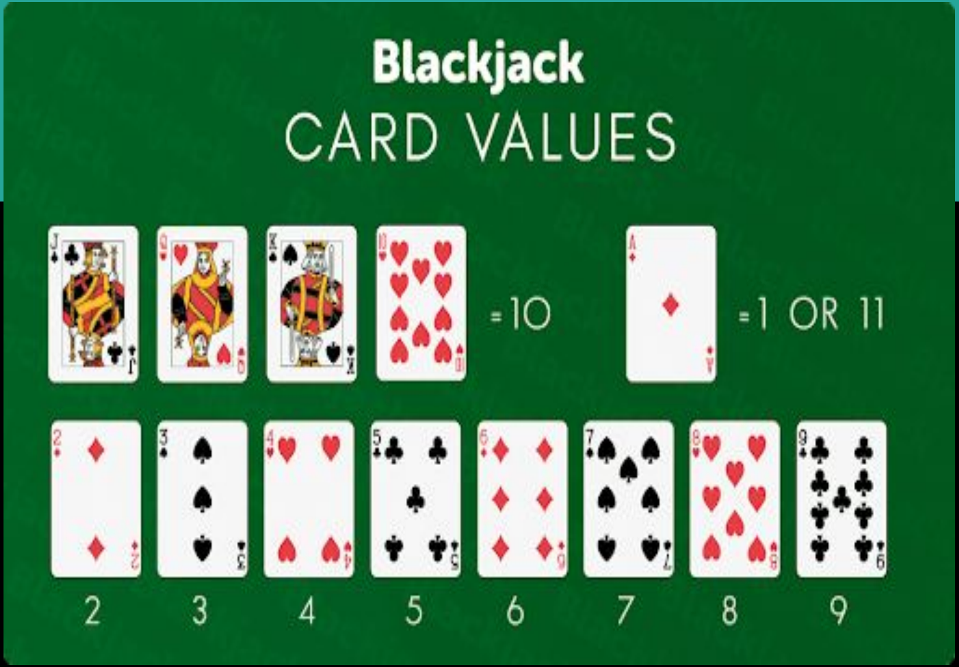
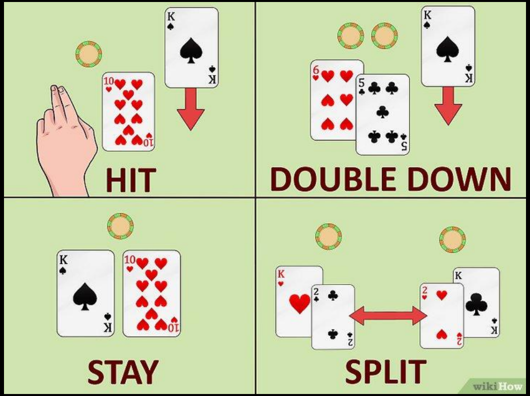
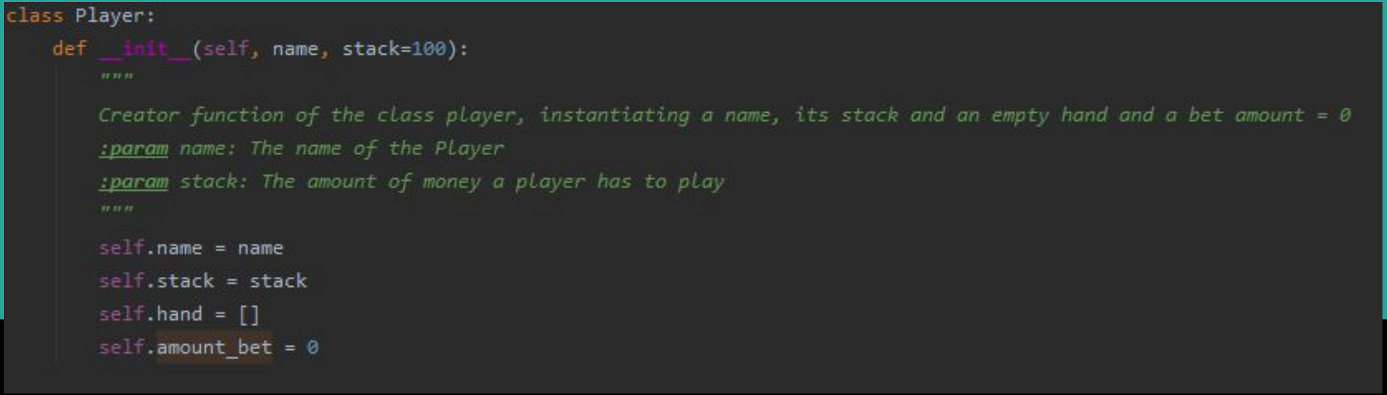
A simple python blackjack game

By Alexander Giménez





1- Understand the rules of the game



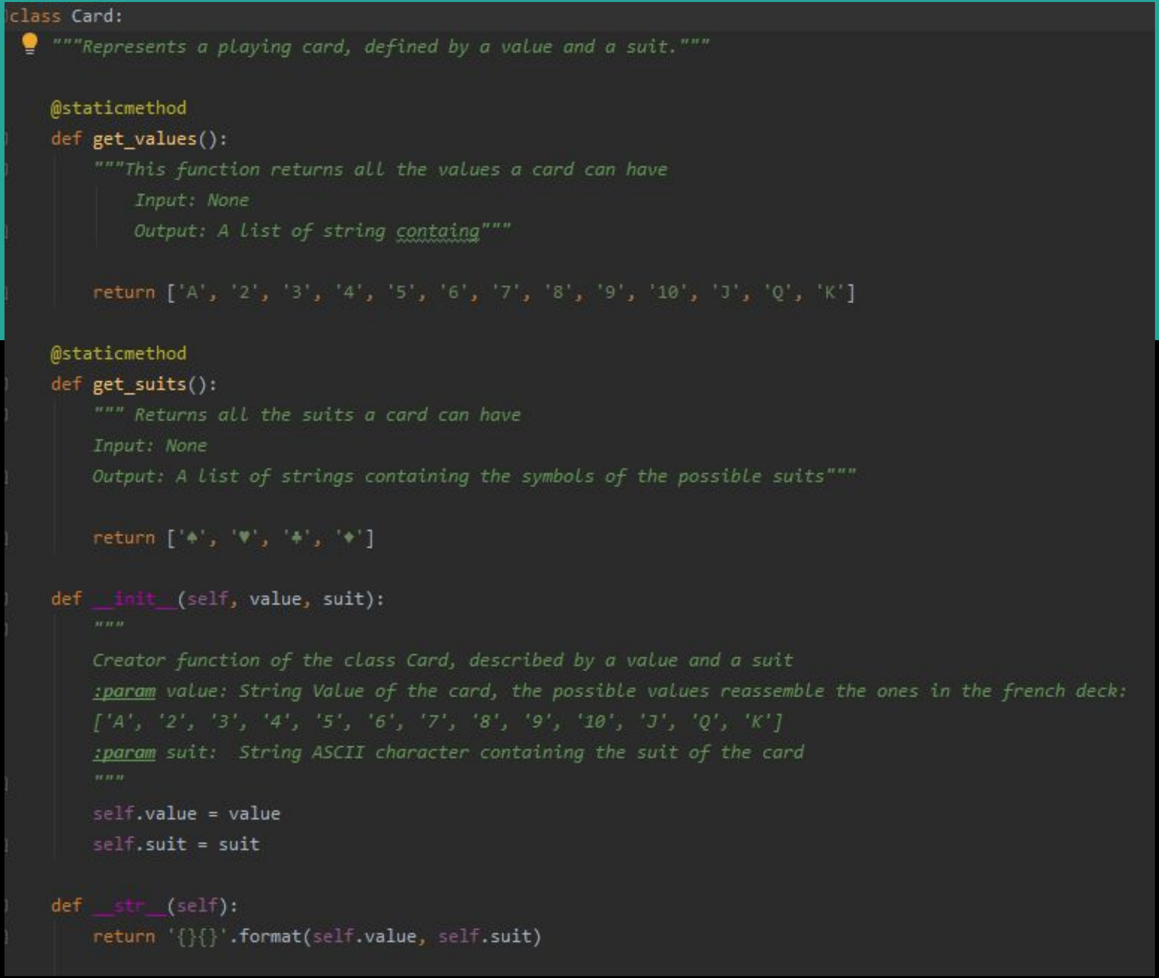
```
class Player:
    def __init__(self, name, stack=100):
        """
        Creator function of the class player, instantiating a name, its stack and an empty hand and a bet amount = 0
        :param name: The name of the Player
        :param stack: The amount of money a player has to play
        """
        self.name = name
        self.stack = stack
        self.hand = []
        self.amount_bet = 0
```

2- Abstract the problem

Think of which entities constitute the problem and create them as clases ,while the “other qualities” that those entities have will constitute their Attributes.

Start by something simple and add more features as you go along, otherwise you can trap yourself if you don’t plan ahead enough!

Think all the actions that you might be doing multiple times across your code and abstract them as methods. If some action has a clear task to do is also good to abstract it into a method.

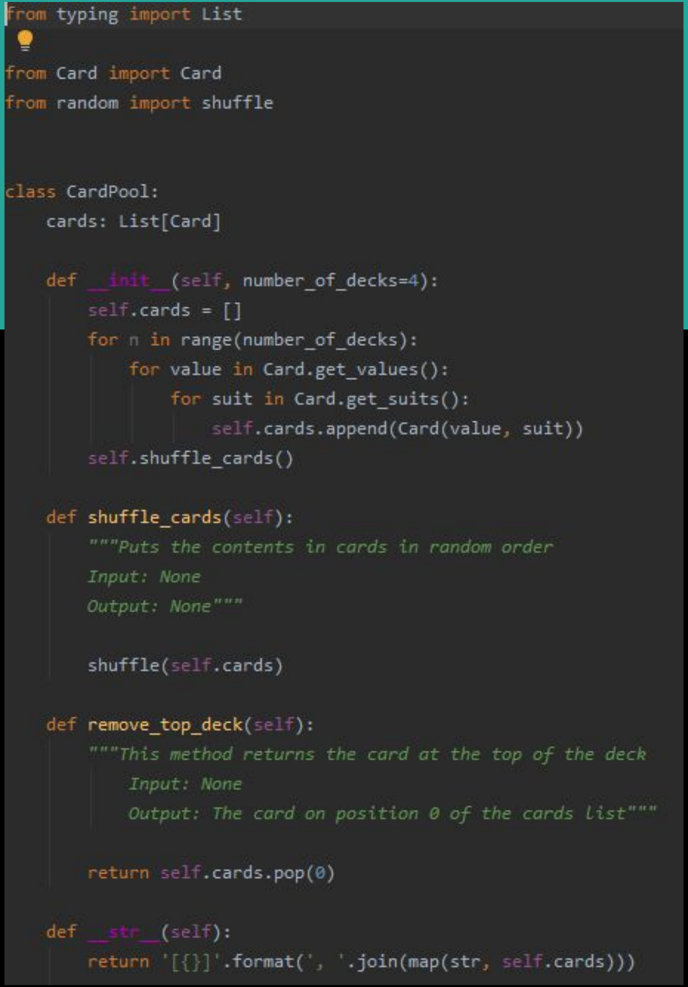


```python
class Card:
    """Represents a playing card, defined by a value and a suit."""

    @staticmethod
    def get_values():
        """This function returns all the values a card can have
            Input: None
            Output: A list of string containg"""

        return ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']

    @staticmethod
    def get_suits():
        """ Returns all the suits a card can have
        Input: None
        Output: A list of strings containing the symbols of the possible suits"""

        return ['♠', '♥', '♣', '♦']

    def __init__(self, value, suit):
        """
        Creator function of the class Card, described by a value and a suit
        :param value: String Value of the card, the possible values reassemble the ones in the french deck:
        ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']
        :param suit:  String ASCII character containing the suit of the card
        """
        self.value = value
        self.suit = suit

    def __str__(self):
        return '{}{}'.format(self.value, self.suit)
```

3- Code!

Let’s dive into some of the code decisions!

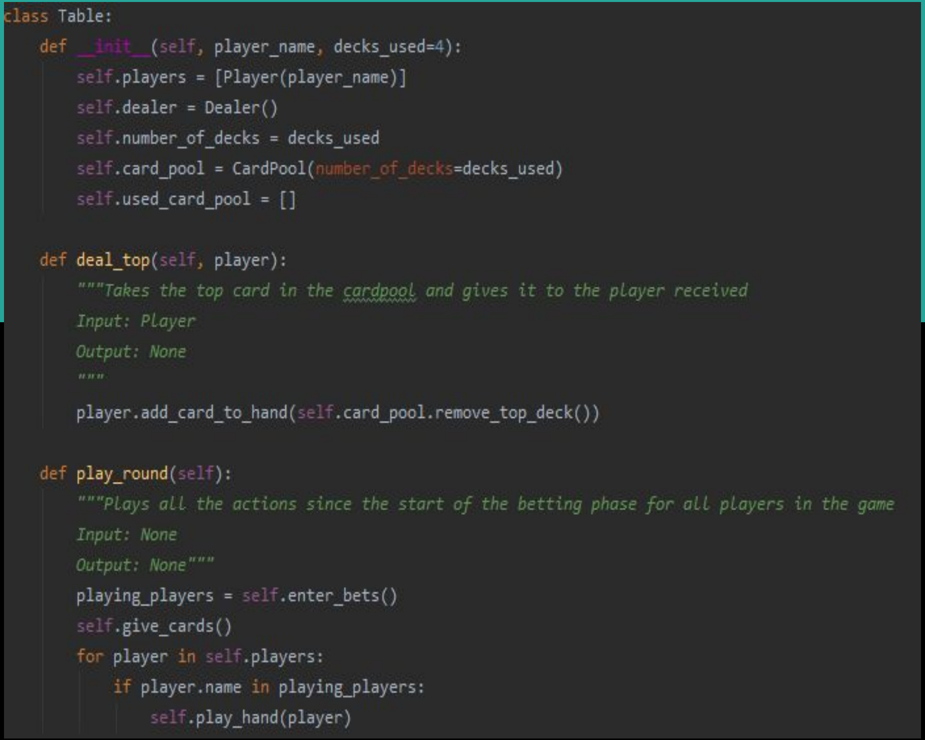
The Card class uses 2 attributes to be represented, value and suit. The good thing about suits being represented in a static method is that even if we wanted to add more suits we can do so so easily!
A simple way to represent a card that allows flexibility!



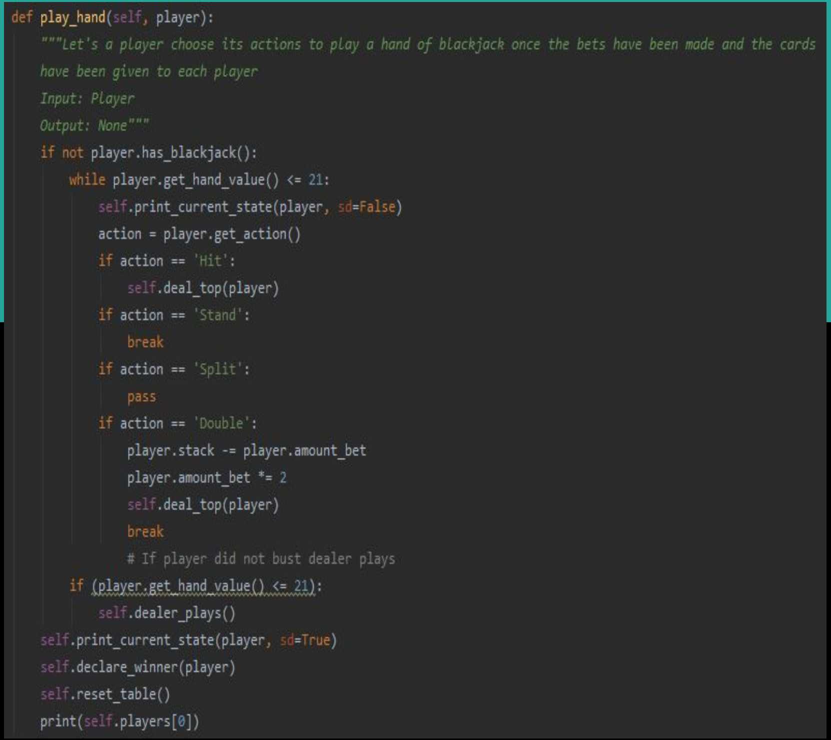
```
from typing import List

from Card import Card
from random import shuffle


class CardPool:
    cards: List[Card]

    def __init__(self, number_of_decks=4):
        self.cards = []
        for n in range(number_of_decks):
            for value in Card.get_values():
                for suit in Card.get_suits():
                    self.cards.append(Card(value, suit))
        self.shuffle_cards()

    def shuffle_cards(self):
        """Puts the contents in cards in random order
        Input: None
        Output: None"""

        shuffle(self.cards)

    def remove_top_deck(self):
        """This method returns the card at the top of the deck
            Input: None
            Output: The card on position 0 of the cards list"""

        return self.cards.pop(0)

    def __str__(self):
        return '[{}]'.format(', '.join(map(str, self.cards)))
```

The CardPool class reassembles the total amount of decks used in a game.

Ic chose to put in a separate class because it allows flexibility when creating and manipulating it



```python
class Table:
    def __init__(self, player_name, decks_used=4):
        self.players = [Player(player_name)]
        self.dealer = Dealer()
        self.number_of_decks = decks_used
        self.card_pool = CardPool(number_of_decks=decks_used)
        self.used_card_pool = []

    def deal_top(self, player):
        """Takes the top card in the cardpool and gives it to the player received
        Input: Player
        Output: None
        """
        player.add_card_to_hand(self.card_pool.remove_top_deck())

    def play_round(self):
        """Plays all the actions since the start of the betting phase for all players in the game
        Input: None
        Output: None"""
        playing_players = self.enter_bets()
        self.give_cards()
        for player in self.players:
            if player.name in playing_players:
                self.play_hand(player)
```



```python
def play_hand(self, player):
    """Let's a player choose its actions to play a hand of blackjack once the bets have been made and the cards
    have been given to each player
    Input: Player
    Output: None"""
    if not player.has_blackjack():
        while player.get_hand_value() <= 21:
            self.print_current_state(player, sd=False)
            action = player.get_action()
            if action == 'Hit':
                self.deal_top(player)
            if action == 'Stand':
                break
            if action == 'Split':
                pass
            if action == 'Double':
                player.stack -= player.amount_bet
                player.amount_bet *= 2
                self.deal_top(player)
                break
                # If player did not bust dealer plays
        if (player.get_hand_value() <= 21):
            self.dealer_plays()
    self.print_current_state(player, sd=True)
    self.declare_winner(player)
    self.reset_table()
    print(self.players[0])
```

Implementation of the core of the game and some interesting observations:

- Sd parameter is quite important in this implementation
- We could just check if player.get_hand_value == 21 but instead we abstract it in a function that makes our code so easy to understand

4- Analize for possible improvements

1. Current build doesn’t support multiplayer, and doesn’t allow splits a quite important action for the player in blackjack.
2. Player hand structure assumes 1 dimension and could be improved towards being able to handle splits, therefore design was flawed from start.

3. AI, it we both great to add a AI player and also increase the parameters and behaviours of the dealer like allowing him to hit on soft 17.
4. Add statistics, we could just store every play in a csv file, and then load it and be able to check how we played our hands, how money we won or lost, etc.

THANK YOU