```
"""
Module that has a Card class

:author: Neil Daterao
"""
#constants for cards
RANKS = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
SUITS = ["C", "D", "H", "S"]


class Card:
    """
    Class that creates a card object of a specific rank and suit
    """

    def __init__(self, rank, suit):
        """
        Initialize a card object with a given rank (an integer from 2-14) and a suit (a single character ["H", "D", "S", "C"])
        """

        self.__card = {"rank" : rank, "suit" : suit}


    def get_rank(self):
        """
        Function that returns the rank of the card

        :return: Rank of card
        """

        return self.__card["rank"]

    def get_suit(self):
        """
        Function that returns the rank of the card

```

```
37          :return: Suit of card
38          """
39
40          return self.__card["suit"]
41
42      def __rank_as_face_card(self):
43          """
44          Function that translates the rank into a
    face card if it applies
45
46          :return: Rank as a string of face card
47          """
48          rank = self.get_rank()
49          if rank == 11:
50              new_rank = "Jack"
51          elif rank == 12:
52              new_rank = "Queen"
53          elif rank == 13:
54              new_rank = "King"
55          elif rank == 14:
56              new_rank = "Ace"
57          return new_rank
58
59
60      def __suit_full_name(self):
61          """
62          Function that translates the suit of the
    card into the full name of the suit, ex: "Jack of
    Clubs", "Ace of Diamonds", "3 of Spades" etc...
63
64          :return: Full suit name of the card
65          """
66          suit = self.get_suit()
67          if suit == "H":
68              full_suit = "Hearts"
69          elif suit == "D":
70              full_suit = "Diamonds"
71          elif suit == "S":
72              full_suit = "Spades"
73          elif suit == "C":
74              full_suit = "Clubs"
```

```
75          return full_suit
76
77
78      def __str__(self):
79          """
80          Function to turn the card object into a
    readable string form
81
82          :return: Readable string version of the
    card
83          """
84          rank = self.get_rank()
85          suit = self.__suit_full_name()
86          if rank > 10:
87              rank = self.__rank_as_face_card()
88
89          return str(rank) + " of " + suit
90
91
92  if __name__ == "__main__":
93      #messy tests
94      card = Card(12, "D")
95      print(card)
96
97      print(card.get_rank())
98      print(card.get_suit())
99
```

```
 1 from card import *
 2 from random import shuffle
 3
 4 """
 5 Module that contains a Deck class
 6
 7 :author: Neil Daterao
 8 """
 9
10 class Deck:
11     """
12     Class that creates a deck object
13     """
14
15     def __init__(self):
16         """
17         Constructor that initializes a deck object
18         """
19
20         self.__deck_contents = []
21         for rank in RANKS:
22             for suit in SUITS:
23                 card = Card(rank,suit)
24                 self.__deck_contents.append(card)
25
26
27     def shuffle(self):
28         """
29         Function that shuffles the contents of the
   deck
30
31         :return: None
32         """
33         shuffle(self.__deck_contents)
34
35
36     def deal(self):
37         """
38         Function that deals a card of the top off
   the deck, will return None if deck is empty
39
```

```
40          :return: A card object
41          """
42
43          if self.size == 0:
44              return None
45
46          else:
47              card_to_be_dealt = self.__deck_contents
   [self.size()-1]
48              self.__deck_contents.pop()
49              return card_to_be_dealt
50
51
52      def size(self):
53          """
54          Function to return the current length of
   the deck
55
56          :return: Integer representing the length of
    the deck
57          """
58          return len(self.__deck_contents)
59
60
61      def __str__(self):
62          """
63          Return string version of deck with one card
    per line
64
65          :return: String version of the deck with
   one card per line
66          """
67
68          string_version_of_deck = ""
69          for card in self.__deck_contents:
70              string_version_of_deck += str(card) + "
   \n"
71          return string_version_of_deck
72
73 if __name__ == "__main__":
74      #messy tests
```

```
75      deck = Deck()
76      print(deck)
77      deck.shuffle()
78      print("\n \n \n")
79      print(deck)
80      print(deck.size())
81      print(deck.deal())
82      print(deck.size())
83      print("\n" ,deck)
```

```
"""
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("")
```

```
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 run tests using
61     assert_equals.
62     :param header: A header to print at the
   beginning
63     of the tests.
64     """
65     global num_pass, num_fail
66     print(header)
67     for i in range(0,len(header)):
68         print("=",end="")
69     print("")
70     num_pass = 0
71     num_fail = 0
72
73 def finish_tests():
74     """
75     Prints summary statistics after the tests are
   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 from card import *
 2 from deck import *
 3 from poker_hand import *
 4 """
 5 Simple game where the user chooses which poker hand
    they think is stronger.
 6
 7 :author: Neil Daterao
 8 :note: I affirm that I have carried out the
   attached academic endeavors with full academic
   honesty, in
 9 accordance with the Union College Honor Code and
   the course syllabus.
10 """
11
12 def poker_simulation_game():
13     """
14     Simple game where the user chooses which poker
   hand they think is stronger.
15     """
16
17     start_message = input("Welcome to the poker
   strength test game! Press return to get your first
   set of hands!")
18     deck_of_cards = Deck()
19     deck_of_cards.shuffle()
20     total_points = 0
21
22     while deck_of_cards.size() >= (
   max_cards_in_hand * 2):
23         poker_hand = []
24         other_poker_hand = []
25         for dealing_card_count in range(
   max_cards_in_hand):
26             poker_hand.append(deck_of_cards.deal())
27             other_poker_hand.append(deck_of_cards.
   deal())
28         poker_hand = PokerHand(poker_hand)
29         other_poker_hand = PokerHand(
   other_poker_hand)
30         expected_winner = poker_hand.compare_to(
```

```
30 other_poker_hand)
31
32         print("    Hand #1 \n")
33         print("---------------")
34         print(poker_hand)
35         print("---------------")
36         print("    Hand #2 \n")
37         print("---------------")
38         print(other_poker_hand)
39         print("---------------")
40         #print("Cheat:", expected_winner) -> If you
    want to cheat
41
42         answer = input("Which hand do you think is
   stronger, 1 or 2? If you think it's a tie enter 0!
   \n")
43
44         accepted_answers = ["0", "1", "2"]
45         while answer not in accepted_answers:
46             print("Invalid Input! \n")
47             answer = input("Which hand do you think
    is stronger, 1 or 2? If you think it's a tie enter
    0! \n")
48
49         if int(answer) == 1 and expected_winner > 0
   :
50             total_points += 1
51             print("\nGood job you got it right! Let
   's see if you can get the next set of hands right
   too! \n")
52
53         elif int(answer) == 2 and expected_winner
    < 0:
54             total_points += 1
55             print("\nGood job you got it right! Let
   's see if you can get the next set of hands right
   too! \n")
56
57         elif int(answer) == 0 and expected_winner
    == 0:
58             total_points += 1
```

```
59              print("\nGood job you got it right! Let
   's see if you can get the next set of hands right
   too! \n")
60
61          else:
62              print("Incorrect! Game Over!")
63              print("Total Points: ", total_points)
64              return 0
65      print("Game Over! You got the whole deck right
   , good job!")
66      print("Total Points: ", total_points)
67
68      return 0
69
70
71  def main():
72      play_again = "Y"
73      while play_again == "Y":
74          poker_simulation_game()
75          play_again = input("Would you like to play
   again (Y/N) ")
76          if play_again == "N":
77              print("Awh Bummer! Have a nice day!")
78              return 0
79
80  if __name__ == "__main__":
81      main()
```

```
 1 from card import *
 2 from deck import *
 3 from copy import deepcopy
 4 from collections import Counter
 5 """
 6 Module containing a PokerHand class
 7
 8 :author: Neil Daterao
 9 """
10
11 max_cards_in_hand = 5
12
13 class PokerHand:
14     """
15     Class that creates a poker hand object
16     """
17
18     def __init__(self,list_of_cards):
19         """
20         Constructor initializes a copy of a deck of
    cards and an empty hand
21         """
22         self.__list_of_cards = deepcopy(
   list_of_cards) #Creates a copy of the list of card
   objects
23         self.__hand = []
24         for count in range(max_cards_in_hand):
25             self.__hand.append(self.__list_of_cards
   [count])
26
27
28     def add_card(self, card):
29         """
30         Function that adds card object passed in as
    a parameter to the hand. Will throw an error if
   hand is already full
31         """
32         #make so it can't add 6 cards
33         if self.size() < 5:
34             self.__hand.append(card)
35         else:
```

```
36              print("Error: Hand is full")
37
38
39      def get_ith_card(self, index):
40          """
41          Return a card from the hand at the given
    index
42
43          :return: a card object
44          """
45          if index < 0:
46              print("Error: Invalid Index, must be
     >= 0")
47              return None
48
49          return self.__hand[index]
50
51
52      def size(self):
53          """
54          Get the current size of the hand
55
56          :return: Integer representing the size of
    the hand
57          """
58          return len(self.__hand)
59
60
61      def __str__(self):
62          """
63          Return string version of hand with one card
     per line
64
65          :return: String version of the hand with
    one card per line
66          """
67
68          string_version_of_hand = ""
69          for card in self.__hand:
70              string_version_of_hand += str(card) + "
    \n"
```

```
 71         return string_version_of_hand
 72
 73
 74     def __is_flush(self):
 75         """
 76         Determines if hand is a flush
 77
 78         :return: Boolean, True if a flush and
    False if not
 79         """
 80         first_card = self.get_ith_card(0)
 81         previous = first_card.get_suit()
 82         for card in self.__hand:
 83             if card.get_suit() != previous:
 84                 return False
 85             previous = card.get_suit()
 86         return True
 87
 88
 89     def __num_of_pairs(self):
 90         """
 91         Determines number of pairs in a hand
 92
 93         :return: An integer representing the
    number of pairs
 94         """
 95
 96         rank_of_cards = []
 97         for card in self.__hand:
 98             rank_of_cards.append(card.get_rank())
 99
100         count_of_ranks = Counter(rank_of_cards)
101
102         pairs = 0
103         for rank in count_of_ranks:
104             if count_of_ranks[rank] ==  4:
105                 pairs += 2
106             elif count_of_ranks[rank] >= 2:
107                 pairs += 1
108             else:
109                 pairs += 0
```

```
110         return pairs
111 
112 
113     def __determine_classification_of_hand(self):
114         """
115         Determines the classification of a hand.
    In other words, if the hand is a pair, 2 pair,
    flush, or high card.
116 
117         :return: A string of the classification of
     the hand
118 
119         """
120         if self.__is_flush():
121             return "Flush"
122         elif self.__num_of_pairs() == 2:
123             return "Two pair"
124         elif self.__num_of_pairs() == 1:
125             return "Pair"
126         else:
127             return "High card"
128 
129 
130     def
    __translate_classification_of_hand_to_power_rankin
    g(self):
131         """
132         Function to translate the hand type into a
     number, 4 (strongest, Flush) to 1 (Weakest, High
    Card)
133 
134         :return: Integer of power ranking
135         """
136         classification_of_hand = self.
    __determine_classification_of_hand()
137         if classification_of_hand == "Flush":
138             power_ranking = 4
139         elif classification_of_hand == "Two pair":
140             power_ranking = 3
141         elif classification_of_hand == "Pair":
142             power_ranking = 2
```

```
143         else:
144             power_ranking = 1
145         return power_ranking
146 
147 
148     def __get_ranks_of_hand_in_list(self):
149         """
150         Translates the ranks of the cards in the hand into a list
151 
152         :return: A list of the ranks of the cards in the hands
153         """
154 
155         ranks_of_hand = []
156         for index in range(max_cards_in_hand):
157             ranks_of_hand.append(self.get_ith_card(index).get_rank())
158         return ranks_of_hand
159 
160 
161     def __determine_winning_flush(self, other_hand):
162         """
163         Function to determine a winning flush given two hands that are flushes. Returns 1 if the "self" hand wins, -1 if the other hand wins and 0 if the hands tie.
164 
165         :param other_hand: A PokerHand object which is a flush
166         :return: 1 if the "self" hand wins, -1 if the other hand wins and 0 if the hands tie.
167         """
168         #Since we alreay know this is a flush, we have to check the ranks of the card and we can use the same algorithm we used to check high cards.
169 
170         return self.__determine_winning_high_card(other_hand)
171 
```

```
172
173     def __determine_winning_two_pair(self,
    other_hand):
174         """
175         Function to determine which hand of two
    pairs is stronger
176
177         :param other_hand: Another PokerHand
    Object which is classified as a two pair
178         :return: Will return 1 if the "self"
    PokerHand is stronger, -1 if the other_hand
    PokerHand is stronger and 0 if they are of equal
    strength
179         """
180         pairs_in_self = []
181         extra_card_in_self = None
182         pairs_in_other_hand = []
183         extra_card_in_other_hand = None
184         max_index = 2 #Magic number but since the
    function is private it's okay! The reason for this
     magic number is to account for hands where we are
     comparing 4 of a kind to a 2 regular 2 pair.
185
186         ranks_in_self = self.
    __get_ranks_of_hand_in_list()
187         ranks_in_other_hand = other_hand.
    __get_ranks_of_hand_in_list()
188
189         counts_of_ranks_in_self = Counter(
    ranks_in_self)
190         counts_of_ranks_in_other_hand = Counter(
    ranks_in_other_hand)
191
192         for card in counts_of_ranks_in_self:
193             if counts_of_ranks_in_self[card] >= 2:
194                 pairs_in_self.append(card)
195             else:
196                 extra_card_in_self = card
197
198         for card in counts_of_ranks_in_other_hand:
199             if counts_of_ranks_in_other_hand[card
```

```
199 ] >= 2:
200                 pairs_in_other_hand.append(card)
201             else:
202                 extra_card_in_other_hand = card
203 
204         pairs_in_self.sort(reverse=True)
205         pairs_in_other_hand.sort(reverse=True)
206 
207         if len(pairs_in_self) != len(
    pairs_in_other_hand):
208             max_index = 1
209 
210         for card_count in range(0,max_index):
211             if pairs_in_self[card_count] >
    pairs_in_other_hand[card_count]:
212                 return 1
213             elif pairs_in_self[card_count] <
    pairs_in_other_hand[card_count]:
214                 return -1
215 
216         return extra_card_in_self -
    extra_card_in_other_hand
217 
218 
219     def __determine_winning_one_pair(self,
    other_hand):
220         """
221         Function to determine which hand of pairs
    is stronger
222 
223         :param other_hand: Another PokerHand
    Object which is classified as a Pair
224         :return: Will return 1 if the "self"
    PokerHand is stronger, -1 if the other_hand
    PokerHand is stronger and 0 if they are of equal
    strength
225         """
226         pair_in_self = None
227         extra_cards_in_self = []
228         pair_in_other_hand = None
229         extra_cards_in_other_hand = []
```

```
230         max_index = 3 #Magic number again, but it'
    s okay since it is a private function, this is to
    account for comparing three of a kind with one
    pair, the list of extra cards will be different
    lengths and we want to iterate through the
    smallest sorted array.
231 
232         ranks_in_self = self.
    __get_ranks_of_hand_in_list()
233         ranks_in_other_hand = other_hand.
    __get_ranks_of_hand_in_list()
234 
235         for card_rank in ranks_in_self:
236             if card_rank in extra_cards_in_self:
237                 pair_in_self = card_rank
238             else:
239                 extra_cards_in_self.append(
    card_rank)
240 
241         for card_rank in ranks_in_other_hand:
242             if card_rank in
    extra_cards_in_other_hand:
243                 pair_in_other_hand = card_rank
244             else:
245                 extra_cards_in_other_hand.append(
    card_rank)
246 
247         extra_cards_in_self.sort(reverse=True)
248         extra_cards_in_other_hand.sort(reverse=
    True)
249 
250         if len(extra_cards_in_self) != len(
    extra_cards_in_other_hand):
251             max_index = 2
252 
253         if pair_in_self == pair_in_other_hand:
254             for card_count in range(max_index):
255                 if extra_cards_in_self[card_count
    ] > extra_cards_in_other_hand[card_count]:
256                     return 1
257                 elif extra_cards_in_self[
```

```
257 card_count] < extra_cards_in_other_hand[card_count
    ]:
258                     return -1
259 
260         return pair_in_self - pair_in_other_hand
261 
262 
263     def __determine_winning_high_card(self,
    other_hand):
264         """
265         Function to determine which hand of high
    cards is stronger
266 
267         :param other_hand: Another PokerHand
    Object which is classified as a High Card
268         :return: Will return 1 if the "self"
    PokerHand is stronger, -1 if the other_hand
    PokerHand is stronger and 0 if they are of equal
    strength
269         """
270 
271         ranks_of_self_hand = self.
    __get_ranks_of_hand_in_list()
272         ranks_of_other_hand = other_hand.
    __get_ranks_of_hand_in_list()
273 
274         ranks_of_self_hand.sort(reverse= True)
275         ranks_of_other_hand.sort(reverse= True)
276 
277         card_count = 0
278         while card_count < max_cards_in_hand:
279             if ranks_of_self_hand[card_count] ==
    ranks_of_other_hand[card_count]:
280                 card_count += 1
281             elif ranks_of_self_hand[card_count] >
    ranks_of_other_hand[card_count]:
282                 return 1
283             elif ranks_of_self_hand[card_count] <
    ranks_of_other_hand[card_count]:
284                 return -1
285         return 0
```

```
286
287
288     def compare_to(self, other_hand):
289         """
290         Determines which of two poker hands is
    worth more. Returns an int
291         which is either positive, negative, or
    zero depending on the comparison.
292
293         :param self: The first hand to compare
294         :param other_hand: The second hand to
    compare
295         :return: a negative number if self is
    worth LESS than other_hand,
296         zero if they are worth the SAME (a tie),
    and a positive number if
297         self is worth MORE than other_hand
298         """
299         classification_of_self_hand = self.
    __determine_classification_of_hand()
300         classification_of_other_hand = other_hand.
    __determine_classification_of_hand()
301         power_ranking_of_hand = self.
    __translate_classification_of_hand_to_power_rankin
    g()
302         power_ranking_of_other_hand = other_hand.
    __translate_classification_of_hand_to_power_rankin
    g()
303
304         if power_ranking_of_hand -
    power_ranking_of_other_hand == 0:
305             if classification_of_self_hand == "
    Flush":
306                 return self.
    __determine_winning_flush(other_hand)
307             elif classification_of_self_hand == "
    Two pair":
308                 return self.
    __determine_winning_two_pair(other_hand)
309             elif classification_of_self_hand == "
    Pair":
```

```
310                 return self.
    __determine_winning_one_pair(other_hand)
311             else:
312                 return self.
    __determine_winning_high_card(other_hand)
313 
314         else:
315             return power_ranking_of_hand -
    power_ranking_of_other_hand
316 
317 
318 if __name__ == "__main__":
319     #messy tests
320     hand = PokerHand([Card(13, "D"), Card(10,"D"
    ), Card(5,"D"), Card(4,"D"), Card(2, "D")])
321     hand1 = PokerHand([Card(8, "C"), Card(8,"D"),
    Card(3,"H"), Card(3,"S"), Card(6, "D")])
322     hand3 = PokerHand([Card(8, "C"), Card(8,"D"),
    Card(3,"H"), Card(3,"S"), Card(7, "D")])
323 
324     hand4 = PokerHand([Card(4, "D"), Card(4, "C"
    ), Card(4, "S"), Card(4,"D"), Card(12, "S")])
325     hand5 = PokerHand([Card(7, "D"), Card(7, "C"
    ), Card(7, "S"), Card(6,"D"), Card(6, "S")])
326 
327     hand2 = PokerHand([Card(13, "D"), Card(10,"D"
    ), Card(7,"D"), Card(4,"D"), Card(2, "D")])
328     print(hand.compare_to(hand1))
329     print(hand1.compare_to(hand3))
330     print(hand4.compare_to(hand5))
331 
```

```
 1 from card import *
 2 from testing import *
 3
 4 """
 5 Test cases for the card class
 6
 7 :author: Neil Daterao
 8 """
 9
10 def test_card_as_string():
11     """
12     Test the card class such that it correctly
   returns the string version of a card object
13     """
14     start_tests("Testing the string version of the
   card class")
15     __test_card_as_string_for_clubs_suit()
16     __test_card_as_string_for_diamonds_suit()
17     __test_card_as_string_for_hearts_suit
18     __test_card_as_string_for_spades_suit()
19     __test_card_as_string_for_nonface_card()
20     __test_card_as_string_for_jack_rank()
21     __test_card_as_string_for_queen_rank()
22     __test_card_as_string_for_king_rank()
23     __test_card_as_string_for_ace_rank()
24     finish_tests()
25
26
27 def __test_card_as_string_for_clubs_suit():
28     """
29     Test a card that has a clubs suit, i.e "C"
30     """
31     rank = 14
32     suit = "C"
33     card = Card(rank, suit)
34     actual = str(card)
35     expected = "Ace of Clubs"
36     assert_equals("String of clubs suited card",
   expected, actual)
37
38 def __test_card_as_string_for_nonface_card():
```

```
39      """
40      Test a card thats a non face card
41      """
42      rank = 10
43      suit = "H"
44      card = Card(rank, suit)
45      actual = str(card)
46      expected = "10 of Hearts"
47      assert_equals("String of non-face card",
    expected, actual)
48
49  def __test_card_as_string_for_diamonds_suit():
50      """
51      Test a card that has a diamond as a suit, i.e "
    D"
52      """
53      rank = 3
54      suit = "D"
55      card = Card(rank, suit)
56      actual = str(card)
57      expected = "3 of Diamonds"
58      assert_equals("String of diamond suited card",
    expected, actual)
59
60  def __test_card_as_string_for_spades_suit():
61      """
62      Test a card that has a spades as a suit, i.e "S
    "
63      """
64      rank = 8
65      suit = "S"
66      card = Card(rank, suit)
67      actual = str(card)
68      expected = "8 of Spades"
69      assert_equals("String of spade suited card",
    expected, actual)
70
71  def __test_card_as_string_for_hearts_suit():
72      """
73      Test a card that has hearts as a suit, i.e "H"
74      """
```

```
 75     rank = 9
 76     suit = "H"
 77     card = Card(rank, suit)
 78     actual = str(card)
 79     expected = "9 of Hearts"
 80     assert_equals("String of hearts suited card",
    expected, actual)
 81
 82 def __test_card_as_string_for_jack_rank():
 83     """
 84     Test a card that has jack as a rank, i.e "11"
 85     """
 86     rank = 11
 87     suit = "C"
 88     card = Card(rank, suit)
 89     actual = str(card)
 90     expected = "Jack of Clubs"
 91     assert_equals("String of jack ranked card",
    expected, actual)
 92
 93
 94 def __test_card_as_string_for_queen_rank():
 95     """
 96     Test a card that has a queen as a rank, i.e "
    12"
 97     """
 98     rank = 12
 99     suit = "H"
100     card = Card(rank, suit)
101     actual = str(card)
102     expected = "Queen of Hearts"
103     assert_equals("String of queen ranked card",
    expected, actual)
104
105 def __test_card_as_string_for_king_rank():
106     """
107     Test a card that has a king as a rank, i.e "13
    "
108     """
109     rank = 13
110     suit = "D"
```

```
111     card = Card(rank, suit)
112     actual = str(card)
113     expected = "King of Diamonds"
114     assert_equals("String of king ranked card",
    expected, actual)
115 
116 def __test_card_as_string_for_ace_rank():
117     """
118     Test a card that has an ace as a rank, i.e "14
    "
119     """
120     rank = 14
121     suit = "D"
122     card = Card(rank, suit)
123     actual = str(card)
124     expected = "Ace of Diamonds"
125     assert_equals("String of ace ranked card",
    expected, actual)
126 
127 test_card_as_string()
```

```
 1 from cmath import exp
 2 from poker_hand import *
 3 from testing import *
 4
 5 """
 6 Test cases for the PokerHand class, specifically
   the compare_to function
 7
 8 :author: Neil Daterao
 9 """
10
11
12 def test_compare_to():
13     """
14     Test the compare_to function in the PokerHand
   class such that it successfully compares various
   hands
15     """
16     start_tests("Testing compare_to function")
17     __test_one_pair_aces_v_fours()
18     __test_two_pair_94_v_93()
19     __test_two_pair_ace_v_ace_high_card_win()
20
   __test_two_pair_ace_v_ace_high_card_tie_second_win
   ()
21     __test_two_pair_58_v_76()
22     __test_two_pair_of_same_pairs_diff_high_card()
23     __test_9_high_flush_v_6_high_flush()
24     __test_king_10_7_flush_v_king_10_5_flush()
25     __test_flush_actual_tie()
26     __test_high_card_tie()
27     __test_two_pair_tie()
28     __test_one_pair_tie()
29     __test_flush_vs_high_card()
30     __test_two_pair_vs_one_pair()
31     __test_full_house_vs_straight()
32     __test_royal_flush_vs_straight_flush()
33     __test_four_of_a_kind_vs_two_pair()
34     __test_high_card_tie_up_to_third_card()
35     __test_one_pair_tie_up_to_fourth_card()
36     __test_three_of_a_kind_vs_two_pair()
```

```
37      finish_tests()
38
39
40
41  def __test_one_pair_aces_v_fours():
42      """
43      Testing hands of 1 pairs, aces v fours
44      """
45      poker_hand = PokerHand([Card(14, "S"), Card(14,
    "C"), Card(5,"H"), Card(4,"D"), Card(2, "D")])
46      other_poker_hand = PokerHand([Card(4, "S"),
    Card(4,"C"), Card(5,"H"), Card(3,"D"), Card(2, "D"
    )])
47      actual = poker_hand.compare_to(other_poker_hand
    )
48      expected = 10
49      assert_equals("Testing hands of 1 pairs, aces v
     fours.", expected, actual)
50
51  def __test_two_pair_94_v_93():
52      """
53      Testing hands of 2 pairs, 9 and 4 v 9 and 3
54      """
55      poker_hand = PokerHand([Card(9, "S"), Card(9,"C
    "), Card(4,"H"), Card(4,"D"), Card(2, "D")])
56      other_poker_hand = PokerHand([Card(9, "S"),
    Card(9,"C"), Card(3,"H"), Card(3,"D"), Card(2, "D"
    )])
57      actual = poker_hand.compare_to(other_poker_hand
    )
58      expected = 1
59      assert_equals("Testing hands of 2 pairs, 9 and
    4 v 9 and 3 .", expected, actual)
60
61  def __test_two_pair_ace_v_ace_high_card_win():
62      """
63      Testing hands of 1 pairs where the aces tie but
     the high card wins
64      """
65      poker_hand = PokerHand([Card(14, "S"), Card(14,
    "C"), Card(7,"H"), Card(2,"D"), Card(9, "D")])
```

```
66      other_poker_hand = PokerHand([Card(14, "S"),
    Card(14,"C"), Card(8,"H"), Card(4,"D"), Card(2, "D
    ")])
67      actual = poker_hand.compare_to(
    other_poker_hand)
68      expected = 1
69      assert_equals("Testing hands of 1 pairs where
    the aces tie but the high card wins ", expected,
    actual)
70
71  def
    __test_two_pair_ace_v_ace_high_card_tie_second_win
    ():
72      """
73      Testing hands of 1 pairs where the aces tie
    and the high card ties, but the second high card
    wins
74      """
75      poker_hand = PokerHand([Card(14, "S"), Card(14
    ,"C"), Card(7,"H"), Card(2,"D"), Card(9, "D")])
76      other_poker_hand = PokerHand([Card(14, "S"),
    Card(14,"C"), Card(9,"H"), Card(4,"D"), Card(2, "D
    ")])
77      actual = poker_hand.compare_to(
    other_poker_hand)
78      expected = 1
79      assert_equals("Testing hands of 1 pairs where
    the aces tie and the high card ties, but the
    second high card wins ", expected, actual)
80
81  def __test_two_pair_58_v_76():
82      """
83      Testing hands of 2 pairs, 5 and 8 v 7 and 6
84      """
85      poker_hand = PokerHand([Card(5, "S"), Card(5,"
    C"), Card(8,"H"), Card(8,"D"), Card(9, "D")])
86      other_poker_hand = PokerHand([Card(7, "S"),
    Card(7,"C"), Card(6,"H"), Card(6,"D"), Card(2, "D"
    )])
87      actual = poker_hand.compare_to(
    other_poker_hand)
```

```
 88     expected = 1
 89     assert_equals("Testing hands of 2 pairs, 5 and
     8 v 7 and 6 ", expected, actual)
 90
 91 def __test_two_pair_of_same_pairs_diff_high_card
    ():
 92     """
 93     Testing hands of 2 pairs, of the same pairs
    but the high card wins it
 94     """
 95     poker_hand = PokerHand([Card(10, "S"), Card(10
    ,"C"), Card(2,"H"), Card(2,"D"), Card(13, "D")])
 96     other_poker_hand = PokerHand([Card(10, "S"),
    Card(10,"C"), Card(2,"H"), Card(2,"D"), Card(4, "D
    ")])
 97     actual = poker_hand.compare_to(
    other_poker_hand)
 98     expected = 9
 99     assert_equals(" Testing hands of 2 pairs, of
    the same pairs but the high card wins it ",
    expected, actual)
100
101 def __test_9_high_flush_v_6_high_flush():
102     """
103     Testing hands of flushes, 9 high v 6 high
104     """
105     poker_hand = PokerHand([Card(8, "S"), Card(9,"
    S"), Card(3,"S"), Card(2,"S"), Card(7, "S")])
106     other_poker_hand = PokerHand([Card(6, "C"),
    Card(2,"C"), Card(3,"C"), Card(5,"C"), Card(4, "C"
    )])
107     actual = poker_hand.compare_to(
    other_poker_hand)
108     expected = 1
109     assert_equals(" Testing hands of flushes, 9
    high v 6 high ", expected, actual)
110
111 def __test_king_10_7_flush_v_king_10_5_flush():
112     """
113     Testing hands of flushes, King-10-7 v King-10-
    5
```

```
114     """
115     poker_hand = PokerHand([Card(13, "S"), Card(10
    ,"S"), Card(7,"S"), Card(2,"S"), Card(6, "S")])
116     other_poker_hand = PokerHand([Card(13, "C"),
    Card(10,"C"), Card(5,"C"), Card(2,"C"), Card(4, "C
    ")])
117     actual = poker_hand.compare_to(
    other_poker_hand)
118     expected = 1
119     assert_equals(" Testing hands of flushes, King
    -10-7 v King-10-5 ", expected, actual)
120
121 def __test_flush_actual_tie():
122     """
123     Test a tie between two poker hands with flush
    as the winning criteria.
124     """
125     poker_hand = PokerHand([Card(13, "S"), Card(10
    ,"S"), Card(7,"S"), Card(2,"S"), Card(6, "S")])
126     other_poker_hand = PokerHand([Card(13, "C"),
    Card(10,"C"), Card(7,"C"), Card(2,"C"), Card(6, "C
    ")])
127     actual = poker_hand.compare_to(
    other_poker_hand)
128     expected = 0
129     assert_equals(" Testing flush actual tie ",
    expected, actual)
130
131 def __test_high_card_tie():
132     """
133     Test a tie between two poker hands with high
    card as the winning criteria.
134     """
135     poker_hand = PokerHand([Card(13, "D"), Card(10
    ,"H"), Card(7,"S"), Card(2,"C"), Card(6, "S")])
136     other_poker_hand = PokerHand([Card(13, "S"),
    Card(10,"C"), Card(7,"D"), Card(2,"C"), Card(6, "D
    ")])
137     actual = poker_hand.compare_to(
    other_poker_hand)
138     expected = 0
```

```
139     assert_equals("Testing high card tie",
    expected, actual)
140
141 def __test_two_pair_tie():
142     """
143     Test a tie between two poker hands with two
    pair as the winning criteria.
144     """
145     poker_hand = PokerHand([Card(13, "S"), Card(13
    ,"C"), Card(7,"S"), Card(7,"C"), Card(2, "S")])
146     other_poker_hand = PokerHand([Card(13, "D"),
    Card(13,"H"), Card(7,"D"), Card(7,"H"), Card(2, "D
    ")])
147     actual = poker_hand.compare_to(
    other_poker_hand)
148     expected = 0
149     assert_equals("Testing two-pair tie", expected
    , actual)
150
151 def __test_one_pair_tie():
152     """
153     Test a tie between two poker hands with one
    pair as the winning criteria.
154     """
155     poker_hand = PokerHand([Card(13, "S"), Card(13
    ,"C"), Card(7,"S"), Card(2,"D"), Card(6, "S")])
156     other_poker_hand = PokerHand([Card(13, "D"),
    Card(13,"H"), Card(7,"D"), Card(2,"C"), Card(6, "D
    ")])
157     actual = poker_hand.compare_to(
    other_poker_hand)
158     expected = 0
159     assert_equals("Testing one-pair tie", expected
    , actual)
160
161 def __test_flush_vs_high_card():
162     """
163     Test a comparison between a flush and a high
    card.
164     """
165     poker_hand = PokerHand([Card(13, "S"), Card(10
```

```
165 , "S"), Card(7, "S"), Card(2, "S"), Card(6, "S")])
166     other_poker_hand = PokerHand([Card(13, "D"),
    Card(11, "S"), Card(8, "H"), Card(5, "C"), Card(2
    , "D")])
167     actual = poker_hand.compare_to(
    other_poker_hand)
168     expected = 3
169     assert_equals("Testing flush vs high card",
    expected, actual)
170
171 def __test_two_pair_vs_one_pair():
172     """
173     Test a comparison between two poker hands, one
     with two pairs and the other with one pair.
174     """
175     poker_hand = PokerHand([Card(13, "S"), Card(13
    , "C"), Card(7, "S"), Card(7, "C"), Card(2, "S")])
176     other_poker_hand= PokerHand([Card(13, "D"),
    Card(13, "H"), Card(9, "D"), Card(6, "C"), Card(4
    , "D")])
177     actual = poker_hand.compare_to(
    other_poker_hand)
178     expected = 1
179     assert_equals("Testing two pair vs one pair",
    expected, actual)
180
181 def __test_full_house_vs_straight():
182     """
183     Test a comparison between a full house and a
    straight.
184     """
185     poker_hand = PokerHand([Card(13, "S"), Card(13
    , "C"), Card(13, "D"), Card(7, "S"), Card(7, "C"
    )])
186     other_poker_hand = PokerHand([Card(10, "S"),
    Card(9, "D"), Card(8, "C"), Card(7, "S"), Card(6,
    "H")])
187     actual = poker_hand.compare_to(
    other_poker_hand)
188     expected = 2
189     assert_equals("Testing full house vs straight"
```

```
189 , expected, actual)
190
191 def __test_royal_flush_vs_straight_flush():
192     """
193     Test a comparison between a royal flush and a
    straight flush.
194     """
195     poker_hand = PokerHand([Card(10, "S"), Card(11
    , "S"), Card(12, "S"), Card(13, "S"), Card(14, "S"
    )])
196     other_poker_hand = PokerHand([Card(9, "H"),
    Card(8, "H"), Card(7, "H"), Card(6, "H"), Card(5,
    "H")])
197     actual = poker_hand.compare_to(
    other_poker_hand)
198     expected = 1
199     assert_equals("Testing royal flush vs straight
     flush", expected, actual)
200
201 def __test_four_of_a_kind_vs_two_pair():
202     """
203     Test a comparison between a four of a kind and
     a two pair.
204     """
205     poker_hand = PokerHand([Card(13, "S"), Card(13
    , "C"), Card(13, "D"), Card(13, "H"), Card(7, "S"
    )])
206     other_poker_hand = PokerHand([Card(10, "H"),
    Card(10, "S"), Card(9, "D"), Card(9, "C"), Card(4
    , "H")])
207     actual = poker_hand.compare_to(
    other_poker_hand)
208     expected = 1
209     assert_equals("Testing four of a kind vs two
    pair", expected, actual)
210
211 def __test_high_card_tie_up_to_third_card():
212     """
213     Test a comparison between two high card hands
    that tie up until the third card.
214     """
```

```
215     poker_hand = PokerHand([Card(13, "S"), Card(10
    , "C"), Card(8, "H"), Card(5, "D"), Card(2, "S")])
216     other_poker_hand = PokerHand([Card(13, "C"),
    Card(10, "D"), Card(8, "S"), Card(6, "H"), Card(3
    , "C")])
217     actual = poker_hand.compare_to(
    other_poker_hand)
218     expected = -1
219     assert_equals("Testing high card tie up to
    third card", expected, actual)
220
221 def __test_one_pair_tie_up_to_fourth_card():
222     """
223     Test a comparison between two one pair hands
    that tie up until the fourth card.
224     """
225     poker_hand = PokerHand([Card(10, "S"), Card(10
    , "C"), Card(9, "H"), Card(7, "D"), Card(2, "S")])
226     other_poker_hand = PokerHand([Card(10, "D"),
    Card(10, "H"), Card(9, "S"), Card(6, "H"), Card(3
    , "C")])
227     actual = poker_hand.compare_to(
    other_poker_hand)
228     expected = 1
229     assert_equals("Testing one pair tie up to
    fourth card", expected, actual)
230
231 def __test_three_of_a_kind_vs_two_pair():
232     """
233     Test a comparison between a three of a kind
    hand and a two pair hand.
234     """
235     poker_hand = PokerHand([Card(9, "S"), Card(9,
    "C"), Card(9, "D"), Card(5, "H"), Card(2, "S")])
236     other_poker_hand = PokerHand([Card(10, "H"),
    Card(10, "S"), Card(9, "D"), Card(9, "C"), Card(4
    , "H")])
237     actual = poker_hand.compare_to(
    other_poker_hand)
238     expected = -1
239     assert_equals("Testing three of a kind vs two
```

```
239 pair", expected, actual)
240
241
242 test_compare_to()
243
244
```