CS4 Blackjack

Due 11:59 PM, Wednesday, May 9, 2018

Note: This is a partner-optional project. You may choose any partner you like. Please remember that, as per the course syllabus, there are no late days available on projects.

Premise

Ash has reached a new gym, but the gym leader will not accept his challenge until he masters the game of Blackjack.

In order to understand the basics, Ash and Pikachu begin with the `Card` class to represent the individual playing cards, followed by a `Hand` class to represent the multiple cards they are holding. And then a `Shoe` class to contain decks of playing cards. Then, before playing interactively, they complete the `BlackjackHand` class by writing a method to calculate the best value of their hand. Gotta catch 'em all!

After playing interactively Ash realizes he isn't doing very well. This is because mastering Blackjack requires more than understanding the rules, it requires a strategy! His gym leader provides him with some automated blackjack playing code, and provides 5 different strategy tasks for Ash and Pikachu to complete to in order to become a Blackjack masters.

First, they will create and compare some simple AI strategies. Then they will create and analyze a mathematically optimal strategy (when using an infinite deck) called. Finally, they will develop and explore strategies involving foresight and card counting.

Introduction

In this project you will complete six code files: `setup1.py`, `setup2.py`, `ai_simple3.py`, `ai_perfect4.py`, `ai_clairvoyant5.py`, `ai_hilo6.py`. They will be completed in this order. You will also make Jupyter notebooks to exhibit your understanding and detail your experiments related to each ai strategy you develop. These notebooks will be called `ai_simple3.ipynb`, `ai_perfect4.ipynb`, `ai_clairvoyant5.ipynb`, `ai_hilo6.ipynb`.

Important Notes

This assignment requires that if you work with a partner, you use pair programming. Please refer to Piazza for more information on finding partners and collaboration policy clarifications. If you can't find a partner, we have added a partner-searching post to Piazza that you can use to help you find a partner.

You and your partner (or yourself only, if you're working alone) must sign up at the Google form **here** by **11:59pm, Thursday, May 3**. **You must still fill out the form if you are working independently.**

Installation and Handin

Project setup. For each project, there may be support files that you will need to complete the assignment. These can be copied to your home directory by using the `cs4_install` command in a CIT Terminal window. For this project, type the command:

```
cs4_install blackjack
```

There should now be a `blackjack` folder within your `projects` directory. Using Terminal, you can move into the folders with the cd command:

```
cd ~/course/cs0040/projects/blackjack
```

Project hand-in. When you and your partner are ready to submit the files for Part 1: Markstrings, run:

```
cs4_handin blackjack
```

from a CIT Terminal window from your `~/course/cs0040/projects/blackjack` directory, and the entire contents of the directory will be handed in.

You can re-submit this assignment using the `cs4_handin` command at any time, but only your most recent submission with be graded. **Only a single partner should hand in the project.** If more than one partner hands in the project, the most recent submission from either of the partners will be graded.

PLEASE READ BEFORE CONTINUING

For the code to output correctly, please use PyCharm to edit and run your files OR if you are using remote SSH you can add an additional flag, as shown below:

`PYTHONIOENCODING='utf8' python3 ****FILE NAME****`

If you cannot get it to output correctly, you can remove the ascii characters from the `__repr__` functions.

Phase 1a: General Setup `[setup1.py]`

In this part, you'll be editing the `setup1.py` file.

Card Class

Each `Card` object has two attributes:

- a **rank**, which is an UPPERCASE string equal to:
  - the card's numeric value for cards 2-10
  - `'K'` for King, `'J'` for Jack, `'Q'` for Queen and `'A'` for Ace
- a **suit**, which is an UPPERCASE string equal to:
  - `'C'` for Clubs, `'S'` for Spades, `'H'` for Hearts, and `'D'` for Diamonds

We have provided a simple `__repr__` method.

For example:

```
>>> Card('Q', 'C')
QC
>>> Card("7","H")
7H
```

__init__

Locate and modify the constructor that we've provided for `Card` objects. You should make the constructor flexible, such that it can account for multiple different input formats and store the corresponding rank and suit values in the same internal format. Expand on the approach supplied to accommodate upper and lower cases letters, words.

For example, the following four cards (`c1`, `c2`, `c3`, `c4`) should all store the rank and suit in the the same internal format:

```
>>> c1 = Card("Queen", 'Clubs')
>>> c2 = Card("queen", 'club')
>>> c3 = Card("Q", 'C')
>>> c4 = Card('q', 'c')
```

get_value

Write a function called `get_value()` which returns the value of the called Card object. The returned value should be of type int. In Blackjack, the value of each card is derived from the rank of the card. Numbered cards have value equal to their number, face cards (Jack/Queen/King) should return a value of 10, and, although it can count as either 11 or 1 in Blackjack, the Ace should return a value of 1 for now.

For example:

```
>>> c1 = Card(4, 'C')
>>> c1.get_value()
4
>>> c2 = Card('queen', 'diamonds')
>>> c2.get_value()
10
>>> c3 = Card('A', 'S')
>>> c3.get_value()
1
```

Hand Class

Next, we'll consider the second object that is useful when implementing a card game, one representing a player's hand.

__init__

Locate the class called `Hand`, which will serve as a blueprint for objects that represent a single player's cards. Complete the constructor, use self.cards to store a list of cards in a Hand instance. You can make use of the .add_card() method (below) to handle Construction of hands with a card or list of cards.

add_card

Add a method called `add_card(self, card)` which adds a card or list cards to the cards attribute of the Hand object.

Work through the following example to check your functionality:

- Create a `Card` object for an 7 of Hearts and assign it to variable `c1`
- Create a `Card` object for an Ace of Diamonds and assign it to variable `c2`
- Create a `Hand` object and assign it to variable `h1`

- Add the `c1` and `h2` card objects to the `h1` object.

Now when we print `h1`, we should get both `c1` and `c2` (`__repr__` is already provided):

```
>>> c1 = Card("7","H")
>>> c2 = Card("A","D")
>>> c3 = Card("2","D")
>>> h = Hand()
>>> h.add_card(c1)
>>> h.add_card([c2, c3])
>>> h.add_card(c2)
>>>> h
7H, AD, 2D, AD
>>> h1 = Hand([c1, c2])
>>> h1
7H, AD
```

num_cards

Add a method called `num_cards()` to the `Hand` class that returns the number of cards in the calling `Hand` object (`self`). The returned value should be an integer.

For example:

```
>>> print('number of cards:', h1.num_cards())
number of cards: 2
```

get_value

Add a method called `get_value()` to the `Hand` class that returns the total value of the cards in the calling `Hand` object (`self`). The returned value should be an integer.

Hint: Use the `Card.get_value()` method to determine the individual value of each `Card` object in the `Hand`.

For example:

```
>>> print('total value:', h1.get_value())
total value: 8
```

has_any

Add a method called `has_any()` to the `Hand` class that takes a card rank as its only input, and returns `True` if there is at least one occurrence of a `Card` with that rank in the called `Hand` object (`self`), and `False` otherwise.

Note: In this method the suit of the `Card` objects do not matter; only the rank.

For example, using `h1` as defined above:

```
>>> print('has at least one 7:', h1.has_any(7))
has at least one 7: True
>>> print('has at least one Ace:', h1.has_any('A'))
has at least one Ace: True
>>> print('has at least one Queen:', h1.has_any('Q'))
has at least one Queen: False
```

Phase 2: Blackjack Setup `[setup2.py]`

Blackjack hand

Our existing `Hand` class has most of the functionality that we need to work with hands of cards, but we need the the value of a hand to be computed differently within the context of a game of Blackjack. Rather than redefining all our methods in a completely new class, we'll take advantage of inheritance to define a subclass of the `Hand` class.

Locate the class `BlackjackHand`. It is a subclass of the `Hand` class and inherits all variables and methods from its `Hand` parent class.

In a blackjack, an ace in a hand count can count as 1 or 11. The rule for doing so is passed on the following logic: It counts as an 11 unless doing so would lead the hand to have a total value that is greater than 21, in which case it should count as a 1.

get_value

Write a new method called `get_value()` to implement blackjack scoring in the BlackjackHand class. This method will override the Hand.get_value() method.

Note: J,Q,K are all have a value of 10.

Here is some example output:

```
>>> h2 = BlackjackHand()
>>> h2.add_card(c1)
>>> h2.add_card(c2)
>>> print('total value:', h2.get_value())
total value: 18
>>> c3 = Card(4, 'C')
>>> h2.add_card(c3)
>>> print('number of cards:', h2.num_cards())
number of cards: 3
>>> print('total value:', h2.get_value())
total value: 12
```

Notice how the second hand (which starts out with the same cards as as the first hand), originally has a value of 18, since the Ace is counted as 11. However, once we add a third card to it, its value becomes 12, since counting the Ace as 11 would bring the total value of the hand over 21, so we count the Ace as 1 instead. What should the value of a hand with two Aces be? What should the value of three Aces be? How about 4?

soft_value

In Blackjack, hands are classified as “soft” or “hard” depending on whether or not their current total value utilizes an Ace with a value of 11. (More information on this is given at the end of the Lecture 20 notes.) Write a .soft_value() method that returns True when a hand is “soft” and False when it is not.

Card Shoe

When playing blackjack at a casino the multiple decks are used and kept in a card shoe that holds some number of decks. As an exercise we complete a few methods for this class below.

__init__

Locate the class called `Shoe`. The `Shoe` in a card game consists of `num_decks` of 52-card decks. **Fill in the definition** of the `cards` attribute in the `__init__` method.

If a list of cards is not supplied by the user, create one and the shuffle it. (Use supplied lists of cards allow for testing, so don’t shuffle them)

If creating the list of cards, first create a list with the the correct number of cards (num_decks*52) with the correct distribution of cards over rank and suit, then shuffle them using .shuffle.

shuffle

Now, **locate the method** called `shuffle`. This method should randomly shuffles the elements in self.cards. For now, you can use random.shuffle to do this. Note: When using an infinite deck there is no need to shuffle the cards. In fact, doing so will just slow your code down, so don't! See the implementation of .deal() to see why this is the case. **After the cards have been shuffled reset self.next to zero so that cards now being once again dealt from the front of the shoe.**

The remaining methods are already complete.

`.deal(n)` deals n cards from the front of the shoe.

`.num_dealt()`, `.num_remaining()` return the number of cards dealt and the number of cards remaining.

You have now completed the `Shoe` class.

Intermezzo: Interactive Blackjack Play [simulate.py]

The CS4 Blackjack Rules are as follows:

1. Before cards are dealt player places an initial bet amount.
2. 4 cards are dealt:
3. Two face up to player
4. One face up and one face done to dealer
5. Dealer "peeks" at hole card (face down card
6. 4 possible outcomes:
    - Dealer BJ, Player BJ => draw
    - Dealer BJ => lose bet amount
    - Player BJ => win BJP*(bet amount)
    - No 2-card BJs (natural BJs)
7. Player can now request additional cards for their hand.
    - hitting = asking for a card
    - standing = no more cards desired
    - Player hand value > 21 => lose bet amount (bust)
8. Dealer turns over face down card
    - Hits until her hand value >= 17
    - Dealer hand value > 21 => win bet amount (dealer bust)
9. Finally,
    - Dealer value < Player value => win bet amount
    - Dealer value > Player value => lose bet amount
    - Dealer value == Player value => draw (tie)

Note: These rules do not implement insurance, doubling, splitting, or surrendering. Eliminating them makes the AI program tasks to follow much easier to implement while still providing for an interesting and challenging game environment.

Here is a link to a full set of Blackjack rules for reference:
https://wizardofodds.com/games/blackjack/basics/

We have given you a **Game class** and **play_round function** that lets one simulate the play of many rounds of CS4 Blackjack. Central to the Game and play_round is the use a Dealer and Player objects.

Interactive Play

In order to get started, try running simulate.py. As you play rounds of blackjack, locate the relevant methods that are used in the **Player class** and **Dealer class**, while also looking at the **play_round function** and **Game glass** code a documentation. Try using the debugger to step through the play_round function so that you understand it fully.

For now, the two most important Player methods to focus on are **.bet()** and **.next_move()** since they are used to determine a players bet (placed at the beginning of each round) what hit-stand strategy the payer will employ.

For now, the **.new_shoe()** method is not implemented, but later, when building card counting strategies, you will need to add appropriate implementations for it.

The Player class in simulate.py implements an interactive version of the game which prompts the uses for these their bet and hit-stand decisions. AI players will be subclasses of this Player class, will provide the same functionality automatically.

Below is an Game session:

```
Welcome to Interactive Blackjack
Minimum bet =  2
BJ round results are reported as follow:
('round #', 'player bet amount', 'player gain factor', 'type of outcome',
 'deck shuffled', 'Expected Gain', 'ending bankroll')
bet: amount = 2
[Dealer: 2H](2)[Player: 3H, 4H](7)
Would you like to hit, stand or quit? (h/s/q) >? h
[Dealer: 2H](2)[Player: 3H, 4H, 5H](12)
Would you like to hit, stand or quit? (h/s/q) >? h
[Dealer: 2H](2)[Player: 3H, 4H, 5H, 6H](18)
Would you like to hit, stand or quit? (h/s/q) >? h
[Dealer: 2H](2)[Player: 3H, 4H, 5H, 6H, 7H](25)
(1, 2, -1, 'P_Bust', False, -1.0, 8)
bet: amount = 2
```

```
[Dealer: 9H](9)[Player: 10H, JH](20)
Would you like to hit, stand or quit? (h/s/q) >? s
[Dealer: 9H, 8H](17)[Player: 10H, JH](20)
(2, 2, 1, 'P_Won', False, 0.0, 10)
bet: amount = 2
[Dealer: KH](10)[Player: AC, 2C](13)
Would you like to hit, stand or quit? (h/s/q) >? h
[Dealer: KH](10)[Player: AC, 2C, 3C](16)
Would you like to hit, stand or quit? (h/s/q) >? h
[Dealer: KH](10)[Player: AC, 2C, 3C, 4C](20)
Would you like to hit, stand or quit? (h/s/q) >? s
[Dealer: KH, QH](20)[Player: AC, 2C, 3C, 4C](20)
(3, 2, 0, 'PD_Draw', False, 0.0, 10)
bet: amount = 2
[Dealer: 6C](6)[Player: 7C, 8C](15)
Would you like to hit, stand or quit? (h/s/q) >? s
[Dealer: 6C, 5C](11)[Player: 7C, 8C](15)
[Dealer: 6C, 5C, 9C](20)[Player: 7C, 8C](15)
(4, 2, -1, 'P_Lost', True, -0.25, 8)
bet: amount = 2
[Dealer: 3D](3)[Player: 6C, 3S](9)
Would you like to hit, stand or quit? (h/s/q) >? h
[Dealer: 3D](3)[Player: 6C, 3S, 9D](18)
Would you like to hit, stand or quit? (h/s/q) >? s
[Dealer: 3D, 8H](11)[Player: 6C, 3S, 9D](18)
[Dealer: 3D, 8H, 7S](18)[Player: 6C, 3S, 9D](18)
(5, 2, 0, 'PD_Draw', False, -0.2, 8)
bet: amount = 2
[Dealer: 5C](5)[Player: AD, 8H](19)
Would you like to hit, stand or quit? (h/s/q) >? s
[Dealer: 5C, KS](15)[Player: AD, 8H](19)
[Dealer: 5C, KS, 8C](23)[Player: AD, 8H](19)
(6, 2, 1, 'D_Bust', False, 0.0, 10)
bet: amount = 2
[Dealer: 6D](6)[Player: 2H, 5S](7)
Would you like to hit, stand or quit? (h/s/q) >? h
[Dealer: 6D](6)[Player: 2H, 5S, 4D](11)
Would you like to hit, stand or quit? (h/s/q) >? h
[Dealer: 6D](6)[Player: 2H, 5S, 4D, KH](21)
[Dealer: 6D, 6H](12)[Player: 2H, 5S, 4D, KH](21)
[Dealer: 6D, 6H, 4H](16)[Player: 2H, 5S, 4D, KH](21)
[Dealer: 6D, 6H, 4H, 5C](21)[Player: 2H, 5S, 4D, KH](21)
(7, 2, 0, 'PD_Draw', False, 0.0, 10)
bet: amount = 2
[Dealer: 10S](10)[Player: 7H, 8S](15)
Would you like to hit, stand or quit? (h/s/q) >? h
[Dealer: 10S](10)[Player: 7H, 8S, 7C](22)
(8, 2, -1, 'P_Bust', True, -0.125, 8)
bet: amount = 2
[Dealer: QD](10)[Player: JC, KD](20)
```

```
Would you like to hit, stand or quit? (h/s/q) >? s
[Dealer: QD, JD](20)[Player: JC, KD](20)
(9, 2, 0, 'PD_Draw', False, -0.1111111111111111, 8)
bet: amount = 2
[Dealer: JC](10)[Player: 5H, 9S](14)
Would you like to hit, stand or quit? (h/s/q) >? h
[Dealer: JC](10)[Player: 5H, 9S, 10C](24)
(10, 2, -1, 'P_Bust', False, -0.2, 6)

Total Rounds 9 [Wins: 1 Losses: 4 Ties: 4]
tot_bet = 20.0 tot_won = 16.0 (won-bet)=-4.0
E(won-bet) = -0.4 (won-bet)/tb = -0.2 EG = -0.2
EG 95% CI = (-0.7642811372008942, 0.3642811372008941)
```

A player or dealer blackjack occurs when the first cards they are dealt have a value of 21, in this case, for CS4 Blackjack, when a player wins, the player blackjack payout (gain factor) is 1.95. (All other gain factors in the gameare -1, 0, or 1). Below is an example of a player blackjack (denoted P_BJ).

```
bet: amount = 2
[Dealer: QH](10)[Player: KH, AC](21)
[Dealer: QH, JH](20)[Player: KH, AC](21)
(3, 2, 1.95, 'P_BJ', False, 0.65, 23.9)
```

Most casinos offer a (player) blackjack payout of 3 to 2 (i.e., a 1.5 gain factor). Since CS4 Blackjack is a simplified version of blackjack a larger payout is required to keep the CS4 Casino's edge close to what is at regular casinos (i.e. less than 0.5%). Such a small edge allows for card counting strategies to work.

Phase 3: Simple AI Strategies [ai_simple3.{py,ipynb}]

```
class AIPlayerMimicDealer(Player):
    """
    AIPlayer that hits until their hand is 17 or above.

    Notebook Task: In your notebook compare the performance between AIMimicDealer
    and AIPlayerBadLuck.  Is there a clear winner? Support your result by looking
    at each strategies Expected Gains (EG), EG Confidence Intervals. Use a mean
    comparison test(s).  Provide example equity curves.

    Optional: State what each strategies Edge is, and what the performance
    of the better one will be relative to the worse one.
    """
```

```
class AIPlayerS17(Player):
    """
    AIPlayer that hits until their hand is 17 or higher, but also hits
    on a soft 17.  Bets twice the minimum per round (if possible).

    Notebook Task: Perform the same analysis between AIPlayerS17
    and AIPlayerMimicDealer as you did between AIMimicDealer and AIPlayerBadLuck.

    Is one of the strategies clearly better than the other?
    """
```

Phase 4: Perfect AI Strategy [ai_perfect.{py, ipynb}

This CS4 Blackjack Spreadsheet computes probabilities and expected gains for all possible hit or stand scenarios. The sheet named “hit or stand” contains the optimal memoryless infinite deck strategies.

```
class AIPlayerPerfectNoMemory(Player):
    """
    Represents a perfect (non-card counting, non-cheating) Player
    in a game of Blackjack. It implements the optimal strategy for an
    infinite deck.

    Notebook Task: Plot example equity curves.  Compare the measured EG
    with the theoretical EG.  Do the confidence intervals seem
    reasonable?  Use an appropriate statistical test to EG and the
    theoretical EG.

    Optional: Compare AIPlayerPerfectNoMemory with the best of the previous
    strategies, what is the difference in their edges?
    """
```

Phase 5: Clairvoyant AI Strategy [ai_clairvoyant.{py, ipynb}

```
class AIPlayerClairvoyant(AIPlayerPerfectNoMemory):
    """
    Implements a player with limited foresight.  Specifically, at the beginning of a
    round, the player can see the next two cards that will be dealt from the shoe, i.e.,
```

```
    the dealer face_down and face_up cards.  The player adjusts it's bet to be either
    min_bet or self.other_bet*self.min_bet (for other_bet>=1) and other_bet*self.bankroll
    (for other_bet < 1) (when possible).

    Notebook Task: Fine a good value for other_bet.  Explain why and how you selected
    other_bet.  Use equity curves and statistics as appropriate.

    Optional: Compare AIPlayerClairvoyant with AIPlayerXray, using non-fractional and
    fractional betting strategies.   What is the difference in their edges?

    Extra Credit:  Design a version of AIPlayerClairvoyant called AIPlayerSeer that looks at
    all possible outcomes for the current round, and then bets and acts accordingly. Can you
    see how to extend your solution to n possible future rounds? Add an AIPlayerSeer section
    to your notebook if you do this task.

    Optional If you implement AIPlayerSeer, do a notebook comparison of it with
    AIPlayerClairvoyant.  What is the difference in edges?
    """
```

Phase 6: HiLo Card Counting AI Strategy [ai_hilo.{py, ipynb}

```
class AIPlayerHiLo(AIPlayerPerfectNoMemory):
    """Implements a High-Low card Counting AI.  Uses the "True Count" to adjust bet size.

    Notebook Task: Find a good value for bet_by_count.  Explain why and how you arrived
    at the values you selected.  Use equity curves and statistics as appropriate. Explain
    what happens when bet_by_count is too aggressive. Include some statistics or equity
    curves.

    Optional: compare AIPlayerHiLo with the best non-peeking strategy you create, state
    clearly what the edge is for your card counting approach *over* the best non-card
    counting approach.
    """
```

A good high-low card counting system description can be found here. It explains how to compute a running count and convert it to a “True Count”.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this document or have any concerns about this and any other CS4 document by posting on Piazza or filling out our anonymous feedback form.