Blackjack: Using Probability to Count Cards

By Jack D

About Blackjack

- A.K.A. “Twenty-One”
- Stochastic
- 1-5 Players against a House
- Partially observable
- Goal of game is to beat the dealers hand without having a card value which exceeds 21

- Every round, each player is dealt two cards (face up) while the dealer gets two cards (one of which is face down)
  - From here, each player takes a turn. On someone's turn, they can either hit (draw a card) or stand (stop drawing cards and move on to the next player)
  - Be careful! If you draw to many cards and your card value exceeds 21, you lose your bet.
  - Drawing ‘21’ (ace and a card of value 10) is the best hand in the game.
  - After each player is done taking their turn the dealer ends on their turn.
    - The dealer must hit if the combined value of their cards is <17 and must stand if their cards values is >= 17
  - If the players hand beats the dealers, they win double their bet. If the values are the same, the player gets their money returned. Lastly, if the dealer's hand is stronger than the players, the player loses their bet.

About the Program

- Initial State: After placing bets, every player is dealt two cards
- Players: Ourselves and the House (other players can be added as well)
- Actions: When it’s our turn to interact with our hand, we can either hit or stand. We will calculate the expected value of hitting to determine how we should act (via probabilities)
- Results: After assessing probabilities, choose whether to hit or stand. If hitting, reassess probabilities again to determine if we should hit again.
- Utility: After utilizing expected values, if we win the round, we get double our bet. If we push, we get our money returned. If we lose, we lose our bet.

- Betting method: If we lose the current round, double the bet placed on the next round. If we win the current round, return our bet to the original value.

*Although this creates some outliers, most simulations turn a profit

Decision Making Logic

After being dealt an initial set of cards, if hitting, we can observe the probabilities of drawing a given value card as our next card. Based off of:

- the probabilities of drawing a card and (not) passing 21
- the probability of beating the dealer's hand (based off of the only visible card they have)
- the strength of our hand
- How much room we have left to draw with our current hand’s value

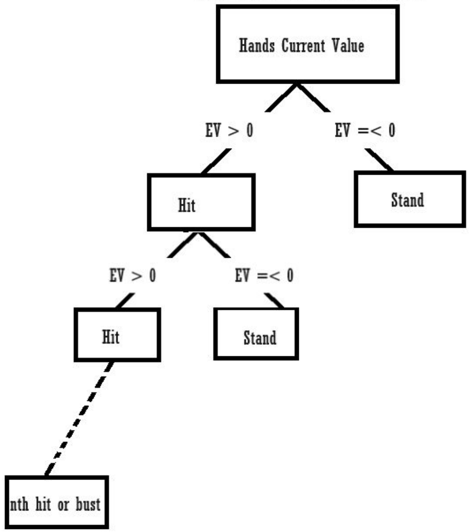
We can derive the follow statistic EV: (REDACTED)

EVHit = [redacted]
EV = EVHit - PDealer   Weighs current hand against dealers. If positive, then positive expected return on bet
PHit = Probability of hitting and not busting
PBust = Probability of hitting and busting
Curval/21 = strength of hand
(1 - Curval/21) = space left before ratio exceeds 21, goes negative if exceeds (ie room to draw)
EVHit = net expected value from hitting
PDealer = probability of dealer having stronger cards
EV = Expected value to gain from hitting with dealer in mind

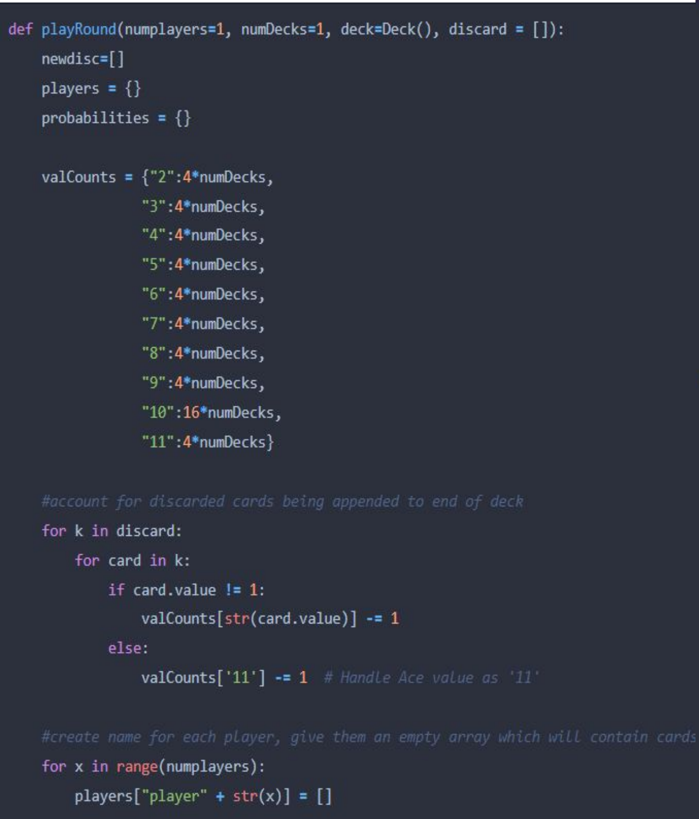
The statistic EV will tell us the expected value to gain from hitting with our current hand

- Negative EV means we will lose value
- Positive means we will gain value
- EV close to 0 means less gain

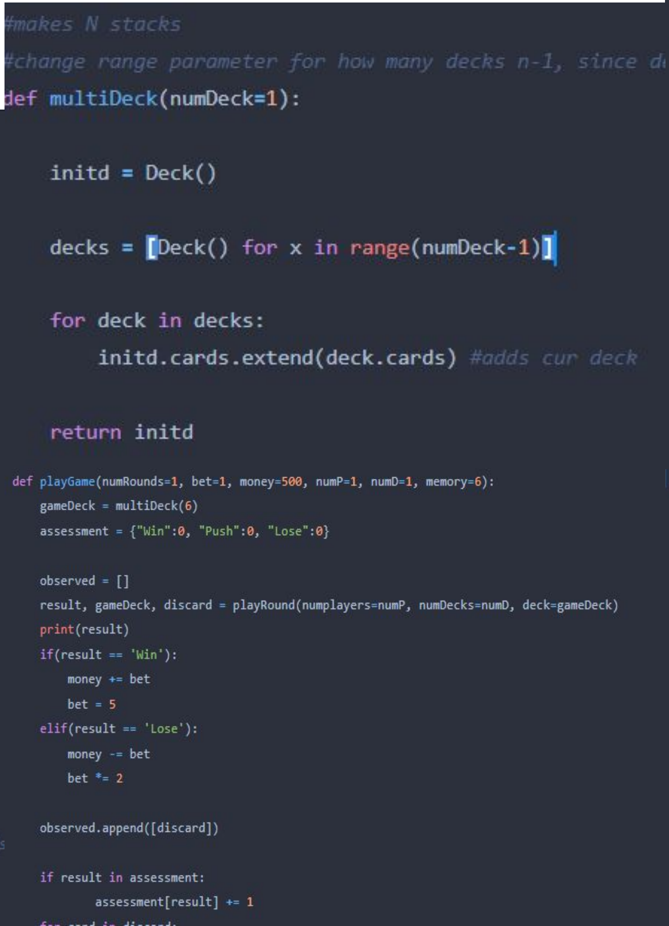
Game Tree for a Round of BlackJack



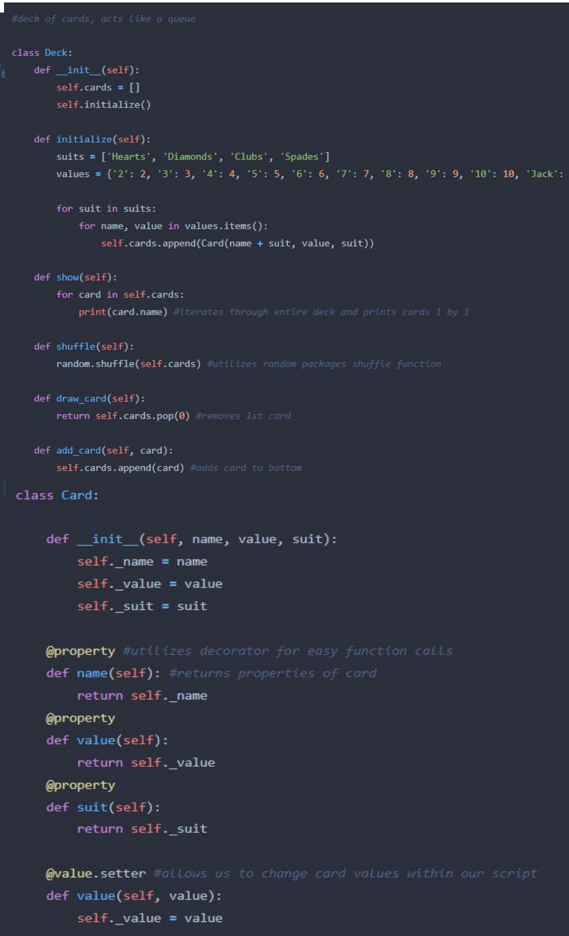
Code



```python
def playRound(numplayers=1, numDecks=1, deck=Deck(), discard = []):
    newdisc=[]
    players = {}
    probabilities = {}

    valCounts = {"2":4*numDecks,
                 "3":4*numDecks,
                 "4":4*numDecks,
                 "5":4*numDecks,
                 "6":4*numDecks,
                 "7":4*numDecks,
                 "8":4*numDecks,
                 "9":4*numDecks,
                 "10":16*numDecks,
                 "11":4*numDecks}

    #account for discarded cards being appended to end of deck
    for k in discard:
        for card in k:
            if card.value != 1:
                valCounts[str(card.value)] -= 1
            else:
                valCounts['11'] -= 1  # Handle Ace value as '11'

    #create name for each player, give them an empty array which will contain cards
    for x in range(numplayers):
        players["player" + str(x)] = []
```



```python
#makes N stacks
#change range parameter for how many decks n-1, since d
def multiDeck(numDeck=1):

    initd = Deck()

    decks = [Deck() for x in range(numDeck-1)]

    for deck in decks:
        initd.cards.extend(deck.cards) #adds cur deck

    return initd

def playGame(numRounds=1, bet=1, money=500, numP=1, numD=1, memory=6):
    gameDeck = multiDeck(6)
    assessment = {"Win":0, "Push":0, "Lose":0}

    observed = []
    result, gameDeck, discard = playRound(numplayers=numP, numDecks=numD, deck=gameDeck)
    print(result)
    if(result == 'Win'):
        money += bet
        bet = 5
    elif(result == 'Lose'):
        money -= bet
        bet *= 2

    observed.append([discard])

    if result in assessment:
            assessment[result] += 1
    for card in discard:
```



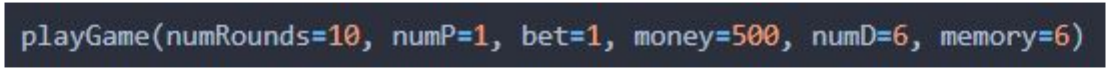
```python
#deck of cards, acts like a queue

class Deck:
    def __init__(self):
        self.cards = []
        self.initialize()

    def initialize(self):
        suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
        values = {'2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9, '10': 10, 'Jack':

        for suit in suits:
            for name, value in values.items():
                self.cards.append(Card(name + suit, value, suit))

    def show(self):
        for card in self.cards:
            print(card.name) #iterates through entire deck and prints cards 1 by 1

    def shuffle(self):
        random.shuffle(self.cards) #utilizes random packages shuffle function

    def draw_card(self):
        return self.cards.pop(0) #removes 1st card

    def add_card(self, card):
        self.cards.append(card) #adds card to bottom
class Card:

    def __init__(self, name, value, suit):
        self._name = name
        self._value = value
        self._suit = suit

    @property #utilizes decorator for easy function calls
    def name(self): #returns properties of card
        return self._name
    @property
    def value(self):
        return self._value
    @property
    def suit(self):
        return self._suit

    @value.setter #allows us to change card values within our script
    def value(self, value):
        self._value = value
```

Program in Action

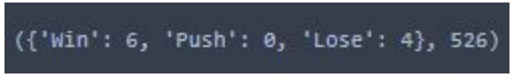
After given the following parameters

- number of rounds to play
- initial bet
- money to spend
- number of players (default is 1 = us),
- number of decks in play
- X previous rounds to retain card memorization

we can simulate a game. The function playgame will call playround which utilizes the class Card and class Deck to make a functional simulation. Playgame will return the number of rounds won, pushed, lost, and total money walked away with along with a code dump.

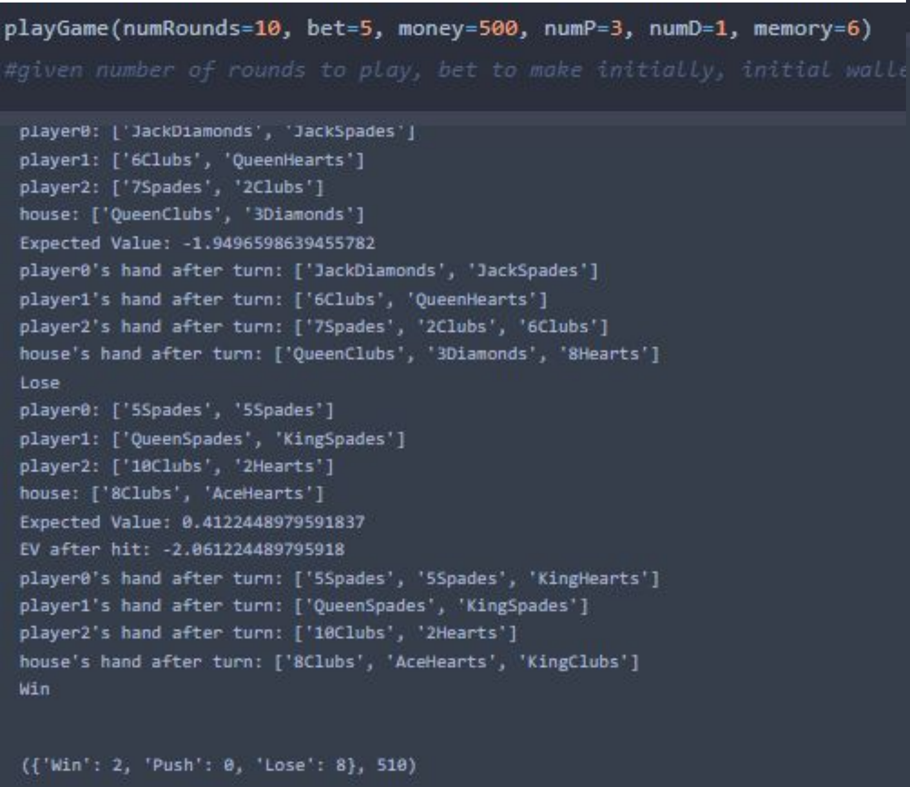


```
playGame(numRounds=10, numP=1, bet=1, money=500, numD=6, memory=6)
```



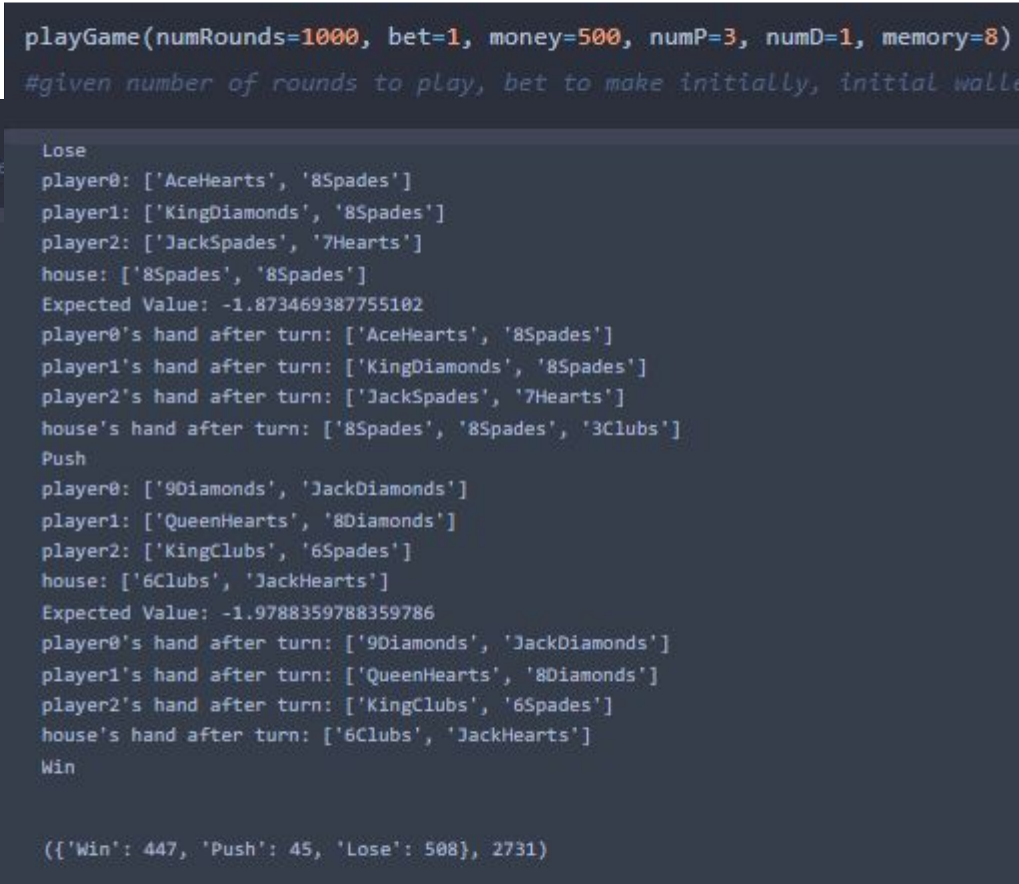
```
({'Win': 6, 'Push': 0, 'Lose': 4}, 526)
```

A few examples:



```
playGame(numRounds=10, bet=5, money=500, numP=3, numD=1, memory=6)
#given number of rounds to play, bet to make initially, initial walle
```

```
player0: ['JackDiamonds', 'JackSpades']
player1: ['6Clubs', 'QueenHearts']
player2: ['7Spades', '2Clubs']
house: ['QueenClubs', '3Diamonds']
Expected Value: -1.9496598639455782
player0's hand after turn: ['JackDiamonds', 'JackSpades']
player1's hand after turn: ['6Clubs', 'QueenHearts']
player2's hand after turn: ['7Spades', '2Clubs', '6Clubs']
house's hand after turn: ['QueenClubs', '3Diamonds', '8Hearts']
Lose
player0: ['5Spades', '5Spades']
player1: ['QueenSpades', 'KingSpades']
player2: ['10Clubs', '2Hearts']
house: ['8Clubs', 'AceHearts']
Expected Value: 0.4122448979591837
EV after hit: -2.061224489795918
player0's hand after turn: ['5Spades', '5Spades', 'KingHearts']
player1's hand after turn: ['QueenSpades', 'KingSpades']
player2's hand after turn: ['10Clubs', '2Hearts']
house's hand after turn: ['8Clubs', 'AceHearts', 'KingClubs']
Win


({'Win': 2, 'Push': 0, 'Lose': 8}, 510)
```



```
playGame(numRounds=1000, bet=1, money=500, numP=3, numD=1, memory=8)
#given number of rounds to play, bet to make initially, initial walle
```

```
Lose
player0: ['AceHearts', '8Spades']
player1: ['KingDiamonds', '8Spades']
player2: ['JackSpades', '7Hearts']
house: ['8Spades', '8Spades']
Expected Value: -1.873469387755102
player0's hand after turn: ['AceHearts', '8Spades']
player1's hand after turn: ['KingDiamonds', '8Spades']
player2's hand after turn: ['JackSpades', '7Hearts']
house's hand after turn: ['8Spades', '8Spades', '3Clubs']
Push
player0: ['9Diamonds', 'JackDiamonds']
player1: ['QueenHearts', '8Diamonds']
player2: ['KingClubs', '6Spades']
house: ['6Clubs', 'JackHearts']
Expected Value: -1.9788359788359786
player0's hand after turn: ['9Diamonds', 'JackDiamonds']
player1's hand after turn: ['QueenHearts', '8Diamonds']
player2's hand after turn: ['KingClubs', '6Spades']
house's hand after turn: ['6Clubs', 'JackHearts']
Win


({'Win': 447, 'Push': 45, 'Lose': 508}, 2731)
```