

|  |  |
| --- | --- |
| Assessment Module | RGU CM 1601 Programming Fundamentals |
| Module Leader | Rajitha Jayasinghe |
| Student Name | Lasal Chathranjey Jayawardena |
| RGU Id | 2016279 |
| IIT Id | 20200344 |
| Submission Type | Coursework Part 1 |
| Submission Date | 10th December 2020 |
| IDE | Visual Studio Code |

Course Work Report

Summary

The program written for the coursework is an interactive command line OMI game(modified). In the program, the computer plays the role of the second player and also the game administrator as well. There is a separate folder for the game source code, 1 pdf document on the flowchart of the shuffling algorithm and finally another pdf document on the flowchart of the game phase (8 tricks after the game deck is shuffled).

Improvements

1. Additional functionality where each trick player is tracked even after the game is closed and then opened (using text files).
2. Added ASCII art for Welcome message, Result message, thank you message.
3. Changed how the user decks are printed into a more user friendly readable format.
4. Implemented the test cases using try, except and assertion with test cases for function which checks the trick winner.
5. Incorporation of meaningful messages to user when the game rules are not followed.
6. Completely implemented functional decomposition and modularization.
7. Further Strengthened the logic when computer chooses cards (e.g. If there are higher cards than the users’ card then selects the lower out of all).
8. All functions and algorithms written with balanced consideration of Big-Oh, code-readability and memory used.
9. Doc strings used to describe each game module made and meaningful comments used for assistance in understanding code.
10. Added an extra sanitation function for user input for flexibility and improved user experience.

app.py code – This the main part of the program where the entire game happens.

"""

This is a two player OMI Game between computer and player

"""

from deck import intialize\_deck, deal\_cards

from game\_logic import computer\_lead, player\_lead

from display\_func import display\_welcome\_msg, display\_hand, display\_player\_won, display\_computer\_won, display\_player, display\_draw, display\_thank\_you\_message

from validate\_func import validate\_trump\_suit

from computer import choose\_trump

import sys

import os

# Intilaizing global variables

trump\_announce = ""

game\_trick\_player = ""

# End of Global Variables

*def* main():

    # Displays Welcome message

    display\_welcome\_msg()

    # This is to identify who will tell the trumps

    global trump\_announce

    # This is to identify who will lead the trick, this changes every trick

    global game\_trick\_player

    trump\_announce, game\_trick\_player = get\_game\_details()

    while True:

        # intialize scores and the game deck

        Computer\_Score = 0

        Human\_Score = 0

        game\_deck = intialize\_deck()

        # This when all 8 tricks begin, and the most of the game logic is implemented

        game\_tricks(game\_deck, Computer\_Score, Human\_Score,game\_trick\_player, trump\_announce)

        # Switch trump annouonce and game\_trick players after 8 tricks, when player wants to play agian

        if trump\_announce == "player":

            trump\_announce = "computer"

            game\_trick\_player = "player"

        else:

            trump\_announce = "player"

            game\_trick\_player = "computer"

        # Clear deck after each round( after 8 tricks )

        game\_deck.clear()

        # Ask user if he wants play agian

        user\_input = input("Do you want to play anothe round? (y/n)").strip().lower()[0]

        print("\n")

        if user\_input == 'n':

            break

*def* game\_tricks(*game\_deck*, *Computer\_Score*, *Human\_Score*, *game\_trick\_player*, *trump\_announce*):

    NUMBER\_OF\_TRICKS = 8

    # This when all the 8 tricks will happen

    for i in range(1, NUMBER\_OF\_TRICKS+1):

        Trick\_count = i

        if game\_trick\_player == "computer":

            if Trick\_count == 1 and trump\_announce == "player":

                # deal 4 decks for player

                player\_hand = deal\_cards(game\_deck, 4)

                display\_hand(player\_hand)

                Trump\_Card = validate\_trump\_suit()

                # deal the next 4 cards

                player\_hand += deal\_cards(game\_deck, 4)

                # deal 8 cards for the computer

                computer\_hand = deal\_cards(game\_deck, 8)

            winner = computer\_lead(game\_deck, Trick\_count, player\_hand, computer\_hand, Trump\_Card)

        else:

            if Trick\_count == 1 and trump\_announce == "computer":

                computer\_hand = deal\_cards(game\_deck, 4)

                Trump\_Card = choose\_trump(computer\_hand)

                print(f"Computer chose trump as {Trump\_Card}\n")

                computer\_hand += deal\_cards(game\_deck, 4)

                player\_hand = deal\_cards(game\_deck, 8)

            display\_player(Trick\_count, Trump\_Card, player\_hand)

            print("You lead the trick!")

            winner = player\_lead(game\_deck, Trick\_count, player\_hand, computer\_hand, Trump\_Card)

        if winner == "You won":

            game\_trick\_player = "player"

            Human\_Score += 2

            print("Player +2")

        else:

            game\_trick\_player = "computer"

            Computer\_Score += 2

            print("Computer +2")

        print("Computer score is {}".format(Computer\_Score))

        print("Your score is {}\n\n".format(Human\_Score))

    # Check the winner after all tricks

    game\_result(Computer\_Score, Human\_Score)

*def* game\_result(*Computer\_Score*, *Human\_Score*):

    # display appropriate output after 8 tricks

    print("Computer score is {} / Player score is {}".format(Computer\_Score, Human\_Score))

    if Computer\_Score > Human\_Score:

        display\_computer\_won()

    elif Computer\_Score < Human\_Score:

        display\_player\_won()

    else:

        display\_draw()

*def* get\_game\_details():

    # get trump\_announce and game\_trick+player from file

    try:

        with open("./trump.txt", "r") as file:

            data = file.read()

            return data.split()

    except:

        # This when the file in initailly not there.

        return ["player","computer"]

*def* set\_game\_details():

    try:

        with open("./trump.txt","w") as file:

            file.write(trump\_announce+" "+game\_trick\_player)

    except:

        pass

# Run the main function

if \_\_name\_\_ == "\_\_main\_\_":

    try:

        main()

    except *KeyboardInterrupt*:

        set\_game\_details()

        os.system('cls' if os.name == 'nt' else 'clear')

        display\_thank\_you\_message()

        sys.exit(0)

    else:

        set\_game\_details()

        os.system('cls' if os.name == 'nt' else 'clear')

        display\_thank\_you\_message()

computer.py code - This is where the functions done by the computer as a player are stored.

"""

This is where most of the function needed by the computer is written

"""

from random import choice

*def* play\_card(*card\_deck*):

    # This is when the computer's turn to lead the trick

    return choice(card\_deck)

*def* computer\_play\_card(*c\_deck*, *player\_card*, *trump*):

    suit\_ace\_dict = {'J': "11", 'Q': "12", 'K': "13", 'A': "14"}

    # check whether similar cards

    similar = [c for c in c\_deck if c[1] == player\_card[1]]

    # all cards with trump

    trump = [c for c in c\_deck if c[1] == trump]

    # all other card other than trumps, usefule when no similar cards

    other = [c for c in c\_deck if (c[1] != player\_card[1]) and (c[1] != trump)]

    # check if there are similar cards

    if len(similar) > 0:

        higher = []

        # logic to get all similar cards higher than the player card

        # nested if conditions needed to give value to ace and picture cards and then compare

        for card in similar:

            if (suit\_ace\_dict.get(card[0])):

                if suit\_ace\_dict.get(player\_card[0]):

                    if *int*(suit\_ace\_dict[card[0]]) > *int*(suit\_ace\_dict[player\_card[0]]):

                        higher.append(card)

                else:

                    if *int*(suit\_ace\_dict[card[0]]) > *int*(player\_card[0]):

                        higher.append(card)

            else:

                if suit\_ace\_dict.get(player\_card[0]):

                    if *int*(card[0]) > *int*(suit\_ace\_dict[player\_card[0]]):

                        higher.append(card)

                else:

                    if *int*(card[0]) > *int*(player\_card[0]):

                        higher.append(card)

        if len(higher) > 0:

            # if there are cards higher than the card the player played

            # return lowest of them all, which will still win

            highest\_lower = get\_lowest\_card(higher)

            return highest\_lower

        else:

            # if there no cards higher than the card the player playes

            # then give the lowest card of that suit

            lowest\_card = get\_lowest\_card(similar)

            return lowest\_card

    # check for trump options

    # this is when the computer does not have cards for the suit the player played

    elif len(trump) > 0 and player\_card[1] != trump:

        # return the lowest trump card availbale in computers hand

        lowest\_trump = get\_lowest\_card(trump)

        return lowest\_trump

    # lowest in other cards

    else:

        # this when the computer does not have card swith the player suit or trumps

        # so get the cards with lowest value in all suits

        lowest\_other = get\_lowest\_card(other)

        lowest\_available = [c for c in other if c[0] == lowest\_other[0]]

        # randomly return card from the set of lowest ard values if there multiple cards

        return choice(lowest\_available)

*def* get\_lowest\_card(*deck*):

    # returns lowest card from a given deck

    suit\_ace\_dict = {'J': "11", 'Q': "12", 'K': "13", 'A': "14"}

    lowest = deck[0]

    # nested ifs needed because the picture cards and aces dont have a value

    for card in deck:

        if (suit\_ace\_dict.get(card[0])):

            if(suit\_ace\_dict.get(lowest[0])):

                if *int*(suit\_ace\_dict[card[0]]) < *int*(suit\_ace\_dict[lowest[0]]):

                    lowest = card

            else:

                if *int*(suit\_ace\_dict[card[0]]) < *int*(lowest[0]):

                    lowest = card

        else:

            if(suit\_ace\_dict.get(lowest[0])):

                if *int*(card[0]) < *int*(suit\_ace\_dict[lowest[0]]):

                    lowest = card

            else:

                if *int*(card[0]) < *int*(lowest[0]):

                    lowest = card

    return lowest

*def* choose\_trump(*hand*):

    suit\_occurence = {"♠": 0, "♣": 0, "♥": 0, "♦": 0}

    # This will create a dict with number of cards for each suit

    for c in hand:

        if(suit\_occurence.get(c[1]) != None):

            suit\_occurence[c[1]] += 1

    # Find the maximum value of the occurence in each suit

    suit\_most\_occur = max(suit\_occurence, *key*=suit\_occurence.get)

    # get the number of card for the suit which had the highest occurence

    # helpful dealing when more than one suit has the most occurence

    max\_val = suit\_occurence[suit\_most\_occur]

    # Calculate how mny suits with the max occurence

    num\_max = 0

    for i in suit\_occurence.items():

        if i[1] == max\_val:

            num\_max += 1

    # check if a single suit occurs three times then return

    if max\_val >= 3:

        return suit\_most\_occur

    # check if two suits have the max occurence meaning two cards for both suits

    elif max\_val == 2 and num\_max == 2:

        # get both suit with the most occurence into an array

        suits = [i[0] for i in suit\_occurence.items() if i[1] == max\_val]

        # check if the cards of first suit have an Ace

        ace\_in\_suit\_1 = any(i[0] == 'A' for i in hand if i[1] == suits[0])

        # check if the cards of second suit have an Ace

        ace\_in\_suit\_2 = any(i[0] == 'A' for i in hand if i[1] == suits[1])

        if ace\_in\_suit\_1:

            # return second suit if first has ace... Followed the cw tactics

            return suits[1]

        if ace\_in\_suit\_2:

            # return first suit if second suit has ace... Followed the cw tactics

            return suits[0]

        # This will calculate the total value for the cards in each suit

        # Add all card values and even the picture cards and ace is given a value

        suit1\_score = calculate\_suit\_sum(hand, suits[0])

        suit2\_score = calculate\_suit\_sum(hand, suits[1])

        if suit1\_score > suit2\_score:

            # If the the first suit has cards with higher value then return it

            return suits[0]

        elif suit1\_score < suit2\_score:

            # If the the second suit has cards with higher value then return it

            return suits[1]

        else:

            # If both suit have cards with equal value then use

            # get the cards for each suit separately

            suit1 = [i for i in hand if i[1] == suits[0]]

            suit2 = [i for i in hand if i[1] == suits[1]]

            # Then I calculated the range of the cards

            # This is an improvement: Reason is

            # eg: It is better to have Q and 7 instead of J and 8

            # Eben thought they have the same value the first set is better

            suit1\_range = calculate\_suit\_range(suit1)

            suit2\_range = calculate\_suit\_range(suit2)

            if suit1\_range > suit2\_range:

                # So if the values of the cards are far apart for suit1 then return the suit as trumps

                return suits[0]

            elif suit1\_range < suit2\_range:

                # So if the values of the cards are far apart for suit2 then return the suit as trumps

                return suits[1]

            # Last case the the cards are same for both suits:

            # So used a random choice from random module to select one suit between the two

            return choice(suits)

    elif max\_val == 2 and num\_max == 1:

        # This when one suit has two cards but the other suits to has one card each

        # Accordind to cw tactics they have told that if the two cards are lower then,

        # The players choose the suit with no cards, hoping to get those cards in the second deal

        # My assumption was if both the two cards were between 7 and 10 incluive then to choose hte other suit

        #  chose the suit with most cards

        max\_suit = [i[0] for i in suit\_occurence.items() if i[1] == max\_val][0]

        #  select all cards which have the max\_suit

        max\_suit\_deck = [i for i in hand if i[1] == max\_suit]

        # got the numeric total of the cards, including picture cards and ace

        suit\_total = calculate\_suit\_sum(hand, max\_suit)

        # got the averge of the cards

        avg\_suit = round(suit\_total / 2)

        # got the range of the two cards

        range\_suit = calculate\_suit\_range(max\_suit\_deck)

        if avg\_suit < 10 and range\_suit <= 3:

            min\_suit\_deck = [i[0] for i in suit\_occurence.items() if i[1] == 0]

            return min\_suit\_deck[0]

        return max\_suit

    else:

        lowest = get\_lowest\_card(hand)[0]

        lowest\_card = [c for c in hand if c[0] == lowest]

        return choice(lowest\_card)[1]

*def* calculate\_suit\_sum(*cards*, *suit*):

    # Helps to implement in finding the sum of all cards

    suit\_ace\_dict = {'J': "11", 'Q': "12", 'K': "13", 'A': "14"}

    total = 0

    for c in cards:

        if c[1] == suit:

            if (suit\_ace\_dict.get(c[0])):

                total += *int*(suit\_ace\_dict[c[0]])

            else:

                total += *int*(c[0])

    return total

*def* calculate\_suit\_range(*deck*):

    # calculate the range of given cards include the picture cards and the Aces

    # needed when choosing trumps

    # nested if conditions needed to map a value to picture cards and aces

    suit\_ace\_dict = {'J': "11", 'Q': "12", 'K': "13", 'A': "14"}

    card1 = deck[0]

    card2 = deck[1]

    if suit\_ace\_dict.get(card1[0]):

        if suit\_ace\_dict.get(deck[1][0]):

            return abs(*int*(suit\_ace\_dict[card1[0]]) - *int*(suit\_ace\_dict[card2[0]]))

        else:

            return abs(*int*(suit\_ace\_dict[card1[0]]) - *int*(card2[0]))

    else:

        if suit\_ace\_dict.get(card2[0]):

            return abs(*int*(card1[0]) - *int*(suit\_ace\_dict[card2[0]]))

        else:

            return abs(*int*(deck[0][0]) - *int*(deck[1][0]))

deck.py – Holds all functions mainly connected to the deck of game and the check winner function.

"""

This is where the common function involving the deck are stored.

"""

import itertools

import random

*def* shuffle\_deck(*deck*):

    # Implemented Fisher–Yates algorithm to shuffle the cards

    # Because it has a Time complexity of O(n) and the after the shuffle the randomness is also good

    # counter set at at length of deck -1

    start\_counter = len(deck) - 1

    for index in range(start\_counter, 0, -1):

        # generate random index

        rand\_index = random.randint(0, index)

        # switch card on index with card on the randomly generated index

        deck[index], deck[rand\_index] = deck[rand\_index], deck[index]

*def* intialize\_deck():

    suits = ("♠", "♣", "♥", "♦")

    numbers = ("A", "K", "Q", "J", "10", "9", "8", "7")

    # This is for reference only : symbol\_dict= {"clubs":'♣', "diamonds":"♦", "hearts":"♥","spades":"♠"}

    # Chose itertools for readability

    deck = *list*(itertools.product(numbers, suits))

    # deck = [(x,y) for x in suits for y in numbers]

    # Perform a inplace shuffle using the shuffle\_deck function defined above

    shuffle\_deck(deck)

    return deck

*def* deal\_cards(*deck*, *number\_to\_Deal*=0):

    # copy deck

    clone\_deck = deck.copy()

    # Simulate the card being removed from top

    for i in range(number\_to\_Deal):

        deck.pop(0)

    # Could have created a list and added the poped values but chose this as method looks cleaner

    return clone\_deck[:number\_to\_Deal]

*def* check\_trick\_winner(*player\_card*, *computer\_card*, *trump*, *trick\_leader*="player"):

    suit\_ace\_dict = {'J': "11", 'Q': "12", 'K': "13", 'A': "14"}

    # case when the two suits of the cards are same

    if player\_card[1] == computer\_card[1]:

        # Nested if conditions needed as the cards need to be mapped by the dictionary

        # Reason is to give the picture card and aces a value

        # Enter when the first card of the pair is a picture card or ace

        if (suit\_ace\_dict.get(player\_card[0])):

            # Enter when the first card and the second cards of the pair are a picture card or ace

            if(suit\_ace\_dict.get(computer\_card[0])):

                if *int*(suit\_ace\_dict[player\_card[0]]) > *int*(suit\_ace\_dict[computer\_card[0]]):

                    return "You won"

                return "Computer won"

            # Enter when the first card of the pair is a picture card or ace and the second is a number card

            else:

                if *int*(suit\_ace\_dict[player\_card[0]]) > *int*(computer\_card[0]):

                    return "You won"

                return "Computer won"

        # Enter when the first card of the pair is a number card

        else:

            # Enter when the first card of the pair is a number card and the second is a colored card or ace

            if(suit\_ace\_dict.get(computer\_card[0])):

                if *int*(player\_card[0]) > *int*(suit\_ace\_dict[computer\_card[0]]):

                    return "You won"

                return "Computer won"

            else:

                # Enter when the both cards of the pair are number cards

                if *int*(player\_card[0]) > *int*(computer\_card[0]):

                    return "You won"

                return "Computer won"

    # this is when the player enter a trump and computer has another suit

    elif player\_card[1] == trump and computer\_card[1] != trump:

        return "You won"

    # this is when the computer enters a trump and player has another suit

    elif player\_card[1] != trump and computer\_card[1] == trump:

        return "Computer won"

    else:

    # When both player and computer have different cards and both are not trumps

        if trick\_leader == "player":

            return "You won"

        return "Computer won"

"""

try:

    case1 = check\_trick\_winner(("K", "♥"), ("9", "♥"), "♠", "computer")

    assert case1 == "You won", "Case 1 failed"

    case2 = check\_trick\_winner(("8", "♦"), ("9", "♦"), "♦", "player")

    assert case2 == "Computer won", "Case 2 failed"

    case3 = check\_trick\_winner(("10", "♣"), ("J", "♥"), "♣", "computer")

    assert case3 == "You won", "Case 3 failed"

    case4 = check\_trick\_winner(("Q", "♦"), ("9", "♥"), "♥", "computer")

    assert case4 == "Computer won", "Case 4 failed"

    case5 = check\_trick\_winner(("Q", "♠"), ("A", "♦"), "♥", "player")

    assert case5 == "You won", "Case 5 failed"

    case6 = check\_trick\_winner(("10", "♦"), ("10", "♠"), "♣", "computer")

    assert case6 == "Computer won", "Case 6 failed"

except AssertionError as e:

    print(e)

else:

    print("All test cases pass")

    # All test cases pass

"""

display\_func.py – Stores all function, that display content to user.

"""

This stores all the functions which display something to the user.

"""

from time import sleep

*def* display\_player(*Trick\_count*, *Trump\_Card*, *player\_hand*,*computer\_Card*="---", *player\_card*="---"):

    print("-------------------------------------\n")

    print(f"Trick {Trick\_count}")

    print(f"Trump suit : {Trump\_Card}")

    display\_card(computer\_Card, "Computer")

    display\_card(player\_card, "Player")

    display\_hand(player\_hand)

*def* display\_card(*card*, *player*):

    if card == "---":

        print(f"{player} played : {card}")

    else:

        # used of in the middle as it is more user friendly. "K of ♠" instead of "K♠"

        print(f"{player} played : {card[0]} of {card[1]}")

*def* display\_hand(*current\_hand*=[]):

    # This function displays all cards currently in player hand

    base\_msg = "You have"

    if len(current\_hand) == 0:

        base\_msg += " 0 Cards left. "

    else:

        for card in (current\_hand):

            base\_msg += f" {card[0]} of {card[1]},"

    # slicing done to remove extra comma at the end when having cards listed

    base\_msg = base\_msg[:len(base\_msg)-1]+"\n"

    print(base\_msg)

*def* display\_welcome\_msg():

    print(    """

    888       888          888                                              888

    888   o   888          888                                              888

    888  d8b  888          888                                              888

    888 d888b 888  .d88b.  888  .d8888b .d88b.  88888b.d88b.   .d88b.       888888 .d88b.

    888d88888b888 d8P  Y8b 888 d88P"   d88""88b 888 "888 "88b d8P  Y8b      888   d88""88b

    88888P Y88888 88888888 888 888     888  888 888  888  888 88888888      888   888  888

    8888P   Y8888 Y8b.     888 Y88b.   Y88..88P 888  888  888 Y8b.          Y88b. Y88..88P

    888P     Y888  "Y8888  888  "Y8888P "Y88P"  888  888  888  "Y8888        "Y888 "Y88P"

    """)

    sleep(1)

    print("""

                             .d88888b.  888b     d888 8888888

                            d88P" "Y88b 8888b   d8888   888

                            888     888 88888b.d88888   888

                            888     888 888Y88888P888   888

                            888     888 888 Y888P 888   888

                            888     888 888  Y8P  888   888

                            Y88b. .d88P 888   "   888   888

                             "Y88888P"  888       888 8888888

          """)

    sleep(0.5)

*def* display\_player\_won():

    print(

    """

      \_\_   \_\_                                \_

      \ \ / /                               | |

       \ V /\_\_\_  \_   \_  \_\_      \_\_\_\_\_  \_ \_\_ | |

        \ // \_ \| | | | \ \ /\ / / \_ \| '\_ \| |

        | | (\_) | |\_| |  \ V  V / (\_) | | | |\_|

        \\_/\\_\_\_/ \\_\_,\_|   \\_/\\_/ \\_\_\_/|\_| |\_(\_)

    """

    )

*def* display\_computer\_won():

    print(

    """

     \_\_\_\_\_                             \_                                  \_

    /  \_\_ \                           | |                                | |

    | /  \/ \_\_\_  \_ \_\_ \_\_\_  \_ \_\_  \_   \_| |\_ \_\_\_ \_ \_\_  \_\_      \_\_\_\_\_  \_ \_\_ | |

    | |    / \_ \| '\_ ` \_ \| '\_ \| | | | \_\_/ \_ \ '\_\_| \ \ /\ / / \_ \| '\_ \| |

    | \\_\_/\ (\_) | | | | | | |\_) | |\_| | ||  \_\_/ |     \ V  V / (\_) | | | |\_|

     \\_\_\_\_/\\_\_\_/|\_| |\_| |\_| .\_\_/ \\_\_,\_|\\_\_\\_\_\_|\_|      \\_/\\_/ \\_\_\_/|\_| |\_(\_)

                        | |

                        |\_|

    """

    )

*def* display\_draw():

    print(

    """

     \_\_\_\_\_ \_     \_              \_\_\_\_\_\_                    \_

    |\_   \_| |   (\_)             |  \_  \                  | |

      | | | |\_   \_ \_\_\_    \_\_ \_  | | | |\_ \_\_ \_\_ \_\_\_      \_| |

      | | | \_\_| | / \_\_|  / \_` | | | | | '\_\_/ \_` \ \ /\ / / |

     \_| |\_| |\_  | \\_\_ \ | (\_| | | |/ /| | | (\_| |\ V  V /|\_|

     \\_\_\_/ \\_\_| |\_|\_\_\_/  \\_\_,\_| |\_\_\_/ |\_|  \\_\_,\_| \\_/\\_/ (\_)

    """

    )

*def* display\_thank\_you\_message():

    print(

        """

  \_\_\_\_\_ \_                 \_                           \_\_              \_\_\_\_  \_             \_

 |\_   \_| |\_\_   \_\_ \_ \_ \_\_ | | \_\_  \_   \_  \_\_\_  \_   \_   / \_| \_\_\_  \_ \_\_  |  \_ \| | \_\_ \_ \_   \_(\_)\_ \_\_   \_\_ \_

   | | | '\_ \ / \_` | '\_ \| |/ / | | | |/ \_ \| | | | | |\_ / \_ \| '\_\_| | |\_) | |/ \_` | | | | | '\_ \ / \_` |

   | | | | | | (\_| | | | |   <  | |\_| | (\_) | |\_| | |  \_| (\_) | |    |  \_\_/| | (\_| | |\_| | | | | | (\_| |

   |\_| |\_| |\_|\\_\_,\_|\_| |\_|\_|\\_\  \\_\_, |\\_\_\_/ \\_\_,\_| |\_|  \\_\_\_/|\_|    |\_|   |\_|\\_\_,\_|\\_\_, |\_|\_| |\_|\\_\_, |

                                 |\_\_\_/                                              |\_\_\_/         |\_\_\_/

                                      \_\_\_  \_\_  \_\_ \_\_\_   \_

                                     / \_ \|  \/  |\_ \_| | |

                                    | | | | |\/| || |  | |

                                    | |\_| | |  | || |  |\_|

                                     \\_\_\_/|\_|  |\_|\_\_\_| (\_)

        """

    )

Game\_logic.py – It is where the majority of the game phase is implemented.

"""

This is where the game logic is avilable dor each scenario:

1) When the computer leads

2) When the player leads

"""

from deck import check\_trick\_winner

from validate\_func import validate\_for\_Card\_in\_Deck, validate\_trump\_suit, validate\_trick\_play

from display\_func import display\_hand, display\_welcome\_msg, display\_player

from computer import computer\_play\_card, get\_lowest\_card, choose\_trump, play\_card

# This is when player leads a trick

*def* player\_lead(*game\_deck*, *trick\_count*, *player\_hand*, *computer\_hand*, *Trump\_Card*):

    # This is when the player enters the card and it is checked whether card is in player's hand

    player\_card = validate\_for\_Card\_in\_Deck(trick\_count, Trump\_Card, player\_hand)

    # remove the selected card from players hand

    player\_hand.remove(player\_card)

    # this when the computer plays its card relaative to players card

    computer\_card = computer\_play\_card(computer\_hand, player\_card, Trump\_Card)

    # remove the selected card from computers' hand

    computer\_hand.remove(computer\_card)

    print("\n\n")

    # Display message with trick number, the cards played and the current player hand

    display\_player(trick\_count, Trump\_Card, player\_hand, computer\_card, player\_card)

    # print who won the trick

    winner = check\_trick\_winner(player\_card, computer\_card, Trump\_Card)

    print(winner)

    # return who won the trick

    return winner

*def* computer\_lead(*game\_deck*, *Trick\_count*, *player\_hand*, *computer\_hand*, *Trump\_Card*):

    # computer leads the trick

    computer\_card = play\_card(computer\_hand)

    # remove the selected card

    computer\_hand.remove(computer\_card)

    # Display message with trick number, the cards played and the current player hand

    display\_player(Trick\_count, Trump\_Card, player\_hand, computer\_card)

    # Enter the player choice and check if card is in deck

    player\_card = validate\_for\_Card\_in\_Deck(Trick\_count, Trump\_Card, player\_hand, computer\_card)

    # check whether the player followed the OMI rules

    valid = validate\_trick\_play(computer\_card, player\_card, player\_hand)

    # below repeats above process until all conditions are met

    while not valid:

        display\_player(Trick\_count, Trump\_Card, player\_hand, computer\_card)

        print(f"Follow the rules, cannot put a card of different suit if you have same suit.\nPlay card with suit {computer\_card[1]}")

        player\_card = validate\_for\_Card\_in\_Deck(Trick\_count, Trump\_Card, player\_hand, computer\_card)

        valid = validate\_trick\_play(computer\_card, player\_card, player\_hand)

    # remove the card the player played

    player\_hand.remove(player\_card)

    # Display message with trick number, the cards played and the current player hand

    display\_player(Trick\_count, Trump\_Card, player\_hand, computer\_card, player\_card)

    # print who won the trick

    winner = check\_trick\_winner(player\_card, computer\_card, Trump\_Card, "computer")

    print(winner)

    # return who won the trick

    return winner

validate\_func.py – Stores all function related to validation and sanitization.

"""

This is where most of the validastion function are written:

eg. For to check user follows rules and to sanitize user input

"""

import re

from display\_func import display\_player, display\_hand

*def* validate\_trick\_play(*computer\_Card*, *player\_card*, *player\_deck*):

    # check if player puts same suit as computer

    # check if player has card from the suit which the computer selected, if so then return false

    # if the player does not ave then he can play any card

    if computer\_Card[1] != player\_card[1]:

        suit = computer\_Card[1]

        card\_With\_suit = [c for c in player\_deck if c[1] == suit]

        if len(card\_With\_suit) > 0:

            return False

    return True

*def* sanitize\_user\_input(*card*, *upper\_bound*=2, \*\**kwargs*):

    # The reason to include kwargs was that of a special case when the card number is 10

    # When the card is 10, I check for a certain parameter in kwargs and add an extra character instead of the usual upper bpund

    # Functionality to remove all spaces : leading, trailing and all irregular spaces in between

    # used regex as it was cleaner and easier to implement

    card = re.sub(*r*'\s\*', '', card)

    card = card.capitalize()

    sanitized\_card = card

    if len(card) > upper\_bound:

        # Small improvement to get the specifed length, the upperbound,  characters of card after removing spaces

        sanitized\_card = card[:upper\_bound]

        if(kwargs and kwargs["c\_in\_deck"] == True):

            if(card[1] == '0'):

                # eg: If user inputs "    10   ♠opfp c"

                # it will sanitize to "10♠"

                sanitized\_card = card[:upper\_bound+1]

    return sanitized\_card

*def* check\_card\_in\_hand(*current\_hand*, *card*):

    # Check if the card is in list: check if atleast one true, meaning card in deck then return true else false

    in\_deck = any((card == x[0]+x[1] for x in current\_hand))

    return(in\_deck)

*def* validate\_trump\_suit():

    suits = ("♠", "♣", "♥", "♦")

    # check if the trump selected is a valid suit

    while True:

        trump\_selection = input("Please enter the trump suit:\n")

        trump\_selection = sanitize\_user\_input(trump\_selection, 1)

        if trump\_selection in suits:

            break

        print("\nPlease choose a valid suit as Trumps\n")

    return trump\_selection

*def* validate\_for\_Card\_in\_Deck(*Trick\_count*, *Trump\_Card*, *player\_hand*, *computer\_card*="---"):

    # check if the card selected by player is in their deck

    while True:

        card\_chosen = input("Please enter the Card for this trick:\n")

        card\_chosen = sanitize\_user\_input(card\_chosen, 2, *c\_in\_deck*=True)

        if check\_card\_in\_hand(player\_hand, card\_chosen) == True:

            if len(card\_chosen) == 2:

                return (card\_chosen[0], card\_chosen[1])

            else:

                # special case when a number card of 10 is selected

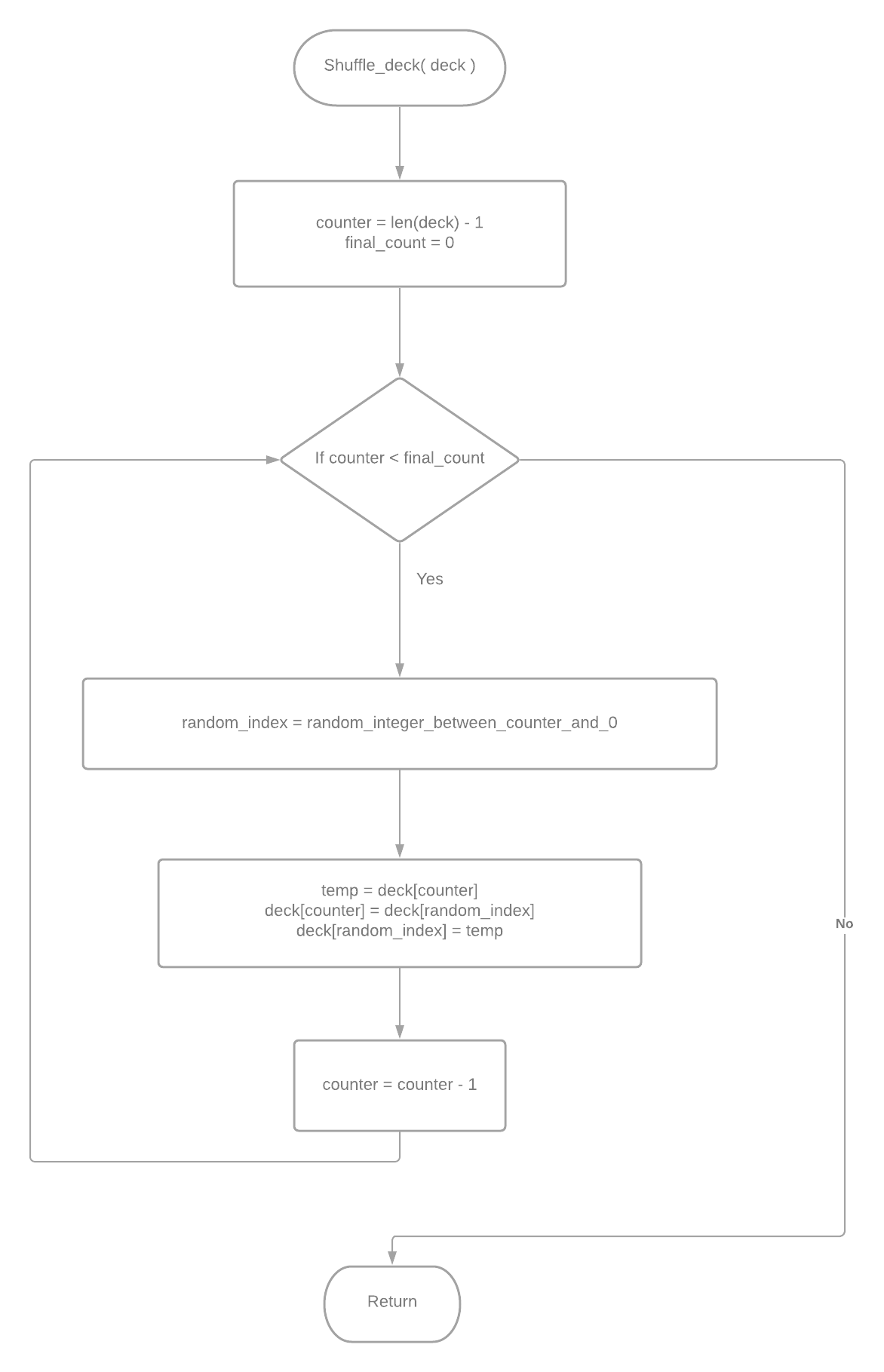
                return (card\_chosen[0:len(card\_chosen)-1], card\_chosen[-1])

            break

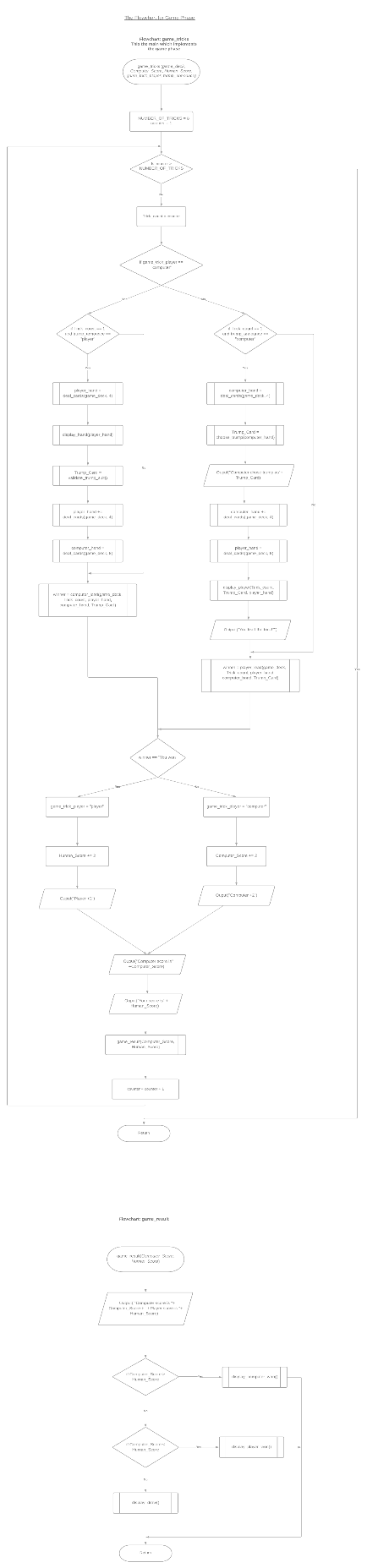
        print("\nPlease choose a valid card in your deck!\n")

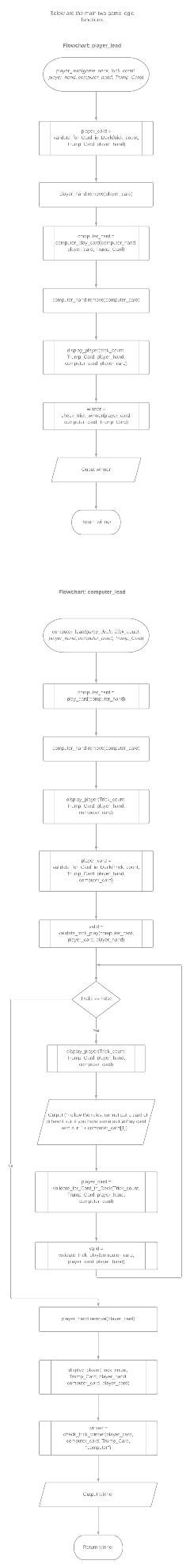
        # show user the trump and the available cards

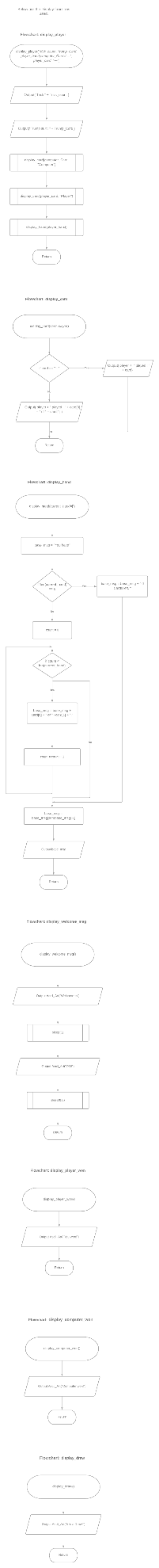
        display\_player(Trick\_count, Trump\_Card, player\_hand, computer\_card)

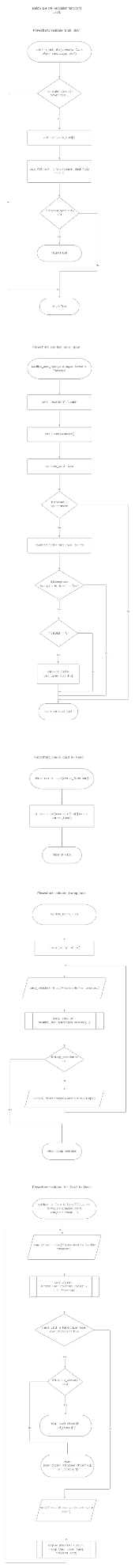
 The shuffling algorithm Flowchart

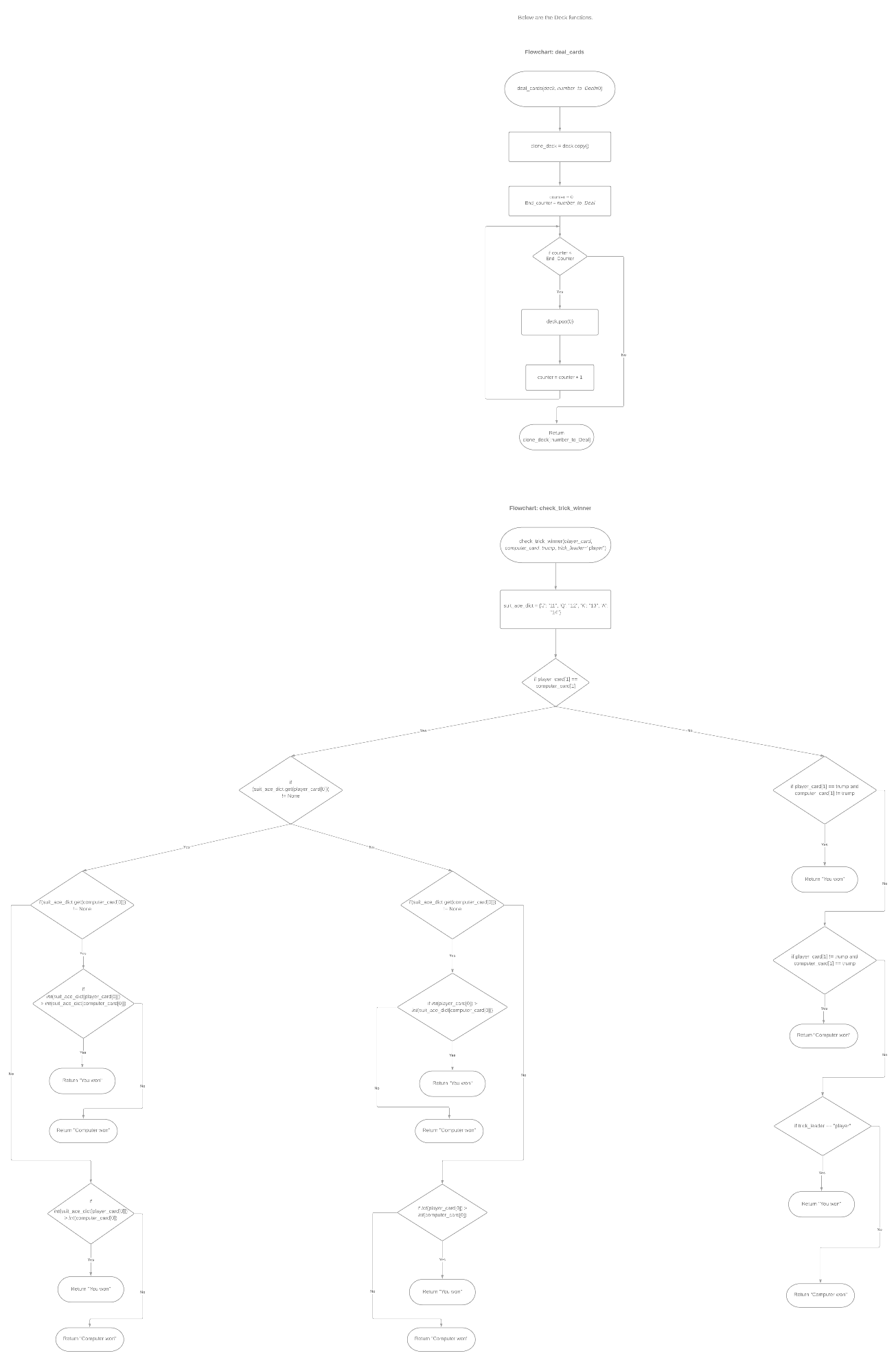
Game Phase Flowcharts

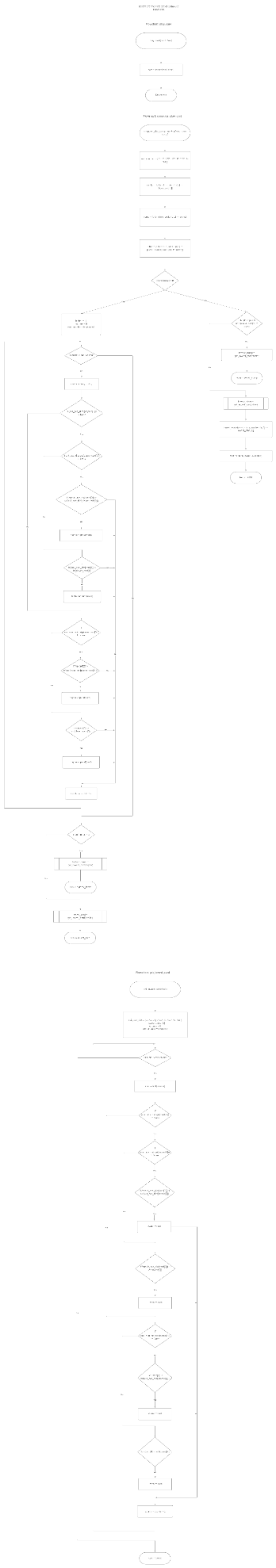


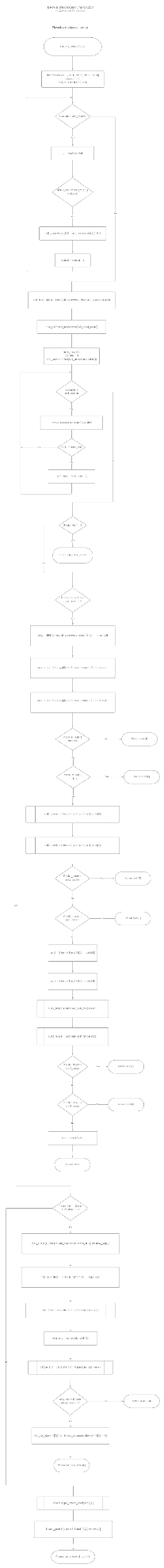


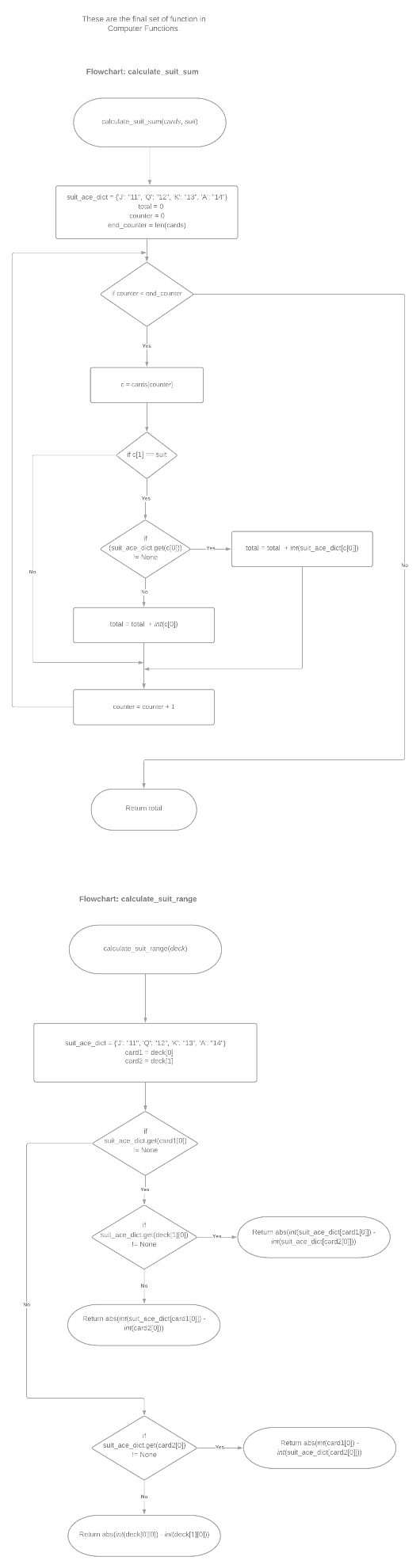


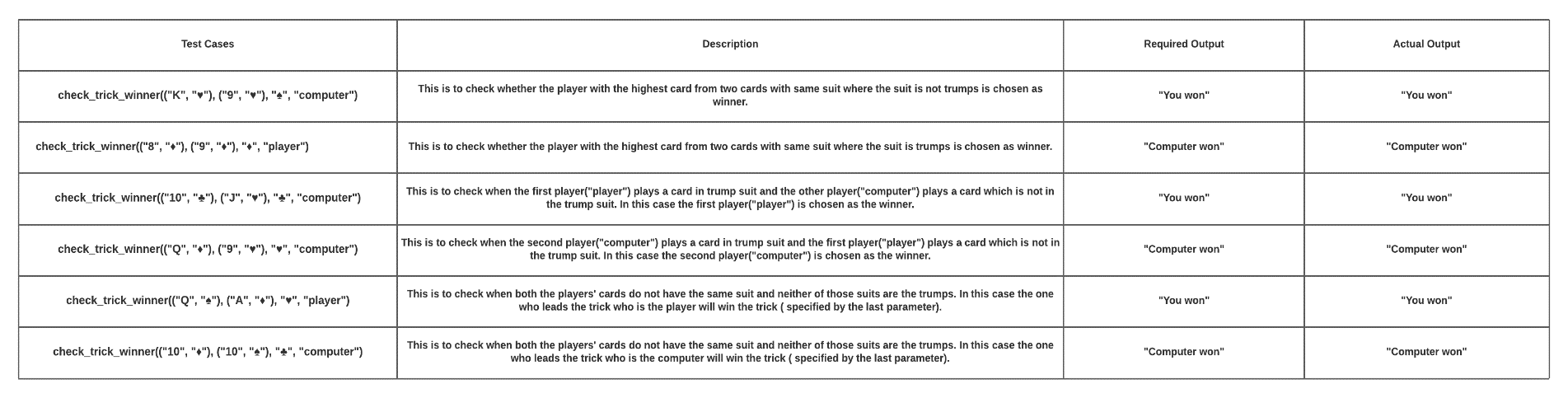










Winner Test Cases