

```
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(
```

```
38 self.get_rank()))
39         suit_str = SUIT_NAMES.get(self.get_suit(), self
    .get_suit())
40         return f"{rank_str} of {suit_str}"
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     global num_fail
51     print(msg)
52     print("*** FAIL")
53     print(err)
54     print("")
55     num_fail += 1
56
57
58 def start_tests(header):
59     """
60     Initializes summary statistics so we are ready to
61     run tests using
62     assert_equals.
63     :param header: A header to print at the beginning
64     of the tests.
65     """
66     global num_pass, num_fail
67     print(header)
68     for i in range(0, len(header)):
69         print("=", end="")
70     print("")
71     num_pass = 0
72     num_fail = 0
73
74 def finish_tests():
75     """
76     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
6  endeavors with full academic honesty,
7  in accordance with the Union College Honor Code and the
8  course syllabus.
9  """
10
11 from deck import Deck
12
13 from poker_hand import PokerHand
14
15 HAND_SIZE = 5
16
17 def deal_two_hands(deck):
18     """
19     Deal two hands from the deck.
20     :param deck: the deck to deal from
21     :return: two lists of cards
22     """
23     cards1 = []
24     cards2 = []
25     for _ in range(HAND_SIZE):
26         cards1.append(deck.deal())
27     for _ in range(HAND_SIZE):
28         cards2.append(deck.deal())
29     return cards1, cards2
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
35     the winner is (or if there's a tie)
36     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
    left to play another round.
37     ""
38
39     deck = Deck()
40     deck.shuffle()
41     game_count = 1
42     win_rate = 0
43
44
45     while deck.size() >= HAND_SIZE * 2:
46         print(f"Round {game_count}")
47         print("-" * 50)
48
49         cards1, cards2 = deal_two_hands(deck)
50         hand1 = PokerHand(cards1)
51         hand2 = PokerHand(cards2)
52
53         print("Hand 1:")
54         print(hand1)
55         print("Hand 2:")
56         print(hand2)
57
58         player_guess = input("Who wins? (1 for Hand 1,
    2 for Hand 2, 0 for Tie): ")
59
60         comparison = hand1.compare_to(hand2)
61
62         if comparison > 0:
63             actual_winner = 1
64             result_msg = "Hand 1 wins!"
65         elif comparison < 0:
66             actual_winner = 2
67             result_msg = "Hand 2 wins!"
68         else:
69             actual_winner = 0
70             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 # • Two pair (includes two pair, four-of-a-kind, and
  full house)
3 # • Pair (includes pair and three-of-a-kind)
4 # • High card (includes high card and straight). Ace
  has the highest rank and Two has the lowest.
5
6
7 from card import Card
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     def __is_straight(self):
62         """
63         Check if this hand is a straight, consecutive
64         cards.
65         :return: a boolean value indicating if the hand
66         is a straight
67         """
68         ranks = []
69         for card in self.cards:
70             ranks.append(card.get_rank())
71         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         return three and two
122
123     def __is_three_of_a_kind(self):
124         """
125         Check if this hand has exactly three cards of
126         the same rank.
127         :return: a boolean value indicating if the
128         hand has exactly three cards of the same rank
129         """
130         counts = self.__rank_counts().values()
131         if 3 in counts and not self.__is_full_house():
132             return True
133         return False
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
        hand has exactly n pairs
140         """
141         counts = self.__rank_counts()
142         pair_count = 0
143         for value in counts.values():
144             if value == 2:
145                 pair_count += 1
146         return pair_count == n
147
148
149     def __evaluate(self):
150         """
151         Evaluate this hand and return its category.
152         Grouped into: flush, two pair, pair, or high
        card.
153         :return: string representing the hand category
154         """
155         flush = self.__is_flush()
156         straight = self.__is_straight()
157
158         if flush and straight:
159             return "flush"
160         elif flush:
161             return "flush"
162         elif self.__is_four_of_a_kind() or self.
        __is_full_house() or self.__has_pairs(2):
163             return "two pair"
164         elif self.__is_three_of_a_kind() or self.
        __has_pairs(1):
165             return "pair"
166         else:
167             return "high card"
168
169
170     def compare_to(self, other_hand):
171         """

```

```

172         Determines which of two poker hands is worth
        more. Returns an int
173         which is either positive, negative, or zero
        depending on the comparison.
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
        LESS than other_hand,
178         zero if they are worth the SAME (a tie), and a
        positive number if
179         self is worth MORE than other_hand.
180         """
181
182         my_category = self.__evaluate()
183         my_value = CATEGORY_VALUE[my_category]
184
185         other_category = other_hand.__evaluate()
186         other_value = CATEGORY_VALUE[other_category]
187
188
189         category_diff = my_value - other_value
190         if category_diff != 0:
191             return category_diff
192
193         my_counts = self.__rank_counts()
194         other_counts = other_hand.__rank_counts()
195
196         my_list = []
197         for rank in my_counts:
198             my_list.append((my_counts[rank], rank))
199
200         other_list = []
201         for rank in other_counts:
202             other_list.append((other_counts[rank],
        rank))
203
204         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     """
13     print("Testing different categories: ")
14     print('-'*20)
15     flush_vs_high_card()
16     flush_vs_pair()
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
37     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
    tests"""
45     start_tests("Testing compare_to")
46     test_different_categories()
47     test_same_category()
48     test_ties()
49     test_complex_same_category()
50     finish_tests()
51
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
61 (9, "H"), Card(13, "H"), Card(14, "H")])
62     high_card = PokerHand([Card(2, "D"), Card(5, "C"),
63 Card(7, "S"), Card(9, "D"), Card(11, "C")])
64     actual = flush.compare_to(high_card)
65     expected = "positive"
66
67     if actual > 0:
68         actual = "positive"
69
70     assert_equals(mess, expected, actual)
71
72 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     if actual > 0:
82         actual = "positive"
83
84     assert_equals(mess, expected, actual)
85
86 def flush_vs_flush():
87     """Test comparing two flushes"""
88     mess = "FLush vs flush"
89     ace_flush = PokerHand([Card(2, "H"), Card(5, "H"),
90         Card(9, "H"), Card(13, "H"), Card(14, "H")])
91     queen_flush = PokerHand([Card(2, "D"), Card(5, "D"),
92         Card(9, "D"), Card(12, "D"), Card(13, "D")])
93     actual = ace_flush.compare_to(queen_flush)
94     expected = "positive"
95
96     if actual > 0:
97         actual = "positive"
98
99     assert_equals(mess, expected, actual)
100
101 def high_card_vs_high_card():
102     """Test comparing two high card hands"""
103     mess = "High card vs high card"
104     king_high = PokerHand([Card(2, "H"), Card(4, "D"),
105         Card(7, "C"), Card(9, "S"), Card(13, "H")])
106     queen_high = PokerHand([Card(2, "D"), Card(4, "C"),
107         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"
), Card(8, "C"), Card(8, "S"), Card(3, "H")])
139     two_pair2 = PokerHand([Card(9, "C"), Card(9, "S"
), 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"
), Card(5, "C"), Card(7, "S"), Card(7, "H")])
153     two_pair = PokerHand([Card(7, "D"), Card(7, "C"),
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"
), Card(13, "C"), Card(13, "S"), Card(2, "H")])
167     full_house = PokerHand([Card(14, "H"), Card(14, "D"
), Card(14, "C"), Card(3, "S"), Card(3, "H")])

```

```
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_compare_to()
183
184
```