```
"""
The Card module. Contains methods of the Card class related to individual cards.
"""


class Card:
    """
    Creates a card object.
    """

    def __init__(self, rank, suit):
        """
        Creates a card.
        Card is dictionary format, with key and value for rank and suit.
        :param rank: The rank of the card. Can be 2-14.
        :param suit: The suit of the card. Can be S, H, D, or C.
        """
        self.__card = {'rank': rank, 'suit': suit}

    def find_suit(self):
        """
        Returns a string of the suit. (clubs instead of 'C'.)
        :return: One of the four suits, in lowercase specifically.
        """

        value = self.__card['suit']

        if value == 'S':
            return 'spades'
        if value == 'H':
            return 'hearts'
        if value == 'D':
            return 'diamonds'
        if value == 'C':
            return 'clubs'
```

```
36
37      def find_rank(self):
38          """
39          Returns the cards rank, as a str, depending
    on rank.
40          :return: The rank of the card. 11-14 are the
    jack, queen, king, and ace.
41          """
42          value = self.__card['rank']
43
44          if value >= 2 and value <= 10:
45              return str(value)
46          elif value == 11:
47              return 'jack'
48          elif value == 12:
49              return 'queen'
50          elif value == 13:
51              return 'king'
52          elif value == 14:
53              return 'ace'
54
55      def __str__(self):
56          """
57          Returns a plainly stated card, in the form "
    The (rank) of (suit)"
58          :return: The card, in plain english.
    Specifically a string.
59          """
60
61          return str(f'The {self.find_rank()} of {self.
    find_suit()}')
62
```

```
 1 """
 2 The Deck module. Contains methods of the Deck class
   related to a deck of cards.
 3 """
 4 import random
 5
 6 from card import *
 7
 8 SUITS = ['H', 'D', 'S', 'C']
 9 RANKS = [
10         2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
11     ]
12 STRING_RANKS = [
13     str(2), str(3), str(4), str(5), str(6), str(7),
   str(8),
14     str(9), str(10), 'jack', 'queen', 'king', 'ace'
15 ]
16
17
18 class Deck:
19     """
20     Models a deck of cards. Will hold no more than 52
    elements.
21     """
22
23     def __init__(self):
24         """
25         Creates the deck.
26         Specifically, a list of 52 dictionaries.
27         """
28         self.__deck = []
29         for rank in RANKS:
30             for suit in SUITS:
31                 self.__deck.append(Card(rank, suit))
32
33     def shuffle(self):
34         """
35         Shuffles the deck in random order.
36         """
37         random.shuffle(self.__deck)
38
```

```
39      def deal_one(self):
40          """
41          As long as there are cards remaining in the
    deck, a card is drawn.
42          :return: card: If there is at least one card
    in the deck, this
43          card is returned.
44          :return: None: If the deck is empty, with no
    more cards remaining,
45          returns keyword None.
46          """
47          if Deck.remaining_cards_in_deck(self) != 0:
48              card = self.__deck[0]
49              self.__deck.pop(0)
50              return card
51          else:
52              return None
53
54      def remaining_cards_in_deck(self):
55          """
56          Returns the amount of cards remaining in the
    deck.
57          Used in Deck function deal_one.
58          :return: A number representing the amount of
    cards in the deck.
59          """
60          return len(self.__deck)
61
62      def __str__(self):
63          """
64          Prints the entire deck, line by line.
65          :return: Returns a variable 'to_return' that
    is continuously updated
66          with a new card from the deck throughout the
    for loop.
67          Starts with header 'DECK CONTENTS' and goes
    through a loop of the deck.
68          When the end of the loop is reached, the
    resulting print is the header
69          above every card from the deck, each placed
    neatly on a line.
```

```
70          """
71          to_return = "DECK CONTENTS:\n"
72          for index in range(0, self.
    remaining_cards_in_deck()):
73              card = self.__deck[index]
74              to_return += str(card) + "\n"
75          return to_return
76
77      def return_deck(self):
78          """
79          Passes down the deck for suture usage.
80          :return: The deck, a list initiated with 52
    cards.
81          """
82          return self.__deck
83
```

```
1  """
2  This Module contains the main function, which plays a
    game. You are given two different card hands and
   asked to rank
3  them. If you do so successfully, you earn a point. If
    you fail once, you receive a game over. You win the
   game if
4  you can correctly rank each card until there are not
   enough cards in the deck to draw another hand.
5  Due to a shortage of time and knowledge, and the
   frank reality that I don't know what I am doing,
6  the main() function is very obviously incomplete.
7  """
8  """
9  Help received from CS Help Desk, Loudi (from class),
   and Eric Zhao.
10 """
11 """
12 I affirm that I have carried out the attached
   academic endeavors with full academic honesty, in
13 accordance with the Union College Honor Code and the
   course syllabus.
14 """
15
16 from poker_hand import *
17
18 def main():
19     """
20     The main function that plays the game. See module
    doc_string above.
21     Many of the lines here would have been replaced
   with functions, given more time.
22     """
23
24     deck = Deck()
25     deck.shuffle()
26
27     list_1 = []
28     list_2 = []
29     for card in range(0,MAX_CARDS_IN_HAND):
30         list_1.append(deck.deal_one())
```

```
31          list_2.append(deck.deal_one())
32
33
34      if deck.remaining_cards_in_deck() >
    MAX_CARDS_IN_HAND:
35
36          # present player with 2 hands
37          hand_1 = PokerHand(list_1)
38          hand_2 = PokerHand(list_2)
39          hand_type_1 = 0
40          hand_type_2 = 0
41          points = 0
42          print(hand_1.__str__())
43          print(hand_2.__str__())
44
45          # GUTS GO HERE \/\/\/
46
47          if hand_1.determine_hand() == 'flush':
48              hand_type_1 = 'flush'
49              hand_1 = hand_1.rank_of_flush()
50          elif hand_1.determine_hand() == 'two_pair':
51              hand_type_1 = 'two_pair'
52              hand_1 = hand_1.rank_of_two_pair()
53          elif hand_1.determine_hand() == 'pair':
54              hand_type_1 = 'pair'
55              hand_1 = hand_1.rank_of_pair()
56          else: # high_card
57              hand_1.determine_high_card()
58              hand_type_1 = 'high_card'
59              hand_1 = hand_1.rank_of_high_card()
60
61          if hand_2.determine_hand() == 'flush':
62              hand_type_2 = 'flush'
63              hand_2 = hand_2.rank_of_flush()
64          elif hand_2.determine_hand() == 'two_pair':
65              hand_type_2 = 'two_pair'
66              hand_2 = hand_2.rank_of_two_pair()
67          elif hand_2.determine_hand() == 'pair':
68              hand_type_2 = 'pair'
69              hand_2 = hand_2.rank_of_pair()
70          else: # high_card
```

```
 71              hand_2.determine_high_card()
 72              hand_type_2 = 'high_card'
 73              hand_2 = hand_2.rank_of_high_card()
 74
 75          if hand_type_1 == hand_type_2: #both hands
    are flushes or pairs or ect.
 76
 77              if hand_1 == 'jack':
 78                  hand_1 = str(11)
 79              if hand_1 == 'queen':
 80                  hand_1 = str(12)
 81              if hand_1 == 'king':
 82                  hand_1 = str(13)
 83              if hand_1 == 'ace':
 84                  hand_1 = str(14)
 85
 86              if hand_2 == 'jack':
 87                  hand_2 = str(11)
 88              if hand_2 == 'queen':
 89                  hand_2 = str(12)
 90              if hand_2 == 'king':
 91                  hand_2 = str(13)
 92              if hand_1 == 'ace':
 93                  hand_2 = str(14)
 94
 95              if hand_1 > hand_2:
 96                  answer = hand_1
 97              elif hand_1 < hand_2:
 98                  answer = hand_2
 99              else:
100                  pass
101
102          else: pass
103
104
105          print(hand_1)
106          print(hand_2)
107          # GUTS GO HERE /\/\/\
108
109          # have input to see if guess matches answer
110          guess = input("Which hand has a greater
```

```
value? Answer 1 or 2\n")
        if guess == 2:
            guess = -1


        # if they match, add a point
        if (hand_1.compare_to(hand_2) == 1 and guess == 1) or (hand_1.compare_to(hand_2) == -1 and guess == -1):
            points += 1
        else:
            print(f'Game over, you scored {points} points.')
            return None


        # if its not a match, end the game

        # game ends when loop closes (too little cards)

        pass #PASS for debugging
    else:
        print('Game over! There are not enough cards in the deck to play another round')


main()

```

```
 1 """
 2 The poker_hand module. Contains methods of the
   PokerHand class, related to a poker hand (typically 5
    cards).
 3 """
 4
 5 from deck import *
 6 MAX_CARDS_IN_HAND = 5
 7
 8
 9 class PokerHand:
10     """
11     A class based around a hand for Poker. Poker uses
    hands of 5 cards, from a deck of 52 cards.
12     There are 13 differently ranked cards for each of
    the 4 suits.
13     """
14
15     def __init__(self, list_of_cards):
16         """
17         Creates a hand of cards, suing the parameter
   list_of_cards.
18         :param: list_of_cards: A list of cards used
   to create a hand.
19         """
20
21         self.__hand = list_of_cards.copy()
22
23     def add_card(self, card_object):
24         """
25         Appends a card to a hand.
26         :param: card_object: The card that will be
   appended to a hand.
27         """
28         self.__hand.append(card_object)
29
30     def get_ith_card(self, index):
31         """
32         A method to grab a card from a list, chosen
   by its index.
33         :param index: The index/card of a list/hand.
```

```
34          :return: Returns an index of the list "__hand
    ", or None if the index is invalid.
35          """
36          if (index < 0) or (index > MAX_CARDS_IN_HAND
    ):
37              return None
38          else:
39              return self.__hand[index]
40
41      def __str__(self):
42          """
43          Represents the hand as 5 easily readable
    cards.
44          :return: Returns a variable 'to_return' that
    is continuously updated
45          with a new easily understandable card
    throughout the for loop.
46          Starts with header 'HAND CONTENTS' and goes
    through a loop of the hand.
47          When the end of the loop is reached, the
    resulting print is the header
48          above every card from the hand, each placed
    neatly on a line.
49          """
50          to_return = "HAND CONTENTS:\n"
51          for index in range(0, MAX_CARDS_IN_HAND):
52              card = self.get_ith_card(index)
53              to_return += str(card) + "\n"
54          return to_return
55
56      def list_hands_ranks(self):
57          """
58          Takes obfuscated card data and turns it into
    the easily understandable ranks of a 5-card hand.
59          :return: An easily readable list of 5 cards'
    ranks.
60          """
61          rank_list = []
62          for i in range(0,MAX_CARDS_IN_HAND):
63              card_index = self.get_ith_card(i)
64              card = card_index.find_rank()
```

```
65
66              rank_list.append(card)
67          return rank_list
68
69      def determine_flush(self):
70          """
71          A function that categorizes a card hand as a
     flush.
72          :return: Returns True if the hand is a flush
    , False otherwise.
73          """
74
75          card = self.get_ith_card(0)
76          card_suit = card.find_suit()
77
78          for index in range(0, MAX_CARDS_IN_HAND):
79              if card_suit != self.get_ith_card(index
    ).find_suit():
80                  return False
81          return True
82
83      def rank_of_flush(self):
84          """
85          This function will only go off when
    comparing flushes.
86          :return: Returns the highest rank of a flush
    , to be compared against
87          another flush.
88          """
89
90          flush = self.__hand
91          for rank in reversed(STRING_RANKS):
92              for card in flush:
93                  cards_rank = card.find_rank()
94                  if cards_rank == rank:
95                      return rank
96
97
98      def determine_pair(self):
99          """
100         A function that categorizes the card hand as
```

```
100  a pair.
101          :return: Returns True if the hand is a pair
    , False otherwise.
102          """
103
104          for i in range(0,(MAX_CARDS_IN_HAND - 1)):
105              for j in range(i+1,MAX_CARDS_IN_HAND):
106                  if self.list_hands_ranks()[i] ==
    self.list_hands_ranks()[j]:
107                      return True
108          return False
109
110      def rank_of_pair(self):
111          """
112          This function will only go off when
    comparing pairs.
113          :return: Returns the highest rank of a pair
    , to be compared against
114          another pair, or None, in case of an
    unpredictable bug.
115          """
116          for i in range(0,(MAX_CARDS_IN_HAND - 1)):
117              for j in range(i+1,MAX_CARDS_IN_HAND):
118                  if self.list_hands_ranks()[i] ==
    self.list_hands_ranks()[j]:
119                      rank = self.list_hands_ranks()[i
    ]
120                      return rank
121          return None
122
123      def break_tie_pair(self):
124          """
125          This function will only go off when
    comparing tied pairs.
126          :return: Returns the highest "chaser rank"
    of a pair, to be compared against
127          another pair's chaser. Currently Unfinished.
128          """
129          pass
130
131      def determine_two_pair(self):
```

```
132         """
133         A function that categorizes the card hand as
     a two_pair.
134         :param: hand: A hand of 5 random cards.
135         :return: Returns True if the hand is a
    two_pair, False otherwise.
136         """
137 
138         matching_card_1 = 0
139         matching_card_2 = 0
140         x = False
141         hand = self.list_hands_ranks()
142 
143         for i in range(0, MAX_CARDS_IN_HAND - 1):
144             for j in range((i + 1),
    MAX_CARDS_IN_HAND):
145                 if self.list_hands_ranks()[i] ==
    self.list_hands_ranks()[j]:
146                     matching_card_1 = hand[i]
147                     matching_card_2 = hand[j]
148                     x = True
149         if x:
150             hand.remove(matching_card_1)
151             hand.remove(matching_card_2)
152         else:
153             return False
154 
155         for i in range(0, 2):
156             for j in range((i + 1), 3):
157                 if hand[i] == hand[j]:
158                     hand.append(matching_card_1)
159                     hand.append(matching_card_2)
160                     return True
161         hand.append(matching_card_1)
162         hand.append(matching_card_2)
163         return False
164 
165     def rank_of_two_pair(self):
166         """
167         This function will only go off when
    comparing two_pairs.
```

```
168         :return: Returns the highest rank of a
    two_pair, to be compared against
169         another two_pair.
170         """
171         matching_card_1 = 0
172         matching_card_2 = 0
173         rank_1 = None
174         rank_2 = None
175         two_pair_hand = self.list_hands_ranks()
176 
177         for i in range(0, MAX_CARDS_IN_HAND - 1):
178             for j in range((i + 1),
    MAX_CARDS_IN_HAND):
179                 if self.list_hands_ranks()[i] ==
    self.list_hands_ranks()[j]:
180                     matching_card_1 = two_pair_hand[
    i]
181                     matching_card_2 = two_pair_hand[
    j]
182                     rank_1 = matching_card_1
183 
184         two_pair_hand.remove(matching_card_1)
185         two_pair_hand.remove(matching_card_2)
186 
187         for i in range(0, 2):
188             for j in range((i + 1), 3):
189                 if two_pair_hand[i] == two_pair_hand
    [j]:
190                     rank_2 = two_pair_hand[i]
191 
192         if rank_1 > rank_2:
193             return rank_1
194         elif rank_1 < rank_2:
195             return rank_2
196         else:
197             return rank_1 # in cases of 4_of_a_kind
    scenarios
198 
199     def determine_high_card(self):
200         """
201         A function that categorizes the card hand as
```

```
201  a high_card. This funtion
202          will always return True if run.
203          :return: Returns True if the hand is a
    high_card. The return value will
204          always be True, because every hand that is
    not a flush,pair, or two_pair
205          must be a high_card.
206          """
207          return True
208
209      def rank_of_high_card(self):
210          """
211          Only to be used when ranking high_cards.
212          :return: Returns the rank of the high card.
213          """
214
215          hand = self.__hand
216
217          for rank in reversed(STRING_RANKS):
218              for card in hand:
219                  cards_rank = card.find_rank()
220                  if cards_rank == rank:
221                      return rank
222
223      def determine_hand(self):
224          """
225          A method to determine what type a hand is.
226          :return: returns a string representing what
    the hand is.
227          """
228          if self.determine_flush():
229              return 'flush'
230          elif self.determine_two_pair():
231              return 'two_pair'
232          elif self.determine_pair():
233              return 'pair'
234          else:
235              self.determine_high_card()
236              return 'high_card'
237
238
```

```
239     def compare_to(self, other_hand):
240         """
241         Determines which of two poker hands is worth more. Returns an int
242         which is either positive, negative, or zero depending on the comparison.
243         :param: self: The first hand to compare
244         :param: other_hand: The second hand to compare
245         :return: a negative number if self is worth LESS than other_hand,
246         zero if they are worth the SAME (a tie), and a positive number if
247         self is worth MORE than other_hand
248         """
249 
250         # This definitely needs a fix
251 
252         if self.__hand > other_hand:
253             return 1
254         elif self.__hand < other_hand:
255             return -1
256         else:
257             return 0
258 
259 # debug_list_1 = []
260 #
261 # c1 = Card(2,'S')
262 # c2 = Card(11,'C')
263 # c3 = Card(3,'H')
264 # c4 = Card(3,'C')
265 # c5 = Card(11,'H')
266 # debug_list_1.append(c1)
267 # debug_list_1.append(c2)
268 # debug_list_1.append(c3)
269 # debug_list_1.append(c4)
270 # debug_list_1.append(c5)
271 #
272 # debug_list_2 = []
273 #
274 # d1 = Card(2,'D')
```

```
275 # d2 = Card(11,'H')
276 # d3 = Card(3,'S')
277 # d4 = Card(3,'S')
278 # d5 = Card(11,'C')
279 # debug_list_2.append(c1)
280 # debug_list_2.append(c2)
281 # debug_list_2.append(c3)
282 # debug_list_2.append(c4)
283 # debug_list_2.append(c5)
284
285 # h = PokerHand(debug_list)
286 # print(PokerHand.list_hands_ranks(h))
287
288 #m = PokerHand(debug_list_1)
289 #print(PokerHand.determine_two_pair(m))
290
291 #print(PokerHand.rank_of_two_pair(m))
292 #print(PokerHand.determine_two_pair(m))
```