```python
RANK_NAMES = {
    11: "Jack",
    12: "Queen",
    13: "King",
    14: "Ace"
}

SUIT_NAMES = {
    "H": "Hearts",
    "D": "Diamonds",
    "C": "Clubs",
    "S": "Spades"
}

class Card:
    def __init__(self, rank, suit):
        self.__card = {"rank": rank, "suit": suit}

    def get_rank(self):
        """
        Get the rank of the card.
        :return: the rank of the card
        """
        return self.__card["rank"]

    def get_suit(self):
        """
        Get the suit of the card.
        :return: the suit of the card
        """
        return self.__card["suit"]

    def __str__(self):
        """
        Return a string representation of the card.
        :return: a string representation of the card
        """
        rank_str = RANK_NAMES.get(self.get_rank(), str(
self.get_rank()))
        suit_str = SUIT_NAMES.get(self.get_suit(), self.get_suit())
        return f"{rank_str} of {suit_str}"



# if __name__ == "__main__":
#     card1 = Card(12, "D")
#     card2 = Card(10, "S")
#     print(card1)
#     print(card2)
```

```python
import random
from card import Card

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

class Deck:

    def __init__(self):
        self.__deck = []
        for suit in SUITS:
            for rank in RANKS:
                self.__deck.append(Card(rank, suit))

    def shuffle(self):
        """
        Shuffles the deck.
        """
        random.shuffle(self.__deck)

    def deal(self):
        """
        Deals a card from the deck.
        :return: the card dealt
        """
        if len(self.__deck) == 0:
            return None
        else:
            return self.__deck.pop(0)

    def size(self):
        """
        Get the size of the deck.
        :return: the size of the deck
        """
        return len(self.__deck)

    def __str__(self):
        """
        Return a string representation of the deck.
        :return: a string representation of the deck
        """
        s = ""
        for card in self.__deck:
            s += str(card) + "\n"
        return s

# if __name__ == "__main__":
#     deck = Deck()
#     print(deck.size())
#     print(deck)
```

```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("")


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
"""
Author: James Lin
Date: 10/5/2025

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 deck import Deck

from poker_hand import PokerHand

HAND_SIZE = 5


def deal_two_hands(deck):
    """
    Deal two hands from the deck.
    :param deck: the deck to deal from
    :return: two lists of cards
    """
    cards1 = []
    cards2 = []
    for _ in range(HAND_SIZE):
        cards1.append(deck.deal())
    for _ in range(HAND_SIZE):
        cards2.append(deck.deal())
    return cards1, cards2


def main():
    """
    1. Draws two new hands from a given deck.
    2. Shows the hands to the player, asking them who the winner is (or if there's a tie)
    3. Tells if the player is correct or incorrect. Go
```

```
to step 1.
    4. The game is over if there are not enough cards left to play another round.
    """

    deck = Deck()
    deck.shuffle()
    game_count = 1
    win_rate = 0


    while deck.size() >= HAND_SIZE * 2:
        print(f"Round {game_count}")
        print("-" * 50)

        cards1, cards2 = deal_two_hands(deck)
        hand1 = PokerHand(cards1)
        hand2 = PokerHand(cards2)

        print("Hand 1:")
        print(hand1)
        print("Hand 2:")
        print(hand2)

        player_guess = input("Who wins? (1 for Hand 1, 2 for Hand 2, 0 for Tie): ")

        comparison = hand1.compare_to(hand2)

        if comparison > 0:
            actual_winner = 1
            result_msg = "Hand 1 wins!"
        elif comparison < 0:
            actual_winner = 2
            result_msg = "Hand 2 wins!"
        else:
            actual_winner = 0
            result_msg = "It's a tie!"
```

```python


        player_guess = int(player_guess)
        if player_guess == actual_winner:
            print(f"Correct! {result_msg}")
            win_rate += 1
        else:
            print(f"Incorrect. {result_msg}")


        print(f"Cards remaining: {deck.size()}")
        print()

        game_count += 1

    print("=" * 50)
    print(f"Game Over! Not enough cards left to play another round.")
    print(f"You played {game_count - 1} rounds and won {win_rate} times")




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

```python
# • Flush (includes normal, royal, and straight flushes
)
# • Two pair (includes two pair, four-of-a-kind, and
full house)
# • Pair (includes pair and three-of-a-kind)
# • High card (includes high card and straight). Ace
has the highest rank and Two has the lowest.


from card import Card


CATEGORY_VALUE = {
    "flush": 3,
    "two pair": 2,
    "pair": 1,
    "high card": 0,
}

class PokerHand:
    def __init__(self, cards):
        self.cards = cards.copy()

    def add_card(self, card):
        """
        Add a card to the hand.
        :param card: the card to add
        """
        self.cards.append(card)

    def get_ith_card(self, i):
        """
        Get the i-th card from the hand.
        :param i: the index of the card to get
        :return: the card at index i
        """
        if 0 <= i < len(self.cards):
            return self.cards[i]
        return None

    def __str__(self):
        """
        Return a string representation of the hand.
        :return: a string representation of the hand
        """
        s = ""
        for card in self.cards:
            s += str(card) + "\n"
        return s

    def __is_flush(self):
        """
        Check if this hand is a flush, all cards same suit.
        :return: a boolean value indicating if the hand is a flush
        """
        suits = []
        for card in self.cards:
            suits.append(card.get_suit())
        removed = set(suits)
        return len(removed) == 1


    def __is_straight(self):
        """
        Check if this hand is a straight, consecutive cards.
        :return: a boolean value indicating if the hand is a straight
        """
        ranks = []
        for card in self.cards:
            ranks.append(card.get_rank())

        ranks.sort()
        for i in range(4):
            if ranks[i] + 1 != ranks[i + 1]:
                return False
        return True


    def __rank_counts(self):
        """
        Count how many cards of each rank are in this hand.
        :return: dictionary with rank counts
        {rank: count}
        {key: value}
        """
        counts = {}
        for card in self.cards:
            rank = card.get_rank()
            if rank in counts:
                counts[rank] += 1
            else:
                counts[rank] = 1
        return counts


    def __is_four_of_a_kind(self):
        """
        Check if this hand has four cards of the same rank.
        :return: a boolean value indicating if the hand has four cards of the same rank
        """
        counts = self.__rank_counts()
        if 4 in counts.values():
            return True
        return False


    def __is_full_house(self):
        """
        Check if this hand has three of one rank and two of another.
        :return: a boolean value indicating if the hand has three of one rank and two of another
        """
        counts = self.__rank_counts().values()
        three = False
        two = False

        for count in counts:
            if count == 3:
                three = True
            elif count == 2:
                two = True

        return three and two


    def __is_three_of_a_kind(self):
        """
        Check if this hand has exactly three cards of the same rank.
        :return: a boolean value indicating if the hand has exactly three cards of the same rank
        """
        counts = self.__rank_counts().values()
        if 3 in counts and not self.__is_full_house():
            return True
        return False


    def __has_pairs(self, n):
        """
        Check if this hand has exactly n pairs.

        precondition: n is a positive integer and n <= 2
        :param n: number of pairs to check
        :return: a boolean value indicating if the hand has exactly n pairs
        """
        counts = self.__rank_counts()
        pair_count = 0
        for value in counts.values():
            if value == 2:
                pair_count += 1
        return pair_count == n


    def __evaluate(self):
        """
        Evaluate this hand and return its category.
        Grouped into: flush, two pair, pair, or high card.
        :return: string representing the hand category
        """
        flush = self.__is_flush()
        straight = self.__is_straight()

        if flush and straight:
            return "flush"
        elif flush:
            return "flush"
        elif self.__is_four_of_a_kind() or self.__is_full_house() or self.__has_pairs(2):
            return "two pair"
        elif self.__is_three_of_a_kind() or self.__has_pairs(1):
            return "pair"
        else:
            return "high card"


    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.
        """

        my_category = self.__evaluate()
        my_value = CATEGORY_VALUE[my_category]

        other_category = other_hand.__evaluate()
        other_value = CATEGORY_VALUE[other_category]


        category_diff = my_value - other_value
        if category_diff != 0:
            return category_diff

        my_counts = self.__rank_counts()
        other_counts = other_hand.__rank_counts()

        my_list = []
        for rank in my_counts:
            my_list.append((my_counts[rank], rank))

        other_list = []
        for rank in other_counts:
            other_list.append((other_counts[rank], rank))

        my_list.sort(reverse=True)
```

```python
        other_list.sort(reverse=True)

        for i in range(len(my_list)):
            my_rank = my_list[i][1]
            other_rank = other_list[i][1]

            if my_rank != other_rank:
                return my_rank - other_rank

        return 0



# if __name__ == "__main__":
#     flush_hand = [Card("2", "H"), Card("5", "H"), Card("9", "H"), Card("K", "H"), Card("A", "H")]
#     hand1 = PokerHand(flush_hand)
#     result1 = hand1._PokerHand__evaluate()
#     print(result1)

```

```python
"""
Unit testing suite for PokerHand compare_to method
"""

from card import Card
from poker_hand import PokerHand
from testing import *


def test_different_categories():
    """Test comparing hands from different categories
"""
    print("Testing different categories: ")
    print('-'*20)
    flush_vs_high_card()
    flush_vs_pair()


def test_same_category():
    """Test comparing hands within the same category"""
    print("Testing same category: ")
    print('-'*20)
    flush_vs_flush()
    high_card_vs_high_card()
    two_pair_vs_two_pair()


def test_ties():
    """Test hands that should tie"""
    print("Testing ties: ")
    print('-'*20)
    identical_hands_tie()
    same_pair_same_other_cards_tie()


def test_complex_same_category():
    """Test complex cases within same category such as
two pair includes full house and four-of-a-kind"""
```

```python
    print("Testing complex same category: ")
    print('-'*20)
    full_house_vs_two_pair()
    four_kind_vs_full_house()


def test_compare_to():
    """Main test function that runs all compare_to tests"""
    start_tests("Testing compare_to")
    test_different_categories()
    test_same_category()
    test_ties()
    test_complex_same_category()
    finish_tests()


"""
Individual unit tests start here
"""

def flush_vs_high_card():
    """Test that flush beats high card"""
    mess = "Flush vs high card "
    flush = PokerHand([Card(2, "H"), Card(5, "H"), Card(9, "H"), Card(13, "H"), Card(14, "H")])
    high_card = PokerHand([Card(2, "D"), Card(5, "C"), Card(7, "S"), Card(9, "D"), Card(11, "C")])
    actual = flush.compare_to(high_card)
    expected = "positive"

    if actual > 0:
        actual = "positive"

    assert_equals(mess, expected, actual)


def flush_vs_pair():
     """Test that flush beats pair"""
     mess = "Flush vs pair "
     flush = PokerHand([Card(2, "D"), Card(4, "D"), Card(6, "D"), Card(8, "D"), Card(10, "D")])
     pair = PokerHand([Card(14, "H"), Card(14, "D"), Card(13, "C"), Card(12, "S"), Card(11, "H")])
     actual = flush.compare_to(pair)
     expected = "positive"

     if actual > 0:
         actual = "positive"

     assert_equals(mess, expected, actual)


 def flush_vs_flush():
     """Test comparing two flushes"""
     mess = "FLush vs flush"
     ace_flush = PokerHand([Card(2, "H"), Card(5, "H"
 ), Card(9, "H"), Card(13, "H"), Card(14, "H")])
     queen_flush = PokerHand([Card(2, "D"), Card(5, "D"
 ), Card(9, "D"), Card(12, "D"), Card(13, "D")])
     actual = ace_flush.compare_to(queen_flush)
     expected = "positive"

     if actual > 0:
         actual = "positive"

     assert_equals(mess, expected, actual)


 def high_card_vs_high_card():
     """Test comparing two high card hands"""
     mess = "High card vs high card"
     king_high = PokerHand([Card(2, "H"), Card(4, "D"
 ), Card(7, "C"), Card(9, "S"), Card(13, "H")])
     queen_high = PokerHand([Card(2, "D"), Card(4, "C"
 ), Card(7, "S"), Card(9, "D"), Card(12, "C")])
```

```python
    actual = king_high.compare_to(queen_high)
    expected = "positive"

    if actual > 0:
        actual = "positive"

    assert_equals(mess, expected, actual)


def identical_hands_tie():
    """Test that same ranks but different suits are a tie"""
    mess = "Same ranks but different suits"
    hand1 = PokerHand([Card(2, "H"), Card(5, "D"), Card(9, "C"), Card(13, "S"), Card(14, "H")])
    hand2 = PokerHand([Card(2, "D"), Card(5, "C"), Card(9, "H"), Card(13, "D"), Card(14, "C")])
    actual = hand1.compare_to(hand2)
    expected = 0

    assert_equals(mess, expected, actual)


def same_pair_same_other_cards_tie():
    """Test that same pair with same other cards are a tie"""
    mess = "Same pair, same other cards"
    hand1 = PokerHand([Card(10, "H"), Card(10, "D"), Card(5, "C"), Card(3, "S"), Card(2, "H")])
    hand2 = PokerHand([Card(10, "C"), Card(10, "S"), Card(5, "D"), Card(3, "H"), Card(2, "C")])
    actual = hand1.compare_to(hand2)
    expected = 0

    assert_equals(mess, expected, actual)


def two_pair_vs_two_pair():
    """Test comparing two two pair hands"""
    mess = "Two pair vs two pair"
    two_pair1 = PokerHand([Card(9, "H"), Card(9, "D"), Card(8, "C"), Card(8, "S"), Card(3, "H")])
    two_pair2 = PokerHand([Card(9, "C"), Card(9, "S"), Card(7, "D"), Card(7, "H"), Card(3, "C")])
    actual = two_pair1.compare_to(two_pair2)
    expected = "positive"

    if actual > 0:
        actual = "positive"

    assert_equals(mess, expected, actual)


def full_house_vs_two_pair():
    """Test that full house beats two pair"""
    mess = "Full house vs two pair"
    full_house = PokerHand([Card(5, "H"), Card(5, "D"), Card(5, "C"), Card(7, "S"), Card(7, "H")])
    two_pair = PokerHand([Card(7, "D"), Card(7, "C"), Card(6, "S"), Card(6, "H"), Card(3, "D")])
    actual = full_house.compare_to(two_pair)
    expected = "negative"

    if actual < 0:
        actual = "negative"

    assert_equals(mess, expected, actual)


def four_kind_vs_full_house():
    """Test four of a kind vs full house"""
    mess = "Four of a kind vs full house"
    four_kind = PokerHand([Card(13, "H"), Card(13, "D"), Card(13, "C"), Card(13, "S"), Card(2, "H")])
    full_house = PokerHand([Card(14, "H"), Card(14, "D"), Card(14, "C"), Card(3, "S"), Card(3, "H")])
    actual = four_kind.compare_to(full_house)
    expected = "negative"

    if actual < 0:
        actual = "negative"

    assert_equals(mess, expected, actual)


"""
Individual unit tests end here
"""

if __name__ == "__main__":
    test_compare_to()


```