

CSCI160 Computer Science I

Simplified Terminal Poker

Worth 100 Points and is 20% of your overall grade

Objectives:

- Apply simple data structures to a common problem: strings, lists, dictionaries, tuples
- Iterate over sequential data structures
- Write functions that accept, mutate and return new structures
- Practice functional decomposition and documentation
- Write legible, concise code with proper documentation
- Use debugging tools
- **Following instructions closely ← MOST IMPORTANT!!** Your adherence to the instructions will be graded. You cannot simply use some random code you found on the Internet. If we have not covered you cannot use it.
- Any additional features and approaches you want to add must be approved

Important Concepts:

- Basic Data Structures: **String, List, Tuple, Dictionary**
- Ranged (indexed) for loops: **for i in range(len(structure))**
- Functional Decomposition, documentation and doctests
- For each loops (non-indexed): **for item in structure:**

Card Games

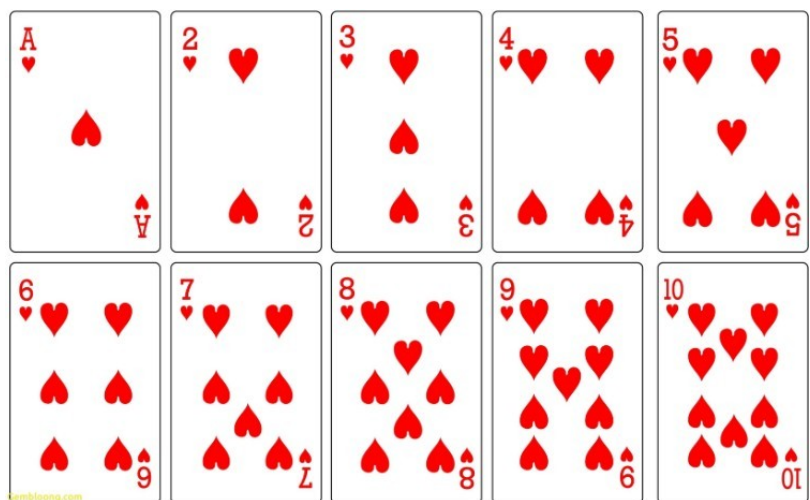
Card games provide excellent practice for dealing with data structures. Let's go through some exercises to build up the necessary objects involved in programming a card game. We will end with the rudiments for **Poker**

The Card:

For our purposes we will represent a playing card with a **tuple** of two values: **rank and suit**

Examining the cards to the right we can see that

- **Suit:** Hearts
- **Rank:** {1 – 10} counting the Ace card as 1
- **Face Card Ranks:**
 - o Jack = 11
 - o Queen = 12
 - o King = 13
- To make our cards easy to organize and test we will define them as a **tuple of two numbers**. Remember, tuples are immutable and cannot be changed. This works well for our game as we would not want to risk the chance of a card being mutated. Since a tuple is a single object it can be placed in a list, passed to a function, returned from a



function . . . etc. This organization is just the underlying methodology for the program to store and process the cards. As you'll see we will be outputting the cards with more descriptive information.

- **Example:**

- o Card = (rank, suit)
- o If the ranks are: {1 – 13}
- o And the suits are {0 – 3} where **Hearts = 0, Diamonds = 1, Clubs = 2, Spades = 3**
- o The cards shown above could be modeled as

- (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0), (8, 0), (9, 0), (10, 0)

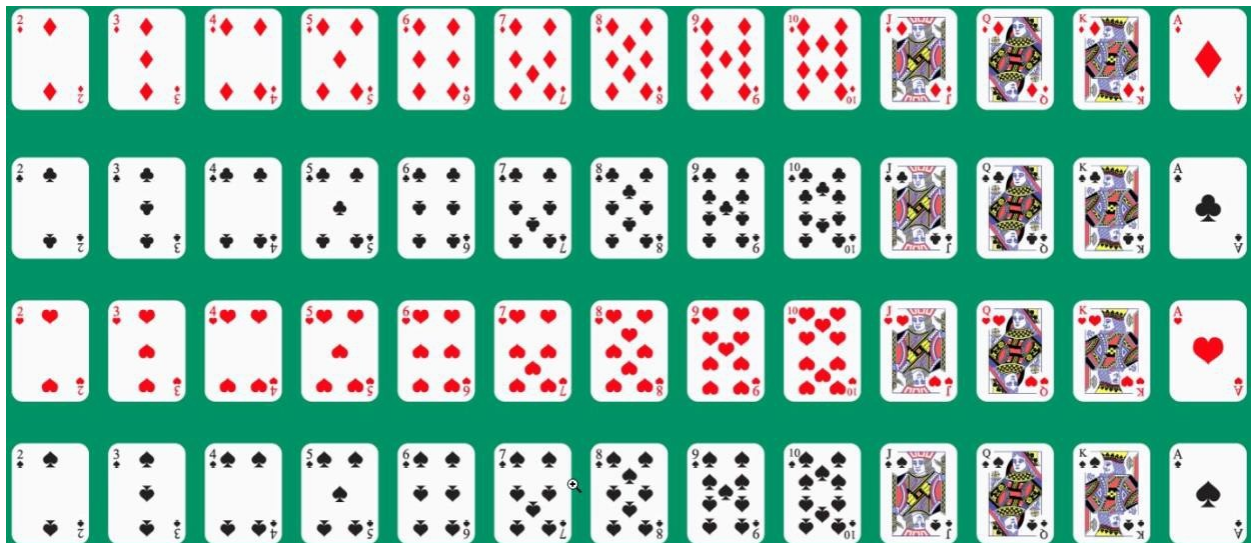
- An additional convenience would be to have a mapping of the suits to the number we are using to track the suit. This would allow us to output the cards in a more traditional fashion.

- o **Hearts = 0, Diamonds = 1, Clubs = 2, Spades = 3**

- A convenient way to model this relationship would be a Python dictionary. Let's use the number as the **key** and the text description as the **value**

- o **suits = {0 : "Hearts", 1 : "Diamonds", 2 : "Clubs", 3 : "Spades"}**

The Deck:



A **Deck** is a collection of 52 cards where you have 13 ranks in 4 suits. A nice way of modeling a deck in our game would be a **list**

- The individual card is modeled as **a single (rank, suit) tuple**
- The deck is modeled as **a list of (rank, suit) tuples**
- To continue with the example above, the 10-card collection of Hearts could be stored in a list
 - o Deck = [(1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0), (8, 0), (9, 0), (10, 0)]
 - o Deck[0] => (1, 0)
 - o Deck[1] => (2, 0)

Task One: Define basic card operations. Create the Python script `poker_functions.py`

1) Given the two following sequences of data, **and only using loops**, create a deck of 52 cards.

- **Dictionary of Suits:** suits = {0 : "Hearts", 1 : "Diamonds", 2 : "Clubs", 3 : "Spades"}
- **Tuple of Ranks:** ranks = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)
- Remember, the card is a simple tuple of (rank, suit) where both are the numerical quantities shown above. We will use the dictionary later, to be able to show the user the actual text pattern. Much easier for us (and the computer) to deal with numbers behind the scenes.
- In `poker_functions.py` define those sequences at the module level.

Write the function `create_deck` that uses **for each loops** to combine the suit and rank sequence values into a card tuple, which is then added to a list. **Return the list**. The deck should be in the following format. Notice that this is a list of tuples.

Test this before moving on!!

```
[(1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0), (8, 0), (9, 0), (10, 0), (11, 0), (12, 0), (13, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1), (10, 1), (11, 1), (12, 1), (13, 1), (1, 2), (2, 2), (3, 2), (4, 2), (5, 2), (6, 2), (7, 2), (8, 2), (9, 2), (10, 2), (11, 2), (12, 2), (13, 2), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (6, 3), (7, 3), (8, 3), (9, 3), (10, 3), (11, 3), (12, 3), (13, 3)]
```

Here is a display of the deck organized by suit.

```
(1, 0) (2, 0) (3, 0) (4, 0) (5, 0) (6, 0) (7, 0) (8, 0) (9, 0) (10, 0) (11, 0) (12, 0)
(1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (6, 1) (7, 1) (8, 1) (9, 1) (10, 1) (11, 1) (12, 1)
(1, 2) (2, 2) (3, 2) (4, 2) (5, 2) (6, 2) (7, 2) (8, 2) (9, 2) (10, 2) (11, 2) (12, 2)
(1, 3) (2, 3) (3, 3) (4, 3) (5, 3) (6, 3) (7, 3) (8, 3) (9, 3) (10, 3) (11, 3) (12, 3)
```

2) Now define the function `print_card`. This function will **accept a single card** and **print it** in the following format: **"rank" of "suit"** such that

- `print_card((1, 0))` => "Ace of Hearts"
- `print_card((2, 0))` => "2 of Hearts"
- `print_card((3, 0))` => "3 of Hearts"
- `print_card((11, 0))` => "Jack of Hearts"
- `print_card((12, 0))` => "Queen of Hearts"
- `print_card((13, 0))` => "King of Hearts"
- ... etc ...

Hints: Use the dictionary pattern to make the association of the **integer suit** with the **text representation of the suit**

- Given the dictionary above: suits = {0 : "Hearts", 1 : "Diamonds", 2 : "Clubs", 3 : "Spades"}

We can pull out the appropriate text by plugging in the Card suit number.

- **Given:**
 - o `card = (1, 0)` index the tuple to get the suit
 - o `suit = card[1]` now take the suit and plug it into the dictionary to get the appropriate text
 - o `suit_text = suits[suit]`

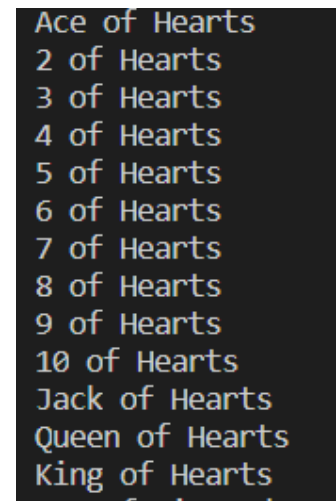
If you use another dictionary to associate the rank with the Ace, Jack, Queen, King you can apply the same pattern to the face cards. You should test your code by printing the deck to the console by calling **print_card** on each item. It should resemble the image to the right.

for card in deck:

print_card(card)

3) Define the function **shuffle** that *accepts the deck list* and scrambles (mutates) the list using the following algorithm.

- Create two random numbers between $\{0 \dots \text{len}(\text{deck}) - 1\}$
 - o You can use the Python random module for this
 - o `from random import randint`
 - o `i = randint(range_begin, range_end)`
- Swap the list items at those index locations. It may help to have a **swap** function
- Repeat this process a bunch of times . . . 1000 or so.
- Print the list to ensure sufficient shuffling
- It should go without saying that I want you to define this logic yourself, so you may not use Python's **random.shuffle** method.
- Realize that this function does not need to return anything because it is modifying the original list.
- You should test this function by printing the original deck, calling shuffle on the deck and then printing the deck again.



4) Define the function **deal** that deals a card from the deck by returning and removing the **top card**. The length of the deck must be reduced by 1 each time this function is called. The signature of this function is up to you. If your deck is module level, then this function does not need to accept an argument. If your deck is local to a block or module other than this one, then this function will need to accept the deck as an argument. Make intelligent choices and be ready to defend them. **Include documentation**

These functions should be thoroughly tested before you move on.

Task Two: Define the Poker Hand

Our simplified Poker Hand will consist of a **collection of 5 cards**. In order to streamline the ranking of poker hands we are going to define the ability to easily classify a poker hand by Suit and by Rank. Here are the poker hands we will be ranking in the order of dominance. Notice how some hands are dependent upon the card rank, while others are dependent upon the suit. And some are dependent upon both.

HAND RANKS

1. **Straight Flush:** All cards are the **same suit** and **in order**
2. **Four of a Kind:** Four Cards of the **same rank**
3. **Full House:** Three Cards of one **rank** and two of another **rank**
4. **Flush:** Five cards of the **same suit** but not in order
5. **Straight:** Five Cards in sequence but not the same **suit**
6. **Three of a Kind:** Three Cards of the **same rank**
7. **Two Pair:** A pair is two cards, **same rank**. Two of these
8. **One Pair:** Two cards of the **same rank**
9. **No combination:** Ranked by highest card

Hand as a Dictionary

The easiest way to rank a hand of five cards by suit is to define a hand as a dictionary with **the suit number as a key** and a list of cards in that suit as values. **Organizing a hand by suit allows for easy ranking by suit.** For example, how would we know that we had some type of flush (all cards of the same suit)? If any suit key points to a list of 5 cards it is a flush and there is no need to even look at the individual cards except to determine if it is a straight or just a regular flush.

Placing the cards into the correct suit list is easy because we can index the card tuple to determine its suit. Here is a sample hand showing a flush. Testing for a suit length of 5 will tell us we have a flush.

Hand: Notice that the hand dictionary has four keys . . . the suits

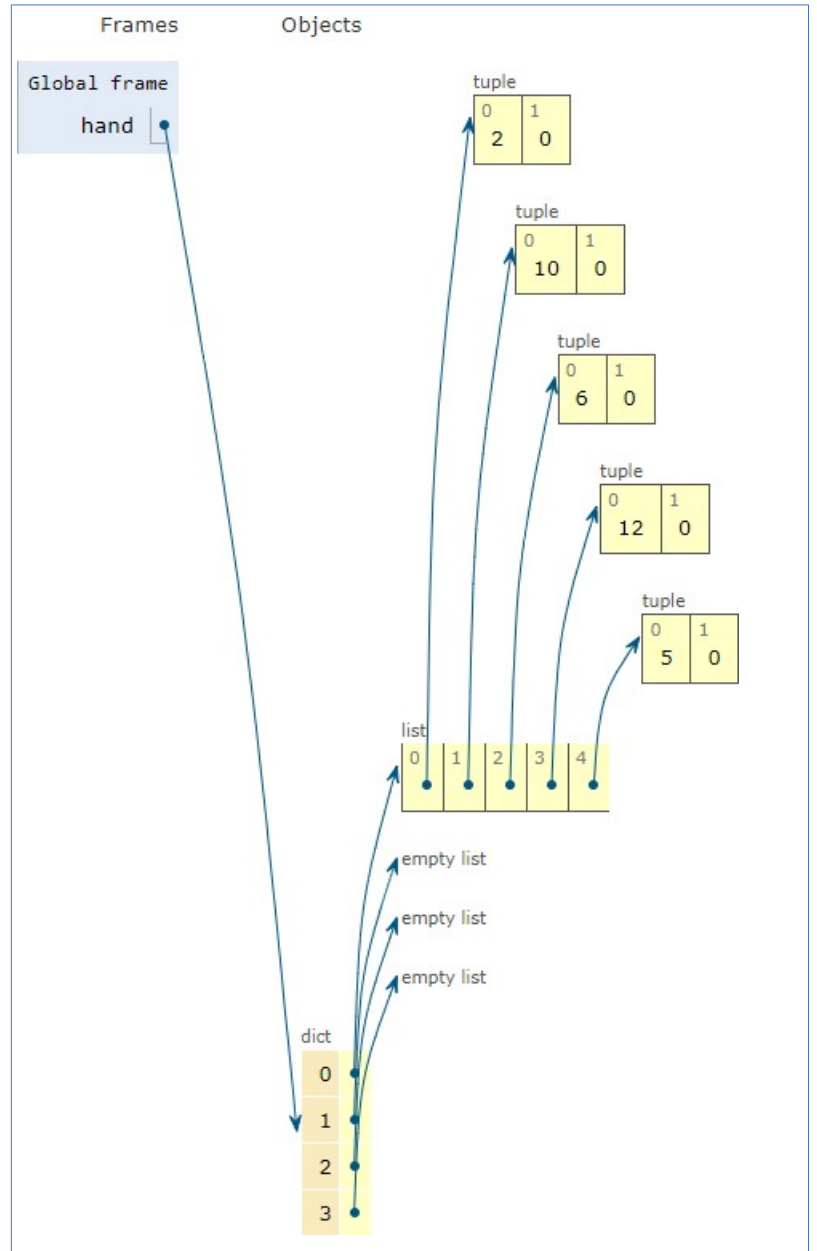
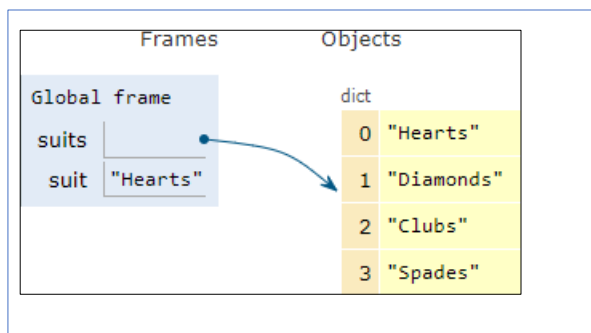
Key: Suit Number	Value: List of cards in this suit.	Length
0	[(2, 0), (10, 0), (6, 0), (12, 0), (5, 0)]	5
1	[]	0
2	[]	0
3	[]	0

Here is the memory mapping of this hand. Pay close attention to the three different sequence types that are being used to model a hand

- **Hand** => Dictionary:
 - o keys are integers (suit)
 - o values are lists (cards)
- **Hand Cards** => Lists
- **Individual Card** => Tuple

How can we determine which suit the flush is in? Take the key, plug it into our **suit dictionary** and extract the suit. No decisions required!!

suit = suits[0]



- Define the function **deal_hands** that **accepts a number representing the number of hands** to create, creates an empty hand dictionary {0: [], 1: [] . . . etc} for each **hand** and fills them with five cards by alternating dealing a single card to each hand. You must use a loop for this. Do not write five individual statements to deal the five cards. **Return the hands in a list.**

The hands should be put in a list and returned

- hands = deal_hand(3)** would deal 3 hands of 5 cards and return the hands in a list
- Define the function **print_hand** that accepts a **single hand** and prints it to the console by defining a loop and calling **print_card** for each card.
- Prove that your function works properly by dealing a hand and printing it to the terminal. The hand should be printed **similar** to this format. =====>

```
Ace of Hearts
2 of Hearts
3 of Hearts
4 of Hearts
5 of Hearts
6 of Hearts
7 of Hearts
8 of Hearts
9 of Hearts
10 of Hearts
Jack of Hearts
Queen of Hearts
King of Hearts
```

Be creative and make it look nice. A neat and intuitive display is crucial

Ranking Hands by Suit

At this point, you have defined a poker hand as a dictionary of suits. Now you can easily determine if you have a flush of some sort. If any suit in your hand has a length of five you know it is a flush. You then need to determine if it is a straight flush or a normal flush.

You will develop your hand ranking over the course of several function definitions; a single function for each type of hand. The functions should return a number that determines its rank. We will use the following table to rank hands

HAND	RANK VALUE
STRAIGHT FLUSH	8
FOUR OF A KIND	7
FULL HOUSE	6
FLUSH	5
STRAIGHT	4
THREE OF A KIND	3
TWO PAIR	2
ONE PAIR	1
NO COMBINATION	0

Task Three: Define the following 2 suit ranking functions

1. **straight_flush:** This function will *accept a hand*, check to see if a single suit contains five cards and determine if all cards are in sequential order. The cards in the hand do not have to be in sequential order for the hand to be a straight flush. It would be helpful to sort. ***This function will return 8 if the hand is a straight flush or 0 if it is not.***
2. **flush:** This function will *accept a hand*, check to see if a single suit contains 5 cards. ***This function will return 5 if the hand is a flush or 0 if it is not.***
3. Is there a logical way to determine the order of these function calls?

Ranking Hands by Card Rank

Now that you have **suit ranking** taken care of, we need to deal with the hands that are not based on suit. In our “rank” ranking we will take a similar approach to suit ranking. Instead of lists of suits we will build a list of ranks. There are 13 ranks. In our simplistic ranking Aces will always be one. Feel free to modify this and make Aces a higher rank.

Given the following **four of a kind: hand = (10, 0) (7, 1) (10, 1), (10, 3) (10, 2)** we would want to generate a structure that shows we have 4 10s so we don't have to explicitly check.

The rank list as a histogram: Since we need to count the number of 13 different ranks, we will need a list with 13 slots. Knowing that lists are indexed from 0 this will give us a list with indices 0 . . . 12

```
rank_list = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

a more Pythonic way of creating this list would be: `rank_list = [0] * 13` # Python is neat


Now we have a list with 13 slots, numbered 0 – 12, one for each rank. We can use these slots to create a histogram of card ranks. We will use a similar process to ranking hands by suit. You should be familiar with the concept of a histogram at this point.

Algorithm with example:


- **four of a kind: hand = (10, 0) (7, 1) (10, 1), (10, 3) (10, 2)**
- **card = (10, 0)** # this card represents a 10 of Hearts
- Indexing the card tuple for the rank (**card[0]**) gives us the rank **10**. This card will be added to the “rank list” at index 9. Add one to the count of cards in this slot
 - o `ranks[card[0] - 1] += 1`
- **card = (7, 1)** # this card represents a 7 of Diamonds
 - o `ranks[card[0] - 1] += 1`
- **card = (10, 1)** # this card represents a 10 of Diamonds
 - o `ranks[card[0] - 1] += 1`
- **card = (10, 3)** # this card represents a 10 of Spades
 - o `ranks[card[0] - 1] += 1`
- **card = (10, 2)** # this card represents a 10 of Clubs
 - o `ranks[card[0] - 1] += 1`

Notice how the statement to increment each count is the same for every card. This lack of decision statements is the convenience provided to us by intelligent usage of data structures. The resulting rank list is shown in the following table. Keep in mind that you need to subtract one from the card rank because list indices begin at zero.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Count	0	0	0	0	0	0	1	0	0	4	0	0	0



1 card of
rank 7



4 cards of
rank 10

Once populated, this list lets us easily determine if we have multiple cards of the same rank.

Task Four: Define the following rank ranking functions

1. **hand_ranks:** This function will *accept a hand* and create and return the rank list described above
2. **four_of_a_kind:** This function will *accept the rank list* created above and *return 7* if there is a slot with a count of four, 0 if there is not
3. **full_house:** This function will *accept the rank list* and *return 6* if there is a slot with 3 cards and a slot with 2 cards; 0 if not
4. **straight:** This function will *accept the rank list* and *return 4* if there are 5 slots *in a sequence* with a count of 1; 0 if not.
5. **three_of_a_kind:** This function will *accept the rank list* and *return 3* if there is a slot with a count of 3; 0 if not
6. **two_pair:** This function will *accept the rank list* and *return 2* if there are two slots with a count of 2; 0 if not
7. **one_pair:** This function will *accept the rank list* and *return 1* if there is a slot with a count of 2; 0 if not

No combination and tie resolution

If two hands are tied or neither hand has a combination listed above, then the highest card wins. This can easily be determined by *iterating backwards* through the rank list and finding the first slot with a count greater than zero. This is the highest card.

Task Five: Define the following functions

1. **highest_card:** This function will *accept the rank list and return the value of the highest card*
2. **compare_hands:** This function will *accept two hands and determine which hand wins*. I will not describe an algorithm in detail here, but you will need to call each of those functions on each hand and determine who has the highest score. Display the results. Print the hands along with a description of the winning hand. This function does not return anything. Here are a few samples.

```
POKER HAND ONE
=====
| Card 1:=> 8 of Hearts | Card 2:=> 4 of Diamonds | Card 3:=> 3 of Clubs | Card 4:=> 10 of Clubs | Card 5:=> 5 of Spades

POKER HAND TWO
=====
| Card 1:=> 3 of Hearts | Card 2:=> 5 of Hearts | Card 3:=> 8 of Diamonds | Card 4:=> Ace of Spades | Card 5:=> 8 of Spades
HAND TWO WINS WITH: One Pair
```

```
POKER HAND ONE
=====
| Card 1:=> Jack of Diamonds | Card 2:=> 9 of Clubs | Card 3:=> 10 of Clubs | Card 4:=> 8 of Spades | Card 5:=> Queen of Spades

POKER HAND TWO
=====
| Card 1:=> Jack of Hearts | Card 2:=> 2 of Hearts | Card 3:=> 6 of Diamonds | Card 4:=> 3 of Clubs | Card 5:=> 3 of Spades
HAND ONE WINS WITH: Straight
```

Testing: I have included a Python script that generates hands of each rank. Use these to test the logic of your own functions. **That is all that script is for. You are not required to use it. It is a convenient tool for testing your functions. All it does is generate poker hands of various ranks.**

It is strongly recommended that you do this because it will ensure that your design fits with the requirements (which will be strictly graded).

Task Six: Putting It All Together

You now have all the rudiments to build a simple Poker game. What remains is to incorporate the functions defined above into an actual working poker game. The game can be as fancy as you like, but the following are **MINIMUM REQUIREMENTS**. In order to get full credit you **MUST** meet the following.

Minimum Requirements

1. All Poker function definitions are in a module called **poker_functions.py**. Docstrings are required for all functions. Include comments in your code to provide context to what the code is doing.
2. Doctests are required where appropriate. This is an incredibly important concept.
3. All “game logic” is in a module called **poker_game.py**
4. The game will consist of two players: **Human vs Computer**
5. The game should begin with a greeting and a display of simplified rules. This description is up to you.
6. Deal two hands, only showing the human’s hand. The computer’s hand will be stored and analyzed behind the scenes
7. Once the two hands have been dealt and the human has been shown their hand, show a menu that allows the user to modify the hand they were dealt.
8. Allow the human to specify up to four cards to replace. **Validate this input**. The easiest method to handle this would be to allow the player to specify a card by it’s number in

the hand . . . as shown in the screen shots above.

9. If the human specifies any cards to replace, remove those cards from the hand and replace them with new cards from the deck. The replacement should be handled by calling the deal function.
10. Define a simple AI to instruct the computer on choosing cards to be replaced. This should be **somewhat** intelligent but does not have to be perfect. For example, if the computer has a straight do not replace any cards. If the computer has a single pair maybe discard the three cards that aren't part of the pair . . . etc. You could accomplish this by running the hand through the ranking functions to assist with the decision making.
11. Clearly display the winner and the type of hand they won with.
12. Allow the user to play as many games as they like, without having to restart the program.
13. However you decide to implement this game, the play must be intuitive and include messages and instructions for the user. **Everything must be crystal clear.**
14. You should include menu-like structures and easy to follow terminal output

Cool Additions:

1. Betting
2. More than one opponent

Would you like to play again? (Y/N): Y

PLAYER HAND – One Pair

=====

- 1.) Ace of Hearts
- 2.) 6 of Spades
- 3.) 7 of Spades
- 4.) Queen of Hearts
- 5.) Queen of Spades

How many cards would you like to discard? [0-4]: 3

(1/3) Please select the card to discard [1-5]: 1

(2/3) Please select the card to discard [1-5]: 2

(3/3) Please select the card to discard [1-5]: 3

COMPUTER HAND – One Pair

=====

- 1.) Ace of Clubs
- 2.) 3 of Diamonds
- 3.) 3 of Spades
- 4.) 6 of Clubs
- 5.) 8 of Spades

PLAYER HAND – Two Pair

=====

- 1.) Ace of Diamonds
- 2.) Ace of Spades
- 3.) 4 of Spades
- 4.) Queen of Hearts
- 5.) Queen of Spades

Winner: PLAYER, with: Two Pair

Would you like to play again? (Y/N):

PLAYER HAND - One Pair

=====

- 1.) 3 of Diamonds
- 2.) 4 of Spades
- 3.) Jack of Diamonds
- 4.) King of Diamonds
- 5.) King of Clubs

How many cards would you like to discard? [0-4]: 0

COMPUTER HAND - One Pair

=====

- 1.) 2 of Hearts
- 2.) 4 of Clubs
- 3.) 6 of Clubs
- 4.) 9 of Diamonds
- 5.) 9 of Clubs

PLAYER HAND - One Pair

=====

- 1.) 3 of Diamonds
- 2.) 4 of Spades
- 3.) Jack of Diamonds
- 4.) King of Diamonds
- 5.) King of Clubs

Winner: PLAYER, with: King

Would you like to play again? (Y/N): n

Thanks for playing Poker

RUBRIC: Partial credit will be awarded

Requirement	Points
All required Poker functions in poker_functions.py	10
All game logic in poker_game.py	10
All required functions are defined according to instructions. Parameters and return values are correct. Restrictions are followed carefully	15
Code is meaningfully commented	5
Nicely formatted and descriptive Docstrings and Doctests are included	5
Card is a tuple of (rank, suit)	5
Poker hand is a dictionary {suit : [cards]}	5
CardDeck is a list of Card tuples	5
Game play is intuitive and descriptive	10
Cards are discarded correctly with validated input	5
Rudimentary functional AI written to guide the computer in card discarding	10
Game correctly identifies winner and displays winning hand type	10
User is allowed to play multiple games without restarting the program	5

Submission: You will submit this lab in Brightspace. Create a ZIP archive of all relevant files. Include any instructions that I may need to play your game. Attach the archive to the submission area in Blackboard.

NOTE: You must follow the instructions carefully. You are not allowed to search the Internet for a random poker game and use that as your solution or even as a basis for your solution. Your submission **must** adhere to the structures described in this document. It will be strictly graded.

If you have an idea for a modification, you must clear it with me first.