```python
"""
Models a single card
"""

JACK = 11
QUEEN = 12
KING = 13
ACE = 14
HEARTS = "H"
DIAMONDS = "D"
SPADES = "S"

class Card:

    def __init__(self, rank, suit):
        """
        Constructor

        :param rank: an integer for the rank of the card
        :param suit: a string for the suit of the card
        """
        self.__card = {"rank": rank, "suit": suit}

    def get_rank(self):
        """
        Gets the rank of the card

        :return: an integer for the rank of the card
        """
        return self.__card["rank"]

    def get_suit(self):
        """
        Gets the suit of the card

        :return: an integer for the suit of the card
        """
        return self.__card["suit"]

    def __str__(self):
        """
        Returns the readable version of the card

        :return: a string for the readable version of the card
        """
        rank = self.get_rank()
        suit = self.get_suit()

        if rank == JACK:
            rank_string = "Jack"

        elif rank == QUEEN:
            rank_string = "Queen"

        elif rank == KING:
            rank_string = "King"

        elif rank == ACE:
            rank_string = "Ace"

        else:
            rank_string = str(rank)

        if suit == HEARTS:
            suit_string = "Hearts"

        elif suit == DIAMONDS:
            suit_string = "Diamonds"

        elif suit == SPADES:
            suit_string = "Spades"

        else:
            suit_string = "Clubs"

        return rank_string + " of " + suit_string

def __confirm_result():
    card = Card(11, "C")

    print(card)

```

```python
if __name__ == "__main__":
    __confirm_result()
```

```python
"""
Models a deck of cards
"""

from card import *
import random

RANKS = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
SUITS = ["H", "D", "S", "C"]
EMPTY = 0
TOP_CARD = 0
MAX_HAND = 5

class Deck:

    def __init__(self):
        """
        Constructor
        """
        self.__deck = []

        for rank in RANKS:
            for suit in SUITS:
                self.__deck.append(Card(rank, suit))

        self.__shuffle()

    def __shuffle(self):
        """
        Shuffles the deck

        :return: a list for the shuffled deck
        """
        return random.shuffle(self.__deck)

    def __deal(self):
        """
        Deals and returns a single card by removing the top card of the deck

        :return: a Card object which is the top card of the
```

```python
deck
        """
        if self.size() == EMPTY:
            return None

        else:
            return self.__deck.pop(TOP_CARD)

    def size(self):
        """
        Returns the number of cards left in the deck

        :return: an integer for cards left in the deck
        """
        return len(self.__deck)

    def __str__(self):
        """
        Returns the readable version of the deck

        :return: a string for the readable version of the deck
        """
        return_string = ""

        for card in self.__deck:
            return_string += str(card) + "\n"

        return return_string

    def list_o_cards(self):
        """
        Creates a list of Card objects

        :return: a list of Card objects
        """
        list_o_cards = []

        for i in range(MAX_HAND):
            list_o_cards.append(self.__deal())

        return list_o_cards
```

```python

def __confirm_result():
    deck = Deck()

    print(deck)

if __name__ == "__main__":
    __confirm_result()
```

```python
"""
Testing utilities.  Do not modify this file!
"""

VERBOSE = True
num_pass = 0
num_fail = 0

def assert_equals(msg, expected, actual):
    """
    Check whether code being tested produces
    the correct result for a specific test
    case. Prints a message indicating whether
    it does.
    :param: msg is a message to print at the beginning.
    :param: expected is the correct result
    :param: actual is the result of the
    code under test.
    """
    if VERBOSE:
        print(msg)

    global num_pass, num_fail

    if expected == actual:
        if VERBOSE:
            print("PASS")
        num_pass += 1
    else:
        if not VERBOSE:
            print(msg)
        print("**** FAIL")
        print("expected: " + str(expected))
        print("actual: " + str(actual))
        if not VERBOSE:
            print("")
        num_fail += 1

    if VERBOSE:
        print("")

```

```python

def fail_on_error(msg,err):
    """
    if run-time error occurs, call this to insta-fail

    :param msg: message saying what is being tested
    :param err: type of run-time error that occurred
    """
    global num_fail
    print(msg)
    print("**** FAIL")
    print(err)
    print("")
    num_fail += 1


def start_tests(header):
    """
    Initializes summary statistics so we are ready to run tests using
    assert_equals.
    :param header: A header to print at the beginning
    of the tests.
    """
    global num_pass, num_fail
    print(header)
    for i in range(0,len(header)):
        print("=",end="")
    print("")
    num_pass = 0
    num_fail = 0

def finish_tests():
    """
    Prints summary statistics after the tests are complete.
    """
    print("Passed %d/%d" % (num_pass, num_pass+num_fail))
    print("Failed %d/%d" % (num_fail, num_pass+num_fail))
    print()
```

```python
"""
A simple pocker game

:author: Chris Hegang Kim
:note: I affirm that I have carried out the attached academic endeavors with full academic honesty,
in accordance with the Union College Honor Code and the course syllabus.
"""

from poker_hand import *

CONTINUE = True
TWO_MAX_HAND = 10

def main():
    deck = Deck()
    continue_game = CONTINUE
    total_point = 0

    while continue_game and deck.size() > TWO_MAX_HAND:
        myhand = PokerHand(deck.list_o_cards())
        other_hand = PokerHand(deck.list_o_cards())
        result = myhand.get_result(other_hand)

        print("my hand: ", myhand)
        print("other hand: ", other_hand)

        player_input = input("Who is the winner? (Type my hand, other hand, or tie)")

        if player_input == result:
            total_point += 1

        else:
           continue_game = not CONTINUE

    print("Game is over, and your total point is ", total_point)

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

```python
"""
Models a 5-card hand of cards
"""

import copy
from deck import *

TIE = 0
MAX_HAND = 5
FIRST_CARD = 0
LAST_CARD = 1
FLUSH = 4
TWO_PAIR = 3
PAIR = 2
HIGH_CARD = 1

class PokerHand:

    def __init__(self, list_o_cards):
        """
        Constructor

        :param list_o_cards: a list of Card objects
        """
        self.__hand = copy.deepcopy(list_o_cards)

    def __add_card(self, card):
        """
        Adds the card to the hand

        :param card: a Card object
        :return: the PokerHand object with the additional card
        """
        return self.__hand.append(card)

    def __get_ith_card(self, index):
        """
        Returns the card at the given index

        :param index: an integer greater or equal to 0
        :return: a Card object at the given index
        """
        return self.__hand[index]

    def __str__(self):
        """
        Returns the readable version of the hand

        :return: a string for the readable version of the hand
        """
        return_string = ""

        for card in self.__hand:
            return_string += str(card) + "\n"

        return return_string

    def compare_to(self, other_hand):
        """
        Determines which of two poker hands is worth more. Returns an int
        which is either positive, negative, or zero depending on the comparison.

        :param self: The first hand to compare
        :param other_hand: The second hand to compare
        :return: a negative number if self is worth LESS than other_hand,
        zero if they are worth the SAME (a tie), and a positive number if
        self is worth MORE than other_hand
        """
        hand_point = self.__get_point()
        other_point = other_hand.__get_point()

        result = hand_point - other_point

        if result == TIE:
            if hand_point == FLUSH or hand_point == HIGH_CARD:
                self.__get_rank_list()
                other_hand.__get_rank_list()

                for i in range (MAX_HAND):
                    result = self.__get_ith_rank(i) - other_hand.__get_ith_rank(i)

                    if result != TIE:
                        return result

            elif hand_point == TWO_PAIR or hand_point == PAIR:

                for i in range (self.__rank_list_size()):
                    result = self.__get_ith_rank(i) - other_hand.__get_ith_rank(i)

                    if result != TIE:
                        return result

        return result

    def __get_point(self):
        """
        Gets the point of the hand

        :return: an integer for the score of the hand
        """
        if self.__is_flush():
            hand_score = FLUSH

        elif self.__is_two_pair():
            hand_score = TWO_PAIR

        elif self.__is_pair():
            hand_score = PAIR

        else:
            hand_score = HIGH_CARD

        return hand_score

    def __get_rank_list(self):
        """
        Gets the list with ranks in the descending order

        :return:
        """
        rank_list = []

        for card in self.__hand:
            rank_list.append(card.get_rank())

        rank_list.sort(reverse = True)

        self.__rank_list = rank_list

    def __get_ith_rank(self, index):
        """
        Returns the rank at the given index
        :param index: an inter for the index of the rank
        :return: an integer at the given index
        """
        return self.__rank_list[index]

    def __rank_list_size(self):
        """
        Returns the size of the rank list

        :return: an integer for the size of the rank list
        """
        return len(self.__rank_list)

    def __copy(self):
        """
        Stores a copy of the original value

        :param original: a value for the copy
        :return: a copied value with the new reference
        """
        return copy.deepcopy(self)

    def __size(self):
        """
        Returns the number of cards left in the hand

        :return: an integer for cards left in the hand
        """
        return len(self.__hand)

    def __remove_ith_card(self, index):
        """
        Removes the card with the given index

        :param index: an integer for the index of the card
        :return: a PokerHand object without the removed card
        """
        return self.__hand.remove(self.__hand[index])

    def __is_flush(self):
        """
        Checks whether the hand is flush

        :return: True if all cards have the same suit
        """
        for i in range (1, MAX_HAND):
            if self.__get_ith_card(i).get_suit() ≠ self.
__get_ith_card(i - 1).get_suit():
                return False

        return True

    def __is_two_pair(self):
        """
        Checks whether the hand is two pair

        :return: True if the hand has 2 pairs of the same
rank
        """
        current_hand = self.__copy()

        i = 1
        total_pair = 0
        pair_rank = []
        other_rank = []

        while current_hand.__size() > i:
            if current_hand.__get_ith_card(FIRST_CARD).get_rank() == current_hand.__get_ith_card(i).get_rank():
                total_pair += 1

                pair_rank.append(current_hand.__get_ith_card(FIRST_CARD).get_rank())

                current_hand.__remove_ith_card(i)
                current_hand.__remove_ith_card(FIRST_CARD)

                i = 1

            else:
                i += 1

            if current_hand.__size() == i:
                other_rank.append(current_hand.__get_ith_card(FIRST_CARD).get_rank())

                current_hand.__remove_ith_card(FIRST_CARD)

                i = 1

            if current_hand.__size() == LAST_CARD:
                other_rank.append(current_hand.__get_ith_card(FIRST_CARD).get_rank())

                current_hand.__remove_ith_card(FIRST_CARD)

        if total_pair == 2:
            pair_rank.sort(reverse = True)
            other_rank.sort(reverse = True)

            pair_rank.extend(other_rank)

            self.__rank_list = pair_rank

            return True

        return False
```

```python

    def __is_pair(self):
        """
        Checks whether the hand is a pair

        :return: True if the hand has a pair of the same rank
        """
        current_hand = self.__copy()

        i = 1
        total_pair = 0
        pair_rank = []
        other_rank = []

        while current_hand.__size() > i:
            if current_hand.__get_ith_card(FIRST_CARD).get_rank() == current_hand.__get_ith_card(i).get_rank():
                total_pair += 1

                pair_rank.append(current_hand.__get_ith_card(FIRST_CARD).get_rank())

                current_hand.__remove_ith_card(i)
                current_hand.__remove_ith_card(FIRST_CARD)

                i = 1

            else:
                i += 1

            if current_hand.__size() == i:
                other_rank.append(current_hand.__get_ith_card(FIRST_CARD).get_rank())

                current_hand.__remove_ith_card(FIRST_CARD)

                i = 1

            if current_hand.__size() == LAST_CARD:
                other_rank.append(current_hand.__get_ith_card(FIRST_CARD).get_rank())
```

```python

                    current_hand.__remove_ith_card(FIRST_CARD)

        if total_pair == 1:
            pair_rank.sort(reverse = True)
            other_rank.sort(reverse = True)

            pair_rank.extend(other_rank)

            self.__rank_list = pair_rank

            return True

        return False

    def get_result(self, other_hand):
        """
        Gets the result according to the given value

        :return: a string according to the given value
        """
        result = self.compare_to(other_hand)

        if result > 0:
            return "my hand"

        elif result < 0:
            return "other hand"

        else:
            return "tie"

def __confirm_result():
    deck = Deck()
    myhand = PokerHand(deck.list_o_cards())
    other_hand = PokerHand(deck.list_o_cards())
    result = myhand.get_result(other_hand)

    print("my hand: ", myhand)
    print("other hand: ", other_hand)
    print("result: ", result, " won.")
```

```python

if __name__ == "__main__":
    __confirm_result()
```

```python
from testing import *
from poker_hand import *

def __test_poker_hand():
    start_tests("Starts testing poker_hand module")
    __test_compare_to()
    finish_tests()

def __test_compare_to():
    __flush_v_two_pair()
    __flush_v_flush_high_card1()
    __flush_v_flush_high_card2()
    __flush_v_flush_high_card3()
    __flush_v_flush_high_card4()
    __flush_v_flush_high_card5()
    __flush_v_flush_tie()

    __two_pair_v_pair()
    __two_pair_v_two_pair_high_card1()
    __two_pair_v_two_pair_high_card2()
    __two_pair_v_two_pair_high_card3()
    __two_pair_v_two_pair_tie()

    __pair_v_high_card()
    __pair_v_pair_high_card1()
    __pair_v_pair_high_card2()
    __pair_v_pair_high_card3()
    __pair_v_pair_high_card4()
    __pair_v_pair_tie()

    __high_card_v_high_card1()
    __high_card_v_high_card2()
    __high_card_v_high_card3()
    __high_card_v_high_card4()
    __high_card_v_high_card5()
    __high_card_v_high_card_tie()

def __flush_v_two_pair():
    hand = PokerHand([Card(14, "H"), Card(14, "H"), Card(8, "H
"), Card(7, "H"), Card(10, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(14, "D"), Card
(8, "S"), Card(8, "C"), Card(9, "H")])
    msg = "Starts testing flush vs two pair"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __flush_v_flush_high_card1():
    hand = PokerHand([Card(14, "H"), Card(13, "H"), Card(12, "
H"), Card(11, "H"), Card(10, "H")])
    other_hand = PokerHand([Card(13, "H"), Card(12, "H"), Card
(11, "H"), Card(10, "H"), Card(9, "H")])
    msg = "Starts testing flush vs flush (first high card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __flush_v_flush_high_card2():
    hand = PokerHand([Card(14, "H"), Card(13, "H"), Card(12, "
H"), Card(11, "H"), Card(10, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(12, "H"), Card
(11, "H"), Card(10, "H"), Card(9, "H")])
    msg = "Starts testing flush vs flush (second high card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __flush_v_flush_high_card3():
    hand = PokerHand([Card(14, "H"), Card(13, "H"), Card(12, "
H"), Card(11, "H"), Card(10, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(13, "H"), Card
(11, "H"), Card(10, "H"), Card(9, "H")])
    msg = "Starts testing flush vs flush (third high card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __flush_v_flush_high_card4():
     hand = PokerHand([Card(14, "H"), Card(13, "H"), Card(12, "H"), Card(11, "H"), Card(10, "H")])
     other_hand = PokerHand([Card(14, "H"), Card(13, "H"), Card(12, "H"), Card(10, "H"), Card(9, "H")])
     msg = "Starts testing flush vs flush (fourth high card)"
     expected = 1
     actual = hand.compare_to(other_hand)

     assert_equals(msg, expected, actual)

 def __flush_v_flush_high_card5():
     hand = PokerHand([Card(14, "H"), Card(13, "H"), Card(12, "H"), Card(11, "H"), Card(10, "H")])
     other_hand = PokerHand([Card(14, "H"), Card(13, "H"), Card(12, "H"), Card(11, "H"), Card(9, "H")])
     msg = "Starts testing flush vs flush (fifth high card)"
     expected = 1
     actual = hand.compare_to(other_hand)

     assert_equals(msg, expected, actual)

 def __flush_v_flush_tie():
     hand = PokerHand([Card(14, "H"), Card(13, "H"), Card(12, "H"), Card(11, "H"), Card(10, "H")])
     other_hand = PokerHand([Card(14, "H"), Card(13, "H"), Card(12, "H"), Card(11, "H"), Card(10, "H")])
     msg = "Starts testing flush vs flush (tie)"
     expected = 0
     actual = hand.compare_to(other_hand)

     assert_equals(msg, expected, actual)


 def __two_pair_v_pair():
     hand = PokerHand([Card(14, "H"), Card(14, "D"), Card(8, "S"), Card(8, "C"), Card(10, "H")])
     other_hand = PokerHand([Card(14, "H"), Card(14, "D"), Card(8, "S"), Card(9, "C"), Card(10, "H")])
     msg = "Starts testing two pair vs pair"
     expected = 1
     actual = hand.compare_to(other_hand)
```

```python

    assert_equals(msg, expected, actual)


def __two_pair_v_two_pair_high_card1():
    hand = PokerHand([Card(14, "H"), Card(14, "H"), Card(8, "H"), Card(8, "H"), Card(9, "H")])
    other_hand = PokerHand([Card(13, "H"), Card(13, "H"), Card(11, "H"), Card(11, "H"), Card(9, "H")])
    msg = "Starts testing two pair vs two pair (first high pair card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)


def __two_pair_v_two_pair_high_card2():
    hand = PokerHand([Card(14, "H"), Card(14, "H"), Card(8, "H"), Card(8, "H"), Card(9, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(14, "H"), Card(7, "H"), Card(7, "H"), Card(9, "H")])
    msg = "Starts testing two pair vs two pair (second high pair card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)


def __two_pair_v_two_pair_high_card3():
    hand = PokerHand([Card(14, "H"), Card(14, "H"), Card(8, "H"), Card(8, "H"), Card(10, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(14, "H"), Card(8, "H"), Card(8, "H"), Card(9, "H")])
    msg = "Starts testing two pair vs two pair (third high card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)
```

```python


def __two_pair_v_two_pair_tie():
    hand = PokerHand([Card(14, "H"), Card(13, "H"), Card(12, "H"), Card(11, "H"), Card(10, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(13, "H"), Card(12, "H"), Card(11, "H"), Card(10, "H")])
    msg = "Starts testing two pair vs two pair (tie)"
    expected = 0
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __pair_v_high_card():
    hand = PokerHand([Card(14, "H"), Card(14, "D"), Card(8, "S"), Card(9, "C"), Card(10, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(13, "D"), Card(8, "S"), Card(9, "C"), Card(10, "H")])
    msg = "Starts testing pair vs high card"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __pair_v_pair_high_card1():
    hand = PokerHand([Card(14, "H"), Card(14, "D"), Card(8, "S"), Card(9, "C"), Card(10, "H")])
    other_hand = PokerHand([Card(13, "H"), Card(13, "D"), Card(8, "S"), Card(9, "C"), Card(10, "H")])
    msg = "Starts testing pair vs pair (first high pair card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __pair_v_pair_high_card2():
    hand = PokerHand([Card(14, "H"), Card(14, "D"), Card(8, "S"), Card(9, "C"), Card(11, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(14, "D"), Card(8, "S"), Card(9, "C"), Card(10, "H")])
    msg = "Starts testing pair vs pair (second high card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __pair_v_pair_high_card3():
    hand = PokerHand([Card(14, "H"), Card(14, "D"), Card(8, "S"), Card(10, "C"), Card(11, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(14, "D"), Card(8, "S"), Card(9, "C"), Card(11, "H")])
    msg = "Starts testing pair vs pair (third high card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __pair_v_pair_high_card4():
    hand = PokerHand([Card(14, "H"), Card(14, "D"), Card(9, "S"), Card(10, "C"), Card(11, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(14, "D"), Card(8, "S"), Card(10, "C"), Card(11, "H")])
    msg = "Starts testing pair vs pair (fourth high card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __pair_v_pair_tie():
    hand = PokerHand([Card(14, "H"), Card(14, "D"), Card(9, "S"), Card(10, "C"), Card(11, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(14, "D"), Card(9, "S"), Card(10, "C"), Card(11, "H")])
    msg = "Starts testing pair vs pair (tie)"
    expected = 0
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)


def __high_card_v_high_card1():
    hand = PokerHand([Card(14, "H"), Card(13, "D"), Card(12, "S"), Card(11, "C"), Card(10, "H")])
    other_hand = PokerHand([Card(13, "H"), Card(12, "D"), Card(11, "S"), Card(10, "C"), Card(9, "H")])
    msg = "Starts testing high card vs high card (first high card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __high_card_v_high_card2():
    hand = PokerHand([Card(14, "H"), Card(13, "D"), Card(12, "S"), Card(11, "C"), Card(10, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(12, "D"), Card(11, "S"), Card(10, "C"), Card(9, "H")])
    msg = "Starts testing high card vs high card (second high card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __high_card_v_high_card3():
    hand = PokerHand([Card(14, "H"), Card(13, "D"), Card(12, "S"), Card(11, "C"), Card(10, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(13, "D"), Card(11, "S"), Card(10, "C"), Card(9, "H")])
    msg = "Starts testing high card vs high card (third high card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __high_card_v_high_card4():
    hand = PokerHand([Card(14, "H"), Card(13, "D"), Card(12, "S"), Card(11, "C"), Card(10, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(13, "D"), Card(12, "S"), Card(10, "C"), Card(9, "H")])
    msg = "Starts testing high card vs high card (fourth high
```

```python
card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __high_card_v_high_card5():
    hand = PokerHand([Card(14, "H"), Card(13, "D"), Card(12, "S"), Card(11, "C"), Card(10, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(13, "D"), Card(12, "S"), Card(11, "C"), Card(9, "H")])
    msg = "Starts testing high card vs high card (fifth high card)"
    expected = 1
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

def __high_card_v_high_card_tie():
    hand = PokerHand([Card(14, "H"), Card(13, "D"), Card(12, "S"), Card(11, "C"), Card(10, "H")])
    other_hand = PokerHand([Card(14, "H"), Card(13, "D"), Card(12, "S"), Card(11, "C"), Card(10, "H")])
    msg = "Starts testing high card vs high card (tie)"
    expected = 0
    actual = hand.compare_to(other_hand)

    assert_equals(msg, expected, actual)

if __name__ == "__main__":
    __test_poker_hand()
```