APS106 – Lab #5

Welcome to the second half of the APS106 labs. In the first half of this course, we focused on learning tools that could be used within your programs. Going forward in the second half, we will be more focused on *using* these tools to solve problems.

Because this lets us ask a wider range of questions, many of you will find the labs 5-9 much more challenging than labs 1-4.

We highly encourage you to read the labs early in the week and attend your practical and tutorial sessions. These are the best way for you to discuss and get help from your TAs. Remember, these labs are designed to challenge you and give you the opportunity to learn and your instructors and TAs are committed to providing you the support you along the way.

Preamble

This week you will practice using loops, lists, and conditionals to implement a simple card game.

Use appropriate variable names and place comments throughout your program.

The name of the source file must be "lab5.py".

Deliverables

For this lab, you must submit the three functions listed below within a single file named 'lab5.py' to MarkUS by the posted deadline.

Functions to implement for this lab:

1. Deal_card
2. Score_hand
3. Play

Two additional functions are provided within the starter code. You may use these functions to complete and test your code as needed. **You are not expected to modify these functions**.

- `generate_deck()`
- `shuffle(deck)`

Five test cases are provided on MarkUs to help you prepare your solution. **Passing all these test cases does not guarantee your code is correct.** You will need to develop your own test cases to verify your solution works correctly. Your programs will be graded using ten secret test cases. These test cases will be released after the assignment deadline.

IMPORTANT:

- Do not change the file name or function names
- Do not use input() inside your program

Problem

For this lab you will complete a program to play a very simplified version of the card game cribbage[1]. The game is played with a standard 52-card deck. During the game, two players are both dealt[2] five cards from the deck according to the rules (outlined in part 3). The player that has the hand[3] with the most points is the winner.

In this exercise, you will use nested lists to represent cards within the deck and within each player's hand. For example, the list:

```
[['diamonds', 7], ['hearts', 11], ['clubs', 8], ['spades', 5], ['spades', 1]]
```

would be used to represent a player with a hand containing the 7 of diamonds, the jack of hearts, the 8 of clubs, the 5 of spades, and the ace of spades. That is, each card is represented with a two-element list where the first element is a string specifying the suit and the second element is an integer representing the card value. The deck and player hands are lists containing these two-element card lists as elements.

For this lab, you will need to complete 3 functions:

- `deal_card(deck,hand)`
- `score_hand(hand)`
- `play(shuffled_deck)`

Both `deal_card` and `score_hand` are helper functions that you will call from within play to execute the game.

Additionally, you are provided two additional helper functions to help you complete and test your code:

- `generate_deck()`
- `shuffle(deck)`

You do not need to edit these functions. They are provided to help you test and complete your code. `generate_deck` will create and return a list representing a standard 52-card deck. Each element of the returned list will be a two-element list. The first element containing a string representing the suit (spades, clubs, diamonds, or hearts) and the second element will be an integer between 1 and 13 inclusive, representing the card value. Values 2-10 represent number cards, 1 represents an ace, 11 a jack, 12 a queen, and 13 a king. Calling the `shuffle` function will return a random permutation of the cards.

We will break down how to complete the different functions in the steps below.

---

[1] We'll provide all the details about the game that you need to complete the game

[2] Dealing a card refers to the act of removing the card currently at the top of the deck and assigning it to one player's hand.

[3] A player's hand is the collection of cards dealt to them during the game

Part 1: Deal Card

For the first part of the lab, you will complete the `deal_card` helper function. By completing this function, you will become more familiar with the concept of aliasing and why you need to be careful when modifying lists passed as inputs to functions!

This function takes two lists as arguments. The first input list represents the game's deck of cards and the second represents a player's hand. The function should remove the *first card* from the deck list and append it to the end of the list representing the player's hand.

Note this function should return `None`. That is, your function should not return either of the modified lists. This is weird, and you may be wondering how the modified lists get passed back to the code that called the function. The answer is that they do not need to be passed back and the reason for this lies in the aliasing property of lists. Aliasing is tricky and (spoiler alert!) this will be an important concept to understand in the remainder of APS106. After implementing the function, try to explain to yourself how and why this function works. Discuss it with your TAs and colleagues to see if your understanding is correct.

The following code snippet illustrates the behaviour of the function.

```
deck = [['spades',10],['hearts',2],['clubs',8]] # deck with 3 cards
player_hand = [['diamonds',3]] # list representing a player's hand,
currently with a single card

print("Deck and hand before deal_hand function call")
print("\tdeck : ",deck)
print("\thand : ",player_hand)
deal_card(deck,player_hand) # Notice no equals sign! Nothing assigned from
the function return
print("\nDeck and hand after deal_hand function call")
print("\tdeck : ",deck)
print("\thand : ",player_hand)
```

Printed Output

```
Deck and hand before deal_hand function call
     deck :  [['spades', 10], ['hearts', 2], ['clubs', 8]]
     hand :  [['diamonds', 3]]

Deck and hand after deal_hand function call
     deck :  [['hearts', 2], ['clubs', 8]]
     hand :  [['diamonds', 3], ['spades', 10]]
```

Notice that after the function call, the first element from the deck (the 10 of spades) list has been removed and has been appended to the end of the hand list. You may assume that the deck list will always have a minimum of one card.

Part 2: Score Hand

Is this part, you will complete the function `score_hand(hand)` which calculates the score for the list of five cards in a player's hand. The input parameter to this function is a list of five cards representing a player's hand. You may assume that the list input to the function will always contain five cards.

The score will be calculated according to the hand scoring rules of cribbage, with very slight modifications. The `score_hand` function should calculate points according to the following:

1. Each *pair* (i.e. same card value) scores **2 points**. If a hand contains three or four of a kind, 2 points are scored for every combination of pairs. So three of a kind scores 6 points and four of a kind scores 12 points.
2. If all five cards are the same *suit*, **5 points** are scored[4]
3. A group of three cards with consecutive values (called a *run* or *straight*) scores **three points**. The suit of the cards does not matter. A run of four consecutive values scores 4 points and a run of five consecutive values scores 5 points. Other points regarding runs:
    a. Points are scored for every unique set of cards that produces a run. For example, the hand: ace of hearts, ace of spaces, two of diamonds, three of hearts, and three of spades would score 12 points for runs because the run 1-3 can be constructed with four distinct sets of cards (note the total score for the hand would be 16 as there are also two pairs in the hand).
    b. For the purposes of defining runs aces, jacks, queens, and kings can be interpreted as the values 1, 11, 12, and 13, respectively.
    c. A run cannot wrap around from a king to an ace (i.e. A queen, king, and an ace would *not* be considered a run)
    d. Points are not awarded for shorter runs within a longer run. For example, in a run of four cards, no points would be awarded for the two runs of three within the hand.
4. All combinations of cards that sum to 15 are worth **2 points**. When summing card combinations, aces are counted as one and jacks, queens, and kings are counted as 10.

Sample test cases:

Hand	Points	Description
10 of hearts 2 of hearts 3 of hearts 6 of hearts 5 of diamonds	4	2 + 3 + 10 = 15 => 2 pts 5 + 10 = 15 => 2 pts

[4] For any experienced cribbage players out there, we will **ignore** the case where four points can be scored for a flush of four cards.

9 of hearts 2 of spades 3 of clubs 3 of hearts 4 of hearts	12	Pair of threes => 2 pts Run of 2, 3 (clubs), 4 => 3 pts Run of 2, 3 (hearts), 4 => 3 pts 2 + 4 + 9 = 15 => 2 pts 3 + 3 + 9 = 15 => 2 pts
5 of diamonds Queen of diamonds King of diamonds Ace of diamonds 5 of hearts	10	Pair of fives => 2 pts 5 (hearts) + 10 (queen) => 2 pts 5 (hearts) + 10 (king) => 2 pts 5 (diamonds) + 10 (queen) => 2 pts 5 (diamonds) + 10 (king) => 2 pts
3 of diamonds 4 of diamonds 5 of diamonds 6 of diamonds 7 of diamonds	14	Five of same suit => 5 pts Run of 3,4,5,6,7 => 5 pts 3 + 5 + 7 = 15 => 2 pts 4 + 5 + 6 = 15 => 2 pts

You can generate additional test cases using this website (http://www.bucktheodds.com/cribbage/score). Set the score hand button to 'Crib'. You can ignore the distinction between common and hand cards as well as any points awarded for Jacks and flushes of four cards.

This function is not trivial and will get very messy if you try to write the code without a clear algorithm plan. It is recommended you spend time trying to think about how to solve this before trying to code a solution.

Here a couple hints to help you get started:

1. Break the problem into smaller pieces. Try developing a solution for scoring pairs of cards first. Once you have a solution for this part of the problem, test it until you are confident it works. Then work on adding to your code to identify other sources of points.
2. One possible strategy for identifying pairs and groups of cards in the same suit is to build a histogram counting the number cards with a value or suit within the hand. You could use a list and a for loop to create the histogram.
3. The function `combinations` from the `itertools` module will generate all possible unique combinations of size C from the set of N elements within a list (combinations of N choose C). You can then convert this into a list of all combinations that you can iterate through using a for loop. You can experiment with this function using this snippet:

```
from itertools import combinations
lst = [1,2,3,4,5]
combos_of_two = list(combinations(lst, 2)) # returns list of all n choose 2 combinations
combos_of_three = list(combinations(lst, 3)) # returns list of all n choose 3 combinations
```

Note that we wrap the return of `combinations` in `list()` because the value returned from combinations is something slightly different from a list which you have not learned about in APS106.

Part 3: Implement the Game

For the final part of the lab, you will need to complete the `play` function. This function should utilize both the functions you completed in parts 1 and 2. The rules for playing the game are as follows:

1. Deal 5 cards to both players in alternating order. That is, player 1 gets the first, third, fifth, seventh, and ninth card in the deck and player2 gets the second, fourth, sixth, eighth, and tenth card from the deck.
2. Calculate the score for both players
3. The player with the high score wins the game. If the scored is tied, neither player wins.

The function should return a three-element list. The first element should be one of the three following strings identifying the outcome of the game:

- `"player1 wins"`
- `"player2 wins"`
- `"tie"`

The second element should be the score of player 1 and the third element should be the score of player 2.

Examples:

If player1 has a score of 14 and player 2 has a score of 5, the list returned from the function should be:

```
['player1 wins', 14, 5]
```

If player 1 has a score of 2 and player 2 has a score of 10, the list returned from the function should be:

```
['player2 wins', 2, 10]
```

If player 1 has a score of 4 and player 2 has a score of 2, the list returned from the function should be:

```
['tie', 2, 2]
```

You can test your code with randomly shuffled decks using the following snippet of code

```
deck = generate_deck()
deck = shuffle(deck)
print("deck: ",deck[:10]) # just print the first cards in the deck
```

```
result = play(deck)
print("result: ",result)
```

Tips for developing test cases and testing part 3

Because the above code snippet uses a randomly generated deck to test the `play` function, it is not ideal for debugging your code when you do find an error. This is because the code won't produce the same deck multiple times in a row and therefore may not result in the same error. There are two potential options to overcome this and test your code using consistent decks.

1. Hardcode your own 10 card deck and pass this to the play function rather than using the `generate_deck` and `shuffle` functions. This will let you develop explicit test cases to ensure your function behaves properly.
2. You can use utilize the `seed` function from the `random` module with different input arguments. If you call this function prior to calling the `shuffle` function, `shuffle` will always return the same randomly shuffled deck. You can generate different random decks by changing the input argument to the `seed` function.

It is recommended that you use approach 1 to develop a few specific test cases first. After testing these test cases, you can use approach 2 to further test your code. Examples of both approaches are given below.

Example of test approach 1 – Hardcode a deck

```
deck = [['diamonds', 7], ['hearts', 6], ['clubs', 8], ['spades', 5],
['spades', 1], ['hearts', 1], ['spades', 6], ['diamonds', 3], ['spades', 10],
['hearts', 11]]
print("deck: ",deck[:10]) # just print the first cards in the deck
result = play(deck)
print("result: ",result)
```

Example of test approach 2 – Use seed to generate consistent deck

```
random.seed(3) # change '3' to different integers to produce different
shuffled deck
deck = generate_deck()
deck = shuffle(deck)
print("deck: ",deck[:10]) # just print the first cards in the deck
result = play(deck)
print("result: ",result)
```