CardHand Class

By Marc Stern
MS in Data Science at Montclair State University

August 5, 2020

Introduction

The game *Hearts* is a card game that bases its rules on suit. If a player puts down a card, all other players must play a card of that suit, unless they do not have it, then they can play a card of their choice. After all players have played a card, the loser of the round takes the pile and stores in it his/her discard pile. After all the cards are played, the players count their respective amount of hearts in their discard pile, and the winner of the round is the player with the fewest number of hearts. This is an example of a game where players might sort their cards by suit, but not necessarily by rank (ie. 2, 7, Jack, Ace). The CardHand class is a fun representation of how to use Python in representing a game of cards. The goal of this class is have our hand of cards organized in such a way that a *Hearts* player would, sorted by suit.

Design

The CardHand class is designed with three methods, *add_card*, *play*, and *all_of_suit*. Additionally, the class supports an __iter__ generator that yields a card's rank and suit. Firstly, however, let's look at how the class is initialized.

When a CardHand object is created, the __init__ creates two dictionaries, a set, and a positional list. The first dictionary whose keys are suit names ('spades', 'hearts, etc.) with values that begin as empty lists, which will represent the cards in our hand. The second dictionary has the same keys, but its values are sets. This will store the cards in our hand as well. We use the first dictionary to store positions in the positional list that will be accessed to return a card's rank and suit, and we also store them in a set to check for the validity of a card. This second dictionary is crucial to preventing duplicates since a real card deck has none. We also create a set with the valid rank choices. This will prevent a player from trying to play a card that does not exist, such as the 13 of Hearts, which is really the King of Hearts.

Moreover, looking at the *add_card* method, two arguments are needed to create the card, a rank and a suit. First we check the validity of the card, making sure it both exists in the deck, but is not already in our hand. If it passes these tests, we check it's suit, assign a position to the positional list, and store that in our *hand* dictionary, as well as our *valid* dictionary. For reference, the determined order of suit is Spades, Hearts, Clubs, then Diamonds. This determines what checks we make when adding a card to the hand. If a card is Spades we add it

to the beginning of the positional list, keeping Spades on the left of our hand. If a card is Diamonds, we add it to the end of the positional list, keeping Diamonds on the right of our hand. For the second highest suit, Hearts, we check to see if any Hearts already exist. If so, we look at our most recently appended Hearts position in the list within our Hearts dictionary, and simply add it after that card. If there are no Hearts, then we check the preceding suit, Spades, and add after its most recently appended position. If there are also no Spades in our hand, we can just add it to the beginning of the positional list. We use the same methodology for the third highest suit, Clubs, checking first for its own suit, then the two preceding suits.

The *play* method takes a suit as an argument, and plays the most recently added card of that suit. By play, what is meant is that it returns the card's rank and suit, and then removes it from the hand, deleting its position in the positional list, and popping the value in the dictionary's list corresponding to the selected suit. We first check to see if a card of that suit exists in our hand or has already been played. If so, it will be inside our *valid* dictionary. Once it passes this check, the card is popped, returning its value and deleting its position. If no cards of that suit are in our hand, then it plays the most recently added card to our hand, which could be of any suit. In the event that there are no cards left in our hand, we cannot play, and a message is returned stating as such. When a card is played, as mentioned, its position is deleted from the positional list, and is also popped from its corresponding list. We don't remove it from our *valid* dictionary's set, because we want to account for the fact that the card has been played. While this is not the case in all card games, we assume that our card game, like that of *Hearts*, involves a discard pile, and not returning a card to the deck after it is played. Therefore, after all possible cards are added to the hand and/or played, there are no cards left in the deck, and we reshuffle for a new round.

The design of the *all_of_suit* method loops over all card positions stored in the positional list. Checking for matches on suit, which is passed as an argument for this method, if a match is found, it appends the positions element, a key/value tuple with the rank and suit, to a new list created in this method. Upon completion, this method will return a list of all cards.

Implementation

```
from ch07.positional_list import PositionalList

class CardHand:

    def __init__(self):
        self.hand = {'spades': [], 'hearts': [], 'clubs': [], 'diamonds': []}
        self.valid = {'spades': set(), 'hearts': set(), 'clubs': set(), 'diamonds': set()}
        self.r = {2, 3, 4, 5, 6, 7, 8, 9, 10, 'J', 'Q', 'K', 'A'}
        self.p = PositionalList()

    def add_card(self, r, s):
        new_card = (r, s)

        if s not in self.valid:
            raise KeyError('suit is not valid')

        if r not in self.r:
            raise ValueError(f'rank is not valid\nchoose from below:\n{self.r}')

        if r in self.valid[s]:
            raise ValueError('card is already in hand or has already been played')

        if s == 'spades':
            pos = self.p.add_first(new_card)

        if s == 'hearts':
            if len(self.hand['hearts']) > 0:
                pos = self.p.add_after(self.hand['hearts'][-1], new_card)
            elif len(self.hand['spades']) > 0:
                pos = self.p.add_after(self.hand['spades'][-1], new_card)
            else:
                pos = self.p.add_first(new_card)

        if s == 'clubs':
            if len(self.hand['clubs']) > 0:
                pos = self.p.add_after(self.hand['clubs'][-1], new_card)
            elif len(self.hand['hearts']) > 0:
                pos = self.p.add_after(self.hand['hearts'][-1], new_card)
            elif len(self.hand['spades']) > 0:
                pos = self.p.add_after(self.hand['spades'][-1], new_card)
            else:
                pos = self.p.add_first(new_card)

        if s == 'diamonds':
            pos = self.p.add_last(new_card)

        self.hand[s].append(pos)
        self.valid[s].add(r)
```

```
def play(self, s):
    if s in self.hand:
        if not len(self.hand[s]) == 0:
            card = self.hand[s].pop()
            e = card.element()
            self.p.delete(card)
            return e

    if not self.p.is_empty():
        i = self.p.last()
        e = i.element()
        self.hand[e[1]].pop()
        self.p.delete(i)
        return e

    return 'hand is empty'

def __iter__(self):
    for card in self.p:
        yield card.element()

def all_of_suit(self, s):
    l = []
    for card in self.p:
        if card.element()[1] == s:
            l.append(card.element())
    return l
```

Run Time

The run time for the *all_of_suits* method is simply *O(n)* where n represents the number of cards in my hand. Because we have to iterate through all the cards in the hand, checking the suit of each one, the worst-case is also the expected case.

Methods *play* and *add_card* both run in constant time, or *O(1)*. This is accomplished by using dictionaries, which can be indexed, and a list, which is only used in the context of returning the last item in the list at *index = [-1]* and popping the item. We see this visually, as there are only if/else checks in the code, and neither of these methods include loops.

Testing

We need only test for a few different cases. Firstly, we check to see that if a card is being added, it is a valid card, meaning its rank is an integer from 2 to 10, or it is a royal card represented by the values ‘J’, ‘Q’, ‘K’, and ‘A’. We also check to see if the suit matches one of the four options. If the suit is invalid, it will raise a KeyError; if the rank is invalid, it will raise a ValueError, and then it will display a list of valid options in string format. Furthermore, if the card is valid, we test to see if it already exists in the hand, or was already played. That will also raise a ValueError, returning with a helpful statement.

To actually test the methods and these Exceptions, we just add cards that are valid, invalid, already in the hand, and ones that have already been played. We can also test the *play* method by trying to play a card whose suit is in my hand, and also a case where the suit is not in my hand. This is best described with the following test code, and corresponding results.

```
c = CardHand()
c.add_card(9, 'diamonds')
c.add_card('K', 'diamonds')
c.add_card(6, 'hearts')
c.add_card(10, 'spades')

a = c.all_of_suit('diamonds')
print(a)

p = c.play('clubs')
print(p)

a = c.all_of_suit('diamonds')
print(a)

p = c.play('clubs')
print(p)

p = c.play('clubs')
print(p)

p = c.play('clubs')
print(p)

p = c.play('clubs')
print(p)
```

Results:

[(9, 'diamonds'), ('K', 'diamonds')]
('K', 'diamonds')
[(9, 'diamonds')]
(9, 'diamonds')
(6, 'hearts')
(10, 'spades')
hand is empty

This test shows that we've never added a Clubs to our hand, yet by playing "Clubs", another card is still returned. And we see our program working as expected.