## Part 1 - Objects

## Magic: The Lambda-ing

[Card Game](https://inst.eecs.berkeley.edu/~cs61a/fa18/lab/lab06/lab06.zip)

In the next part of this lab, we will be implementing a card game!

You can start the game by typing:

python3 cardgame.py

**This game doesn't work yet**. If we run this right now, the code will error, since we haven't implemented anything yet. When it's working, you can exit the game and return to the command line with Ctrl-C or Ctrl-D.

This game uses several different files.

* Code for all the questions in this lab can be found in classes.py.
* Some utility for the game can be found in cardgame.py, but you won't need to open or read this file. This file doesn't actually mutate any instances directly - instead, it calls methods of the different classes, maintaining a strict abstraction barrier.
* If you want to modify your game later to add your own custom cards and decks, you can look in cards.py to see all the standard cards and the default deck; here, you can add more cards and change what decks you and your opponent use. The cards were not created with balance in mind, so feel free to modify the stats and add/remove cards as desired.

**Rules of the Game** This game is a little involved, though not nearly as much as its namesake. Here's how it goes:

There are two players. Each player has a hand of cards and a deck, and at the start of each round, each player draws a card from their deck. If a player's deck is empty when they try to draw, they will automatically lose the game. Cards have a name, an attack stat, and a defense stat. Each round, each player chooses one card to play from their own hands. The card with the higher *power* wins the round. Each played card's power value is calculated as follows:

(player card's attack) - (opponent card's defense) / 2

For example, let's say Player 1 plays a card with 2000 ATK/1000 DEF and Player 2 plays a card with 1500 ATK/3000 DEF. Their cards' powers are calculated as:

P1: 2000 - 3000/2 = 2000 - 1500 = 500

P2: 1500 - 1000/2 = 1500 - 500 = 1000

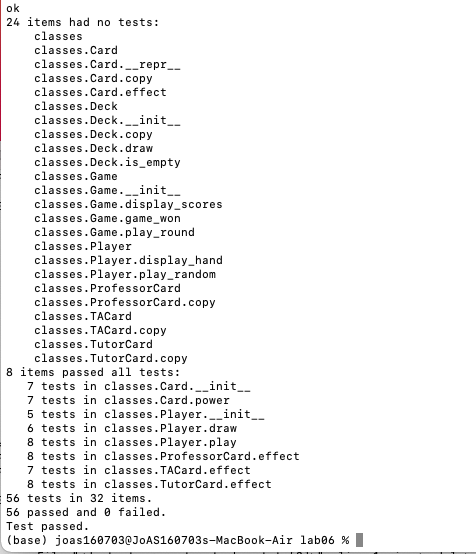
so Player 2 would win this round.

The first player to win 8 rounds wins the match!

However, there are a few effects we can add (in the optional questions section) to make this game a bit more interesting. Cards are split into Tutor, TA, and Professor types, and each type has a different *effect* when they're played. All effects are applied before power is calculated during that round:

* A Tutor will cause the opponent to discard and re-draw the first 3 cards in their hand.
* A TA will swap the opponent card's attack and defense.
* A Professor adds the opponent card's attack and defense to all cards in their deck and then remove all cards in the opponent's deck that share its attack *or* defense!

These are a lot of rules to remember, so refer back here if you need to review them, and let's start making the game!



### Q1: Making Cards

To play a card game, we're going to need to have cards, so let's make some! We're gonna implement the basics of the Card class first.

First, implement the Card class constructor in classes.py. This constructor takes three arguments:

* the name of the card, a string
* the attack stat of the card, an integer
* the defense stat of the card, an integer

Each Card instance should keep track of these values using instance attributes called name, attack, and defense.

You should also implement the power method in Card, which takes in another card as an input and calculates the current card's power. Check the Rules section if you want a refresher on how power is calculated.

|  |
| --- |
| **class** **Card**(object):  cardtype = 'Staff'  **def** **\_\_init\_\_**(self, name, attack, defense):  """  Create a Card object with a name, attack,  and defense.  >>> staff\_member = Card('staff', 400, 300)  >>> staff\_member.name  'staff'  >>> staff\_member.attack  400  >>> staff\_member.defense  300  >>> other\_staff = Card('other', 300, 500)  >>> other\_staff.attack  300  >>> other\_staff.defense  500  """  "\*\*\* YOUR CODE HERE \*\*\*"  **def** **power**(self, other\_card):  """  Calculate power as:  (player card's attack) - (opponent card's defense)/2  where other\_card is the opponent's card.  >>> staff\_member = Card('staff', 400, 300)  >>> other\_staff = Card('other', 300, 500)  >>> staff\_member.power(other\_staff)  150.0  >>> other\_staff.power(staff\_member)  150.0  >>> third\_card = Card('third', 200, 400)  >>> staff\_member.power(third\_card)  200.0  >>> third\_card.power(staff\_member)  50.0  """  "\*\*\* YOUR CODE HERE \*\*\*" |

### Q2: Making a Player

Now that we have cards, we can make a deck, but we still need players to actually use them. We'll now fill in the implementation of the Player class.

A Player instance has three instance attributes:

* name is the player's name. When you play the game, you can enter your name, which will be converted into a string to be passed to the constructor.
* deck is an instance of the Deck class. You can draw from it using its .draw() method.
* hand is a list of Card instances. Each player should start with 5 cards in their hand, drawn from their deck. Each card in the hand can be selected by its index in the list during the game. When a player draws a new card from the deck, it is added to the end of this list.

Complete the implementation of the constructor for Player so that self.hand is set to a list of 5 cards drawn from the player's deck.

Next, implement the draw and play methods in the Player class. The draw method draws a card from the deck and adds it to the player's hand. The play method removes and returns a card from the player's hand at the given index.

Call deck.draw() when implementing Player.\_\_init\_\_ and Player.draw. Don't worry about how this function works - leave it all to the abstraction!

|  |
| --- |
| **class** **Player**(object):  **def** **\_\_init\_\_**(self, deck, name):  """Initialize a Player object.  A Player starts the game by drawing 5 cards from their deck. Each turn,  a Player draws another card from the deck and chooses one to play.  >>> test\_card = Card('test', 100, 100)  >>> test\_deck = Deck([test\_card.copy() for \_ in range(6)])  >>> test\_player = Player(test\_deck, 'tester')  >>> len(test\_deck.cards)  1  >>> len(test\_player.hand)  5  """  self.deck = deck  self.name = name  "\*\*\* YOUR CODE HERE \*\*\*"  **def** **draw**(self):  """Draw a card from the player's deck and add it to their hand.  >>> test\_card = Card('test', 100, 100)  >>> test\_deck = Deck([test\_card.copy() for \_ in range(6)])  >>> test\_player = Player(test\_deck, 'tester')  >>> test\_player.draw()  >>> len(test\_deck.cards)  0  >>> len(test\_player.hand)  6  """  **assert** **not** self.deck.is\_empty(), 'Deck is empty!'  "\*\*\* YOUR CODE HERE \*\*\*"  **def** **play**(self, card\_index):  """Remove and return a card from the player's hand at the given index.  >>> from cards import \*  >>> test\_player = Player(standard\_deck, 'tester')  >>> ta1, ta2 = TACard("ta\_1", 300, 400), TACard("ta\_2", 500, 600)  >>> tutor1, tutor2 = TutorCard("t1", 200, 500), TutorCard("t2", 600, 400)  >>> test\_player.hand = [ta1, ta2, tutor1, tutor2]  >>> test\_player.play(0) is ta1  True  >>> test\_player.play(2) is tutor2  True  >>> len(test\_player.hand)  2  """  "\*\*\* YOUR CODE HERE \*\*\*" |

# **Part 2 - Inheritance**

For the following sections, do **not** overwrite any lines already provided in the code. Additionally, make sure to uncomment any calls to print once you have implemented each method. These are used to display information to the user, and changing them may cause you to fail tests that you would otherwise pass.

### Q3: Tutors: Flummox

To really make this card game interesting, our cards should have effects! We'll do this with the effect function for cards, which takes in the opponent card, the current player, and the opponent player.

Implement the effect method for Tutors, which causes the opponent to discard the first 3 cards in their hand and then draw 3 new cards. Assume there at least 3 cards in the opponent's hand and at least 3 cards in the opponent's deck.

Remember to uncomment the call to print once you're done!

|  |
| --- |
| **class** **TutorCard**(Card):  cardtype = 'Tutor'  **def** **effect**(self, other\_card, player, opponent):  """  Discard the first 3 cards in the opponent's hand and have  them draw the same number of cards from their deck.  >>> from cards import \*  >>> player1, player2 = Player(player\_deck, 'p1'), Player(opponent\_deck, 'p2')  >>> other\_card = Card('other', 500, 500)  >>> tutor\_test = TutorCard('Tutor', 500, 500)  >>> initial\_deck\_length = len(player2.deck.cards)  >>> tutor\_test.effect(other\_card, player1, player2)  p2 discarded and re-drew 3 cards!  >>> len(player2.hand)  5  >>> len(player2.deck.cards) == initial\_deck\_length - 3  True  """  "\*\*\* YOUR CODE HERE \*\*\*"  #Uncomment the line below when you've finished implementing this method!  #print('{} discarded and re-drew 3 cards!'.format(opponent.name)) |

### Q4: TAs: Shift

Let's add an effect for TAs now! Implement the effect method for TAs, which swaps the attack and defense of the opponent's card.

|  |
| --- |
| **class** **TACard**(Card):  cardtype = 'TA'  **def** **effect**(self, other\_card, player, opponent):  """  Swap the attack and defense of an opponent's card.  >>> from cards import \*  >>> player1, player2 = Player(player\_deck, 'p1'), Player(opponent\_deck, 'p2')  >>> other\_card = Card('other', 300, 600)  >>> ta\_test = TACard('TA', 500, 500)  >>> ta\_test.effect(other\_card, player1, player2)  >>> other\_card.attack  600  >>> other\_card.defense  300  """  "\*\*\* YOUR CODE HERE \*\*\*" |

### Q5: The Professor Arrives

A new challenger has appeared! Implement the effect method for the Professor, who adds the opponent card's attack and defense to all cards in the player's deck and then removes *all* cards in the opponent's deck that have the same attack or defense as the opponent's card.

*Note:* You might run into trouble when you mutate a list as you're iterating through it. Try iterating through a copy instead! You can use slicing to copy a list:

>>> lst = [1, 2, 3, 4]

>>> copy = lst[:]

>>> copy

[1, 2, 3, 4]

>>> copy is lst

False

|  |
| --- |
| **class** **ProfessorCard**(Card):  cardtype = 'Professor'  **def** **effect**(self, other\_card, player, opponent):  """  Adds the attack and defense of the opponent's card to  all cards in the player's deck, then removes all cards  in the opponent's deck that share an attack or defense  stat with the opponent's card.  >>> test\_card = Card('card', 300, 300)  >>> professor\_test = ProfessorCard('Professor', 500, 500)  >>> opponent\_card = test\_card.copy()  >>> test\_deck = Deck([test\_card.copy() for \_ in range(8)])  >>> player1, player2 = Player(test\_deck.copy(), 'p1'), Player(test\_deck.copy(), 'p2')  >>> professor\_test.effect(opponent\_card, player1, player2)  3 cards were discarded from p2's deck!  >>> [(card.attack, card.defense) for card in player1.deck.cards]  [(600, 600), (600, 600), (600, 600)]  >>> len(player2.deck.cards)  0  """  orig\_opponent\_deck\_length = len(opponent.deck.cards)  "\*\*\* YOUR CODE HERE \*\*\*"  discarded = orig\_opponent\_deck\_length - len(opponent.deck.cards)  **if** discarded:  #Uncomment the line below when you've finished implementing this method!  #print('{} cards were discarded from {}\'s deck!'.format(discarded, opponent.name))  **Return** |