.

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 have some convention. Our convention is that

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

is actually the following rack from slots 10 down to 1

```
3
17
11
30
33
38
49
46
25
53
```

So yes, this particular rack is a long way from a victory.

We know this is a somewhat awkward way to representing the data, but we are deliberately asking you to do it this way in order for you to do more list exercises.

The deck and discard 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, you need to call the pop function. If you add a card to a pile you call the append function.

- **shuffle(cardstack)** - shuffle the deck to start the game. Also shuffle the discard pile if we ever run out of cards and need to ‘restart’ the game. This function does not return anything. You can import the random module and just use random.shuffle.

- **check_racko(rack)** - given a rack (this will be either the user's or the computer's) determine if Racko has been achieved. Remember that Racko means that the cards are in ascending order. This function returns a boolean value.

- **get_top_card(card_stack)** - get the top card from any stack of cards. Used at the start of game play for dealing cards. This same function will also be used during each player's turn to take the top card from either the discard pile or from the deck.

  This function must return an integer.

- **deal_initial_hands(deck)** - start the game off by dealing two hands of 10 cards each. This function returns two lists - one representing the user's hand and the other representing the computer's hand.

  Make sure that you follow normal card game conventions of dealing. So programatically 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 person that gets dealt to. The user always plays first.

  Remember that the rules of our version of Racko 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.

  In the example above, this would mean that someone was dealt the cards 3, 17, 11, 30, 33, ... in that order.

  The method of returning 2 things from a function is to make a tuple out of the return values. For example, the following piece of code returns the sum and the maximum in a list. Note that a tuple is slightly different from a list. We will explain this in more detail in next lecture.

```
def get_sum_and_max(lst):
   return (sum(lst), max(lst))


def main():
   sum_and_max = get_sum_and_max(range(1, 10))
   print "max is", sum_and_max[1]
   print "sum is", sum_and_max[0]
```

- **print_top_to_bottom(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 (more stack like than list like). See the example above for the exact specification. Please stick to that specification in terms of the representation of the rack.

- **find_and_replace(new_card, card_to_be_replaced, hand, discard)** - find the card to be replaced (represented by a number) in the hand and replace it with newCard. The replaced card then gets put on top of the discard. Check and make sure that the cardToBeReplaced 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.

- **add_card_to_discard(card, discard)** - add the card(represented as just an integer) to the top of the discard pile.

- **computer_play(hand, deck, discard_pile)** - This function is where you can write the computer's strategy down. It is also the function where we are giving you very little guidance in terms of actual code.

  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 which position it should go into.
  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 over there.

  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 of going to the discard. It's decision making should be something that a human should be able to make as well.

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

- **main()** - a function that puts it all together. You play until someone gets Racko.

  We have a basic structure for main() in the next page. We deliberately have not written real code. The intent is that you fill in the required pieces of code once you understand our comments.

  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 a list of integers that represents a deck
    #create an empty discard pile
    shuffle(deck)
    hands = deal_initial_hands(deck)
    #hands is a tuple.
    #assign one element of this tuple as the user's hand
    #assign the other element of the tuple as computer's hand
    print_top_to_bottom(human_hand)
    #reveal one card to begin the discard pile
    while neither the computer nor the user has racko:
        computer_hand = computer_play(computer_hand, deck, discard)
        #ask the user if they want this card
        #print the user's hand
        if user chooses this card:
            #ask the user for the card(number) they want to kick out
            #modify the user's hand and the discard pile
            #print the user's hand
        elif choice == 'n':
            card = deck.pop()
            #print this card to show the user what they got
            #ask the user if they want this
            secondChoice = raw_input('keep it?\n')
            if secondChoice == 'y':
               #modify user's hand and discard pile
               # print user's hand
            else:
                discard.append(card)
                #print the user's hand
         #check and make sure there are still some cards in the deck
         #else reshuffle the discard and restart.
```

Evaluation

The primary goal of this assignment is to get you to feel familiar with lists and to have some level of fun while creating a game.

While we want you to spend time on coming up with some kind of strategy for the computer, that is NOT the primary part of the assignment. Come up with something reasonable. Any reasonable strategy will have you doing some fun things with lists. If the user always does absolutely nothing at all, that is, they reject the discard card and they reject the top card from the deck and move it onto the discard, then we want the computer to beat the user. 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.

Evaluation criteria

- 8 points for getting the game to work.
- 5 points for style - variable names, function names, modularity. The modularity part basically refers to breaking down the computer strategy into smaller functions. We do not want to see one massive function that deals with the computer's strategy.
- 4 points for getting the specifications correct (passing all the TAs tests). As long as you follow our specifications, these tests will pass. Remember to stick to the same function names. Make sure your functions (other than the main function) do not print any extra information. Make sure your functions return the datatype that we expect you to return.
- 3 points for strategy - we just need you to implement a reasonable strategy.

What data structures to use

In this HW you are only allowed to use lists (and tuples in the one place as shown above). Please do not use dictionaries, classes or any in built python libraries. Remember that the deck, the discard pile and the two hands (user and computer) are just lists.

what to submit

Submit one single file called Racko.py. This is still an individual assignment. We will do groups from the next HW.

Please put the following at the end. Now that you have seen it a few times, we will deduct a point if you do not have this at the end.

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