

```
1 RANK_NAMES = {
2     11: "Jack",
3     12: "Queen",
4     13: "King",
5     14: "Ace"
6 }
7
8 SUIT_NAMES = {
9     "H": "Hearts",
10    "D": "Diamonds",
11    "C": "Clubs",
12    "S": "Spades"
13 }
14
15 class Card:
16     def __init__(self, rank, suit):
17         self.__card = {"rank": rank, "suit": suit}
18
19     def get_rank(self):
20         """
21             Get the rank of the card.
22             :return: the rank of the card
23         """
24         return self.__card["rank"]
25
26     def get_suit(self):
27         """
28             Get the suit of the card.
29             :return: the suit of the card
30         """
31         return self.__card["suit"]
32
33     def __str__(self):
34         """
35             Return a string representation of the card.
36             :return: a string representation of the card
37         """
38         rank_str = RANK_NAMES.get(self.get_rank(), str(
```

File - C:\Users\james\Documents\Personal\College Life\Second Year\Courses\CSC\CSC120\Projects\Project2\card.py

```
38 self.get_rank())))
39         suit_str = SUIT_NAMES.get(self.get_suit(), self
40             .get_suit())
41
42
43
44 # if __name__ == "__main__":
45 #     card1 = Card(12, "D")
46 #     card2 = Card(10, "S")
47 #     print(card1)
48 #     print(card2)
```

```
1 import random
2 from card import Card
3
4 RANKS = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
5 SUITS = ["S", "C", "H", "D"]
6
7 class Deck:
8
9     def __init__(self):
10         self.__deck = []
11         for suit in SUITS:
12             for rank in RANKS:
13                 self.__deck.append(Card(rank, suit))
14
15     def shuffle(self):
16         """
17             Shuffles the deck.
18         """
19         random.shuffle(self.__deck)
20
21     def deal(self):
22         """
23             Deals a card from the deck.
24             :return: the card dealt
25         """
26         if len(self.__deck) == 0:
27             return None
28         else:
29             return self.__deck.pop(0)
30
31     def size(self):
32         """
33             Get the size of the deck.
34             :return: the size of the deck
35         """
36         return len(self.__deck)
37
38     def __str__(self):
```

```
39         """
40     Return a string representation of the deck.
41     :return: a string representation of the deck
42     """
43     s = ""
44     for card in self._deck:
45         s += str(card) + "\n"
46     return s
47
48 # if __name__ == "__main__":
49 #     deck = Deck()
50 #     print(deck.size())
51 #     print(deck)
```

```
1 """
2 Testing utilities.  Do not modify this file!
3 """
4
5 VERBOSE = True
6 num_pass = 0
7 num_fail = 0
8
9 def assert_equals(msg, expected, actual):
10     """
11         Check whether code being tested produces
12         the correct result for a specific test
13         case. Prints a message indicating whether
14         it does.
15         :param: msg is a message to print at the beginning.
16         :param: expected is the correct result
17         :param: actual is the result of the
18         code under test.
19     """
20     if VERBOSE:
21         print(msg)
22
23     global num_pass, num_fail
24
25     if expected == actual:
26         if VERBOSE:
27             print("PASS")
28             num_pass += 1
29     else:
30         if not VERBOSE:
31             print(msg)
32         print("**** FAIL")
33         print("expected: " + str(expected))
34         print("actual: " + str(actual))
35         if not VERBOSE:
36             print("")
37             num_fail += 1
38
```

```
39     if VERBOSE:
40         print("")
41
42
43 def fail_on_error(msg,err):
44     """
45     If run-time error occurs, call this to insta-fail
46
47     :param msg: message saying what is being tested
48     :param err: type of run-time error that occurred
49     """
50
51     global num_fail
52     print(msg)
53     print("**** FAIL")
54     print(err)
55     print("")
56     num_fail += 1
57
58
59 def start_tests(header):
60     """
61     Initializes summary statistics so we are ready to
62     run tests using
63     assert_equals.
64     :param header: A header to print at the beginning
65     of the tests.
66     """
67
68     global num_pass, num_fail
69     print(header)
70     for i in range(0,len(header)):
71         print("=",end="")
72     print("")
73     num_pass = 0
74     num_fail = 0
75
76
77 def finish_tests():
78     """
79     Prints summary statistics after the tests are
```

```
75 complete.  
76     """  
77     print("Passed %d/%d" % (num_pass, num_pass+  
    num_fail))  
78     print("Failed %d/%d" % (num_fail, num_pass+  
    num_fail))  
79     print()  
80
```

```
1 """
2 Author: James Lin
3 Date: 10/5/2025
4
5 I affirm that I have carried out the attached academic
endeavors with full academic honesty,
6 in accordance with the Union College Honor Code and the
course syllabus.
7 """
8
9 from deck import Deck
10
11 from poker_hand import PokerHand
12
13 HAND_SIZE = 5
14
15
16 def deal_two_hands(deck):
17     """
18         Deal two hands from the deck.
19         :param deck: the deck to deal from
20         :return: two lists of cards
21     """
22     cards1 = []
23     cards2 = []
24     for _ in range(HAND_SIZE):
25         cards1.append(deck.deal())
26     for _ in range(HAND_SIZE):
27         cards2.append(deck.deal())
28     return cards1, cards2
29
30
31 def main():
32     """
33         1. Draws two new hands from a given deck.
34         2. Shows the hands to the player, asking them who
the winner is (or if there's a tie)
35         3. Tells if the player is correct or incorrect. Go
```

```
35 to step 1.
36     4. The game is over if there are not enough cards
37         left to play another round.
38
39     """
40
41     deck = Deck()
42     deck.shuffle()
43     game_count = 1
44     win_rate = 0
45
46     while deck.size() >= HAND_SIZE * 2:
47         print(f"Round {game_count}")
48         print("-" * 50)
49
50         cards1, cards2 = deal_two_hands(deck)
51         hand1 = PokerHand(cards1)
52         hand2 = PokerHand(cards2)
53
54         print("Hand 1:")
55         print(hand1)
56         print("Hand 2:")
57         print(hand2)
58
59         player_guess = input("Who wins? (1 for Hand 1,
60                             2 for Hand 2, 0 for Tie): ")
61
62         comparison = hand1.compare_to(hand2)
63
64         if comparison > 0:
65             actual_winner = 1
66             result_msg = "Hand 1 wins!"
67         elif comparison < 0:
68             actual_winner = 2
69             result_msg = "Hand 2 wins!"
70         else:
71             actual_winner = 0
72             result_msg = "It's a tie!"
```

```
71
72
73     player_guess = int(player_guess)
74     if player_guess == actual_winner:
75         print(f"Correct! {result_msg}")
76         win_rate += 1
77     else:
78         print(f"Incorrect. {result_msg}")
79
80
81     print(f"Cards remaining: {deck.size()}")
82     print()
83
84     game_count += 1
85
86     print("=" * 50)
87     print(f"Game Over! Not enough cards left to play
another round.")
88     print(f"You played {game_count - 1} rounds and won
{win_rate} times")
89
90
91
92
93 if __name__ == "__main__":
94     main()
```

```
1 # • Flush (includes normal, royal, and straight flushes
2 # )
3 # • Two pair (includes two pair, four-of-a-kind, and
4 # full house)
5 # • Pair (includes pair and three-of-a-kind)
6 # • High card (includes high card and straight). Ace
7 # has the highest rank and Two has the lowest.
8
9
10 CATEGORY_VALUE = {
11     "flush": 3,
12     "two pair": 2,
13     "pair": 1,
14     "high card": 0,
15 }
16
17 class PokerHand:
18     def __init__(self, cards):
19         self.cards = cards.copy()
20
21     def add_card(self, card):
22         """
23             Add a card to the hand.
24             :param card: the card to add
25         """
26         self.cards.append(card)
27
28     def get_ith_card(self, i):
29         """
30             Get the i-th card from the hand.
31             :param i: the index of the card to get
32             :return: the card at index i
33         """
34         if 0 <= i < len(self.cards):
35             return self.cards[i]
```

```
36         return None
37
38     def __str__(self):
39         """
40             Return a string representation of the hand.
41             :return: a string representation of the hand
42         """
43         s = ""
44         for card in self.cards:
45             s += str(card) + "\n"
46         return s
47
48     def __is_flush(self):
49         """
50             Check if this hand is a flush, all cards same
51             suit.
52             :return: a boolean value indicating if the hand
53             is a flush
54         """
55         suits = []
56         for card in self.cards:
57             suits.append(card.get_suit())
58         removed = set(suits)
59         return len(removed) == 1
60
61
62     def __is_straight(self):
63         """
64             Check if this hand is a straight, consecutive
65             cards.
66             :return: a boolean value indicating if the hand
67             is a straight
68         """
69         ranks = []
70         for card in self.cards:
71             ranks.append(card.get_rank())
72
73         ranks.sort()
```

```
70     for i in range(4):
71         if ranks[i] + 1 != ranks[i + 1]:
72             return False
73     return True
74
75
76     def __rank_counts(self):
77         """
78             Count how many cards of each rank are in this
79             hand.
80             :return: dictionary with rank counts
81             {rank: count}
82             {key: value}
83             """
84             counts = {}
85             for card in self.cards:
86                 rank = card.get_rank()
87                 if rank in counts:
88                     counts[rank] += 1
89                 else:
90                     counts[rank] = 1
91             return counts
92
93     def __is_four_of_a_kind(self):
94         """
95             Check if this hand has four cards of the same
96             rank.
97             :return: a boolean value indicating if the
98             hand has four cards of the same rank
99             """
100            counts = self.__rank_counts()
101            if 4 in counts.values():
102                return True
103            return False
104
105     def __is_full_house(self):
```

```
105     """
106     Check if this hand has three of one rank and
107     two of another.
108     :return: a boolean value indicating if the
109     hand has three of one rank and two of another
110     """
111     counts = self.__rank_counts().values()
112     three = False
113     two = False
114
115     for count in counts:
116         if count == 3:
117             three = True
118         elif count == 2:
119             two = True
120
121
122     def __is_three_of_a_kind(self):
123         """
124         Check if this hand has exactly three cards of
125         the same rank.
126         :return: a boolean value indicating if the
127         hand has exactly three cards of the same rank
128         """
129         counts = self.__rank_counts().values()
130         if 3 in counts and not self.__is_full_house():
131             return True
132         return False
133
134
135     def __has_pairs(self, n):
136         """
137         Check if this hand has exactly n pairs.
138
139         precondition: n is a positive integer and n
140         <= 2
```

```
138     :param n: number of pairs to check
139     :return: a boolean value indicating if the
140     hand has exactly n pairs
141     """
142     counts = self.__rank_counts()
143     pair_count = 0
144     for value in counts.values():
145         if value == 2:
146             pair_count += 1
147     return pair_count == n
148
149     def __evaluate(self):
150     """
151     Evaluate this hand and return its category.
152     Grouped into: flush, two pair, pair, or high
153     card.
154     :return: string representing the hand category
155     """
156     flush = self.__is_flush()
157     straight = self.__is_straight()
158
159     if flush and straight:
160         return "flush"
161     elif flush:
162         return "flush"
163     elif self.__is_four_of_a_kind() or self.
164     __is_full_house() or self.__has_pairs(2):
165         return "two pair"
166     elif self.__is_three_of_a_kind() or self.
167     __has_pairs(1):
168         return "pair"
169     else:
170         return "high card"
171
172     def compare_to(self, other_hand):
173     """
```

```
172     Determines which of two poker hands is worth
173     more. Returns an int
174
175     :param self: The first hand to compare
176     :param other_hand: The second hand to compare
177     :return: a negative number if self is worth
178     LESS than other_hand,
179     zero if they are worth the SAME (a tie), and a
180     positive number if
181     self is worth MORE than other_hand.
182     """
183
184
185     my_category = self.__evaluate()
186     my_value = CATEGORY_VALUE[my_category]
187
188
189     other_category = other_hand.__evaluate()
190     other_value = CATEGORY_VALUE[other_category]
191
192
193     category_diff = my_value - other_value
194     if category_diff != 0:
195         return category_diff
196
197     my_counts = self.__rank_counts()
198     other_counts = other_hand.__rank_counts()
199
200     my_list = []
201     for rank in my_counts:
202         my_list.append((my_counts[rank], rank))
203
204     other_list = []
205     for rank in other_counts:
206         other_list.append((other_counts[rank],
207                           rank))
208
209     my_list.sort(reverse=True)
```

```
205     other_list.sort(reverse=True)
206
207     for i in range(len(my_list)):
208         my_rank = my_list[i][1]
209         other_rank = other_list[i][1]
210
211         if my_rank != other_rank:
212             return my_rank - other_rank
213
214     return 0
215
216
217
218 # if __name__ == "__main__":
219 #     flush_hand = [Card("2", "H"), Card("5", "H"),
220 #                   Card("9", "H"), Card("K", "H"), Card("A", "H")]
221 #     hand1 = PokerHand(flush_hand)
222 #     result1 = hand1._PokerHand__evaluate()
223 #     print(result1)
```

```
1 """
2 Unit testing suite for PokerHand compare_to method
3 """
4
5 from card import Card
6 from poker_hand import PokerHand
7 from testing import *
8
9
10 def test_different_categories():
11     """Test comparing hands from different categories"""
12     print("Testing different categories: ")
13     print('-'*20)
14     flush_vs_high_card()
15     flush_vs_pair()
16
17
18 def test_same_category():
19     """Test comparing hands within the same category"""
20     print("Testing same category: ")
21     print('-'*20)
22     flush_vs_flush()
23     high_card_vs_high_card()
24     two_pair_vs_two_pair()
25
26
27 def test_ties():
28     """Test hands that should tie"""
29     print("Testing ties: ")
30     print('-'*20)
31     identical_hands_tie()
32     same_pair_same_other_cards_tie()
33
34
35 def test_complex_same_category():
36     """Test complex cases within same category such as
two pair includes full house and four-of-a-kind"""


```

```
37     print("Testing complex same category: ")
38     print('-'*20)
39     full_house_vs_two_pair()
40     four_kind_vs_full_house()
41
42
43 def test_compare_to():
44     """Main test function that runs all compare_to
45     tests"""
46     start_tests("Testing compare_to")
47     test_different_categories()
48     test_same_category()
49     test_ties()
50     test_complex_same_category()
51     finish_tests()
52
53 """
54 Individual unit tests start here
55 """
56
57 def flush_vs_high_card():
58     """Test that flush beats high card"""
59     mess = "Flush vs high card "
60     flush = PokerHand([Card(2, "H"), Card(5, "H"), Card
(9, "H"), Card(13, "H"), Card(14, "H")])
61     high_card = PokerHand([Card(2, "D"), Card(5, "C"),
Card(7, "S"), Card(9, "D"), Card(11, "C")])
62     actual = flush.compare_to(high_card)
63     expected = "positive"
64
65     if actual > 0:
66         actual = "positive"
67
68     assert_equals(mess, expected, actual)
69
70
71 def flush_vs_pair():
```

```
72     """Test that flush beats pair"""
73     mess = "Flush vs pair "
74     flush = PokerHand([Card(2, "D"), Card(4, "D"),
75                         Card(6, "D"), Card(8, "D"), Card(10, "D")])
76     pair = PokerHand([Card(14, "H"), Card(14, "D"),
77                        Card(13, "C"), Card(12, "S"), Card(11, "H")])
78     actual = flush.compare_to(pair)
79     expected = "positive"
80
81
82     if actual > 0:
83         actual = "positive"
84
85 def flush_vs_flush():
86     """Test comparing two flushes"""
87     mess = "Flush vs flush"
88     ace_flush = PokerHand([Card(2, "H"), Card(5, "H"),
89                           Card(9, "H"), Card(13, "H"), Card(14, "H")])
89     queen_flush = PokerHand([Card(2, "D"), Card(5, "D"),
90                             Card(9, "D"), Card(12, "D"), Card(13, "D")])
90     actual = ace_flush.compare_to(queen_flush)
91     expected = "positive"
92
93     if actual > 0:
94         actual = "positive"
95
96     assert_equals(mess, expected, actual)
97
98
99 def high_card_vs_high_card():
100    """Test comparing two high card hands"""
101    mess = "High card vs high card"
102    king_high = PokerHand([Card(2, "H"), Card(4, "D"),
103                          Card(7, "C"), Card(9, "S"), Card(13, "H")])
103    queen_high = PokerHand([Card(2, "D"), Card(4, "C"),
104                           Card(7, "S"), Card(9, "D"), Card(12, "C")])
```

```
104     actual = king_high.compare_to(queen_high)
105     expected = "positive"
106
107     if actual > 0:
108         actual = "positive"
109
110     assert_equals(mess, expected, actual)
111
112
113 def identical_hands_tie():
114     """Test that same ranks but different suits are a
115     tie"""
116     mess = "Same ranks but different suits"
117     hand1 = PokerHand([Card(2, "H"), Card(5, "D"),
118                        Card(9, "C"), Card(13, "S"), Card(14, "H")])
119     hand2 = PokerHand([Card(2, "D"), Card(5, "C"),
120                        Card(9, "H"), Card(13, "D"), Card(14, "C")])
121     actual = hand1.compare_to(hand2)
122     expected = 0
123
124     assert_equals(mess, expected, actual)
125
126
127 def same_pair_same_other_cards_tie():
128     """Test that same pair with same other cards are a
129     tie"""
130     mess = "Same pair, same other cards"
131     hand1 = PokerHand([Card(10, "H"), Card(10, "D"),
132                        Card(5, "C"), Card(3, "S"), Card(2, "H")])
133     hand2 = PokerHand([Card(10, "C"), Card(10, "S"),
134                        Card(5, "D"), Card(3, "H"), Card(2, "C")])
135     actual = hand1.compare_to(hand2)
136     expected = 0
137
138     assert_equals(mess, expected, actual)
139
140
141 def two_pair_vs_two_pair():
```

```
136     """Test comparing two two pair hands"""
137     mess = "Two pair vs two pair"
138     two_pair1 = PokerHand([Card(9, "H"), Card(9, "D")
139                           ), Card(8, "C"), Card(8, "S"), Card(3, "H")])
139     two_pair2 = PokerHand([Card(9, "C"), Card(9, "S"
140                           ), Card(7, "D"), Card(7, "H"), Card(3, "C")])
140     actual = two_pair1.compare_to(two_pair2)
141     expected = "positive"
142
143     if actual > 0:
144         actual = "positive"
145
146     assert_equals(mess, expected, actual)
147
148
149 def full_house_vs_two_pair():
150     """Test that full house beats two pair"""
151     mess = "Full house vs two pair"
152     full_house = PokerHand([Card(5, "H"), Card(5, "D"
153                           ), Card(5, "C"), Card(7, "S"), Card(7, "H")])
153     two_pair = PokerHand([Card(7, "D"), Card(7, "C"),
154                           Card(6, "S"), Card(6, "H"), Card(3, "D")])
154     actual = full_house.compare_to(two_pair)
155     expected = "negative"
156
157     if actual < 0:
158         actual = "negative"
159
160     assert_equals(mess, expected, actual)
161
162
163 def four_kind_vs_full_house():
164     """Test four of a kind vs full house"""
165     mess = "Four of a kind vs full house"
166     four_kind = PokerHand([Card(13, "H"), Card(13, "D"
167                           ), Card(13, "C"), Card(13, "S"), Card(2, "H")])
167     full_house = PokerHand([Card(14, "H"), Card(14, "D"
168                           ), Card(14, "C"), Card(3, "S"), Card(3, "H")])
```

File - C:\Users\james\Documents\Personal\College Life\Second Year\Courses\CSC\CSC120\Projects\Project2\test\_COMPARETO.py

```
168     actual = four_kind.compare_to(full_house)
169     expected = "negative"
170
171     if actual < 0:
172         actual = "negative"
173
174     assert_equals(mess, expected, actual)
175
176
177 """
178 Individual unit tests end here
179 """
180
181 if __name__ == "__main__":
182     test_COMPARETO()
183
184
```