Lab 7: Iterators, Generators, Object-Oriented Programming

> Adapted from cs61a of UC Berkeley.

Starter Files

Get your starter file by cloning the repository: https://github.com/JacyCui/sicp-lab07.git

```
git clone https://github.com/JacyCui/sicp-lab07.git
```

`lab07.zip` is the starter file you need, you might need to unzip the file to get the skeleton code.

```
unzip lab07.zip
```

`README.md` is the handout for this homework. `solution` is a probrab solution of the lab. However, I might not give my solution exactly when the lab is posted. You need to finish the task on your own first. If any problem occurs, please make use of the comment section.

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Iterators

An iterable is any object that can be iterated through, or gone through one element at a time. One construct that we've used to iterate through an iterable is a for loop:

```python
for elem in iterable:
    # do something
```

`for` loops work on any object that is *iterable*. We previously described it as working with any sequence -- all sequences are iterable, but there are other objects that are also iterable! We define an **iterable** as an object on which calling the built-in function `iter` function returns an *iterator*. An iterator is another type of object that allows us to iterate through an iterable by keeping track of which element is next in the sequence.

To illustrate this, consider the following block of code, which does the exact same thing as a the for statement above:

```python
iterator = iter(iterable)
try:
    while True:
        elem = next(iterator)
        # do something
except StopIteration:
    pass
```

Here's a breakdown of what's happening:

- First, the built-in `iter` function is called on the iterable to create a corresponding *iterator*.
- To get the next element in the sequence, the built-in `next` function is called on this iterator.
- When `next` is called but there are no elements left in the iterator, a `StopIteration` error is raised. In the for loop construct, this exception is caught and execution can continue.

Calling `iter` on an iterable multiple times returns a new iterator each time with distinct states (otherwise, you'd never be able to iterate through a iterable more than once). You can also call `iter` on the iterator itself, which will just return the same iterator without changing its state. However, note that you cannot call `next` directly on an iterable.

Let's see the `iter` and `next` functions in action with an iterable we're already familiar with -- a list.

```python
>>> lst = [1, 2, 3, 4]
>>> next(lst)              # Calling next on an iterable
TypeError: 'list' object is not an iterator
>>> list_iter = iter(lst) # Creates an iterator for the list
>>> list_iter
<list_iterator object ...>
>>> next(list_iter)        # Calling next on an iterator
1
>>> next(list_iter)        # Calling next on the same iterator
2
>>> next(iter(list_iter)) # Calling iter on an iterator returns itself
3
>>> list_iter2 = iter(lst)
>>> next(list_iter2)       # Second iterator has new state
1
>>> next(list_iter)        # First iterator is unaffected by second iterator
4
>>> next(list_iter)        # No elements left!
StopIteration
>>> lst                    # Original iterable is unaffected
[1, 2, 3, 4]
```

Since you can call `iter` on iterators, this tells us that that they are also iterables! Note that while all iterators are iterables, the converse is not true - that is, not all iterables are iterators. You can use iterators wherever you can use iterables, but note that since iterators keep their state, they're only good to iterate through an iterable once:

```
>>> list_iter = iter([4, 3, 2, 1])
>>> for e in list_iter:
...     print(e)
4
3
2
1
>>> for e in list_iter:
...     print(e)
```

> **Analogy**: An iterable is like a book (one can flip through the pages) and an iterator for a book would be a bookmark (saves the position and can locate the next page). Calling `iter` on a book gives you a new bookmark independent of other bookmarks, but calling `iter` on a bookmark gives you the bookmark itself, without changing its position at all. Calling `next` on the bookmark moves it to the next page, but does not change the pages in the book. Calling `next` on the book wouldn't make sense semantically. We can also have multiple bookmarks, all independent of each other.

Iterable Uses

We know that lists are one type of built-in iterable objects. You may have also encountered the `range(start, end)` function, which creates an iterable of ascending integers from start (inclusive) to end (exclusive).

```
>>> for x in range(2, 6):
...     print(x)
...
2
3
4
5
```

Ranges are useful for many things, including performing some operations for a particular number of iterations or iterating through the indices of a list.

There are also some built-in functions that take in iterables and return useful results:

- `map(f, iterable)` - Creates iterator over `f(x)` for each `x` in `iterable`
- `filter(f, iterable)` - Creates iterator over `x` for each `x` in `iterable` if `f(x)`
- `zip(iter1, iter2)` - Creates iterator over co-indexed pairs (x, y) from both input iterables
- `reversed(iterable)` - Creates iterator over all the elements in the input iterable in reverse order
- `list(iterable)` - Creates a list containing all the elements in the input iterable
- `tuple(iterable)` - Creates a tuple containing all the elements in the input iterable
- `sorted(iterable)` - Creates a sorted list containing all the elements in the input iterable

Generators

We can create our own custom iterators by writing a *generator function*, which returns a special type of iterator called a **generator**. Generator functions have `yield` statements within the body of the function instead of `return` statements. Calling a generator function will return a generator object and will *not* execute the body of the function.

For example, let's consider the following generator function:

```
def countdown(n):
    print("Beginning countdown!")
    while n >= 0:
        yield n
        n -= 1
    print("Blastoff!")
```

Calling `countdown(k)` will return a generator object that counts down from `k` to 0. Since generators are iterators, we can call `iter` on the resulting object, which will simply return the same object. Note that the body is not executed at this point; nothing is printed and no numbers are output.

```
>>> c = countdown(5)
>>> c
<generator object countdown ...>
>>> c is iter(c)
True
```

So how is the counting done? Again, since generators are iterators, we call `next` on them to get the next element! The first time `next` is called, execution begins at the first line of the function body and continues until the `yield` statement is reached. The result of evaluating the expression in the `yield` statement is returned. The following interactive session continues from the one above.

```
>>> next(c)
Beginning countdown!
5
```

Unlike functions we've seen before in this course, generator functions can remember their state. On any consecutive calls to `next`, execution picks up from the line after the `yield` statement that was previously executed. Like the first call to `next`, execution will continue until the next `yield` statement is reached. Note that because of this, `Beginning countdown!` doesn't get printed again.

```
>>> next(c)
4
>>> next(c)
3
```

The next 3 calls to `next` will continue to yield consecutive descending integers until 0. On the following call, a `StopIteration` error will be raised because there are no more values to yield (i.e. the end of the function body was reached before hitting a `yield` statement).

```
>>> next(c)
2
>>> next(c)
1
>>> next(c)
0
>>> next(c)
Blastoff!
StopIteration
```

Separate calls to `countdown` will create distinct generator objects with their own state. Usually, generators shouldn't restart. If you'd like to reset the sequence, create another generator object by calling the generator function again.

```
>>> c1, c2 = countdown(5), countdown(5)
>>> c1 is c2
False
>>> next(c1)
5
>>> next(c2)
5
```

Here is a summary of the above:

- A *generator function* has a `yield` statement and returns a *generator object*.
- Calling the `iter` function on a generator object returns the same object without modifying its current state.
- The body of a generator function is not evaluated until `next` is called on a resulting generator object. Calling the `next` function on a generator object computes and returns the next object in its sequence. If the sequence is exhausted, `StopIteration` is raised.
- A generator "remembers" its state for the next `next` call. Therefore,
    - the first `next` call works like this:
        1. Enter the function and run until the line with `yield`.
        2. Return the value in the `yield` statement, but remember the state of the function for future `next` calls.
    - And subsequent `next` calls work like this:
        1. Re-enter the function, start at **the line after the `yield` statement that was previously executed**, and run until the next `yield` statement.
        2. Return the value in the `yield` statement, but remember the state of the function for future `next` calls.

- Calling a generator function returns a brand new generator object (like calling `iter` on an iterable object).
- A generator should not restart unless it's defined that way. To start over from the first element in a generator, just call the generator function again to create a new generator.

Another useful tool for generators is the `yield from` statement (introduced in Python 3.3). `yield from` will yield all values from an iterator or iterable.

```
>>> def gen_list(lst):
...     yield from lst
...
>>> g = gen_list([1, 2, 3, 4])
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
4
>>> next(g)
StopIteration
```

Object-Oriented Programming

Object-oriented programming (OOP) is a style of programming that allows you to think of code in terms of "objects." Here's an example of a `Car` class:

```
class Car(object):
    num_wheels = 4

    def __init__(self, color):
        self.wheels = Car.num_wheels
        self.color = color

    def drive(self):
        if self.wheels <= Car.num_wheels:
            return self.color + ' car cannot drive!'
        return self.color + ' car goes vroom!'

    def pop_tire(self):
        if self.wheels > 0:
            self.wheels -= 1
```

Here's some terminology:

- **class**: a blueprint for how to build a certain type of object. The `Car` class (shown above) describes the behavior and data that all `Car` objects have.

- **instance**: a particular occurrence of a class. In Python, we create instances of a class like this:

  ```
  >>> my_car = Car('red')
  ```

  `my_car` is an instance of the `Car` class.

- **attribute** or **field**: a variable that belongs to the class. Think of an attribute as a quality of the object: cars have *wheels* and *color*, so we have given our `Car` class `self.wheels` and `self.color` attributes. We can access attributes using **dot notation**:

  ```
  >>> my_car.color
  'red'
  >>> my_car.wheels
  4
  ```

- **method**: Methods are just like normal functions, except that they are tied to an instance or a class. Think of a method as a "verb" of the class: cars can *drive* and also *pop their tires*, so we have given our `Car` class the methods `drive` and `pop_tire`. We call methods using **dot notation**:

  ```
  >>> my_car = Car('red')
  >>> my_car.drive()
  'red car goes vroom!'
  ```

- **constructor**: As with data abstraction, constructors describe how to build an instance of the class. Most classes have a constructor. In Python, the constructor of the class defined as `__init__`. For example, here is the `Car` class's constructor:

  ```
  def __init__(self, color):
      self.wheels = Car.num_wheels
      self.color = color
  ```

  The constructor takes in one argument, `color`. As you can see, the constructor also creates the `self.wheels` and `self.color` attributes.

- `self`: in Python, `self` is the first parameter for many methods (in this class, we will only use methods whose first parameter is `self`). When a method is called, `self` is bound to an instance of the class. For example:

  ```
  >>> my_car = Car('red')
  >>> car.drive()
  ```

  Notice that the `drive` method takes in `self` as an argument, but it looks like we didn't pass one in! This is because the dot notation *implicitly* passes in `car` as `self` for us.

Required Questions

Iterators and Generators

Generators also allow us to represent infinite sequences, such as the sequence of natural numbers (1, 2, ...) shown in the function below!

```
def naturals():
    """A generator function that yields the infinite sequence of natural
    numbers, starting at 1.

    >>> m = naturals()
    >>> type(m)
    <class 'generator'>
    >>> [next(m) for _ in range(10)]
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    """
    i = 1
    while True:
        yield i
        i += 1
```

Q1: Scale

Implement the generator function `scale(it, multiplier)`, which yields elements of the given iterable `it`, scaled by `multiplier`. As an extra challenge, try writing this function using a `yield from` statement!

```
def scale(it, multiplier):
    """Yield elements of the iterable it scaled by a number multiplier.

    >>> m = scale([1, 5, 2], 5)
    >>> type(m)
    <class 'generator'>
    >>> list(m)
    [5, 25, 10]

    >>> m = scale(naturals(), 2)
    >>> [next(m) for _ in range(5)]
    [2, 4, 6, 8, 10]
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q scale --local
```

Q2: Hailstone

Write a generator that outputs the hailstone sequence from homework 1.

Here's a quick reminder of how the hailstone sequence is defined:

1. Pick a positive integer `n` as the start.
2. If `n` is even, divide it by 2.
3. If `n` is odd, multiply it by 3 and add 1.
4. Continue this process until `n` is 1.

Note: It is highly encouraged (though not required) to try writing a solution using recursion for some extra practice. Since `hailstone` returns a generator, you can `yield from` a call to `hailstone`!

```
def hailstone(n):
    """
    >>> for num in hailstone(10):
    ...     print(num)
    ...
    10
    5
    16
    8
    4
    2
    1
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q hailstone --local
```

WWPD: Objects

Q3: The Car class

Note: These questions use inheritance. For an overview of inheritance, see the inheritance portion of Composing Programs

Below is the definition of a `Car` class that we will be using in the following WWPD questions. **Note:** This definition can also be found in `car.py`.

```
class Car(object):
    num_wheels = 4
    gas = 30
    headlights = 2
```

```python
    size = 'Tiny'

    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.color = 'No color yet. You need to paint me.'
        self.wheels = Car.num_wheels
        self.gas = Car.gas

    def paint(self, color):
        self.color = color
        return self.make + ' ' + self.model + ' is now ' + color

    def drive(self):
        if self.wheels < Car.num_wheels or self.gas <= 0:
            return 'Cannot drive!'
        self.gas -= 10
        return self.make + ' ' + self.model + ' goes vroom!'

    def pop_tire(self):
        if self.wheels > 0:
            self.wheels -= 1

    def fill_gas(self):
        self.gas += 20
        return 'Gas level: ' + str(self.gas)
```

> Use Ok to test your knowledge with the following What would Python Display questions.
>
> ```
> python3 ok -q wwpd-car -u --local
> ```
>
> If an error occurs, type Error. If nothing is displayed, type Nothing.

```
>>> deneros_car = Car('Tesla', 'Model S')
>>> deneros_car.model

______
>>> deneros_car.gas = 10
>>> deneros_car.drive()

______
>>> deneros_car.drive()

______
>>> deneros_car.fill_gas()

______
>>> deneros_car.gas
```

```

______
>>> Car.gas

______
>>> deneros_car = Car('Tesla', 'Model S')
>>> deneros_car.wheels = 2
>>> deneros_car.wheels

______
>>> Car.num_wheels

______
>>> deneros_car.drive()

______
>>> Car.drive()

______
>>> Car.drive(deneros_car)

______
```

For the following, we reference the `MonsterTruck` class below. **Note**: The `MonsterTruck` class can not be found in `car.py`.

```
class MonsterTruck(Car):
    size = 'Monster'

    def rev(self):
        print('Vroom! This Monster Truck is huge!')

    def drive(self):
        self.rev()
        return Car.drive(self)
```

```
>>> deneros_car = MonsterTruck('Monster', 'Batmobile')
>>> deneros_car.drive()

______
>>> Car.drive(deneros_car)

______
>>> MonsterTruck.drive(deneros_car)

______
>>> Car.rev(deneros_car)

______
```

Magic: The Lambda-ing

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!

Q4: 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.

```python
class Card:
    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 ***"
```

Use Ok to test your code:

```
python3 ok -q Card.__init__ --local
python3 ok -q Card.power --local
```

Q5: 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:
    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 ***"
```

Use Ok to test your code:

```
python3 ok -q Player.__init__ --local
python3 ok -q Player.draw --local
python3 ok -q Player.play --local
```

After you complete this problem, you'll be able to play a working version of the game! Type

```
python3 cardgame.py
```

to start a game of Magic: The Lambda-ing!

This version doesn't have the effects for different cards, yet - to get those working, try out the optional questions below.

Optional Questions

The following code-writing questions will all be in `classes.py`.

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.

Q6: 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))
```

Use Ok to test your code:

```
python3 ok -q TutorCard.effect --local
```

Q7: 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 ***"
```

Use Ok to test your code:

```
python3 ok -q TACard.effect --local
```

Q8: 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
```

Use Ok to test your code:

```
python3 ok -q ProfessorCard.effect --local
```

After you complete this problem, we'll have a fully functional game of Magic: The Lambda-ing! This doesn't have to be the end, though - we encourage you to get creative with more card types, effects, and even adding more custom cards to your deck!

Congratulations! You've finished all problems of the lab. Feel free to run doctest to verify your answer again.

```
python3 -m doctest lab07.py
python3 -m doctest classes.py
```