Lab 7 Solutions lab07.zip (lab07.zip)

Solution Files

Required Questions

What Would Python Display?

Q1: WWPD: Linked Lists

Read over the `Link` class in `lab07.py`. Make sure you understand the doctests.

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python3 ok -q link -u
```

Enter `Function` if you believe the answer is `<function ...>`, `Error` if it errors, and `Nothing` if nothing is displayed.

If you get stuck, try drawing out the box-and-pointer diagram for the linked list on a piece of paper or loading the `Link` class into the interpreter with `python3 -i lab07.py`.

```
>>> from lab07 import *
>>> link = Link(1000)
>>> link.first
______

>>> link.rest is Link.empty
______

>>> link = Link(1000, 2000)
______

>>> link = Link(1000, Link())
______
```

```
>>> from lab07 import *
>>> link = Link(1, Link(2, Link(3)))
>>> link.first
------

>>> link.rest.first
------

>>> link.rest.rest.rest is Link.empty
------

>>> link.first = 9001
>>> link.first
------

>>> link.rest = link.rest.rest
>>> link.rest.first
------

>>> link = Link(1)
>>> link.rest = link
>>> link.rest.rest.rest.rest.first
------

>>> link = Link(2, Link(3, Link(4)))
>>> link2 = Link(1, link)
>>> link2.first
------

>>> link2.rest.first
------
```

```
>>> from lab07 import *
>>> link = Link(5, Link(6, Link(7)))
>>> link                  # Look at the __repr__ method of Link
------

>>> print(link)          # Look at the __str__ method of Link
------
```

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!

Q2: 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
        """
        self.name = name
        self.attack = attack
        self.defense = defense

    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
        """
        return self.attack - other_card.defense / 2
```

Use Ok to test your code:

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

Q3: 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
        self.hand = [deck.draw() for _ in range(5)]

    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!'
        self.hand.append(self.deck.draw())

    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, 4
        >>> 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
        """
```

```
        return self.hand.pop(card_index)
```

Use Ok to test your code:

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

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.

Linked Lists

For the remainder of the lab, code will be in `lab07.py` .

Q4: Link to List

Write a function `link_to_list` that takes in a linked list and returns the sequence as a Python list. You may assume that the input list is shallow; none of the elements is another linked list.

Try to find both an iterative and recursive solution for this problem!

```
def link_to_list(link):
    """Takes a linked list and returns a Python list with the same elements.

    >>> link = Link(1, Link(2, Link(3, Link(4))))
    >>> link_to_list(link)
    [1, 2, 3, 4]
    >>> link_to_list(Link.empty)
    []
    """
    # Recursive solution
    if link is Link.empty:
        return []
    return [link.first] + link_to_list(link.rest)

# Iterative solution
def link_to_list_iterative(link):
    result = []
    while link is not Link.empty:
        result.append(link.first)
        link = link.rest
    return result

    # Video Walkthrough: https://youtu.be/hdO9Ry8d5FU?t=25m6s
```

Use Ok to test your code:

```
python3 ok -q link_to_list
```

Trees

Q5: Cumulative Mul

Write a function `cumulative_mul` that mutates the Tree `t` so that each node's label becomes the product of all labels in the subtree rooted at the node.

```
def cumulative_mul(t):
    """Mutates t so that each node's label becomes the product of all labels in
    the corresponding subtree rooted at t.

    >>> t = Tree(1, [Tree(3, [Tree(5)]), Tree(7)])
    >>> cumulative_mul(t)
    >>> t
    Tree(105, [Tree(15, [Tree(5)]), Tree(7)])
    """
    for b in t.branches:
        cumulative_mul(b)
    total = t.label
    for b in t.branches:
        total *= b.label
    t.label = total
```

Use Ok to test your code:

```
python3 ok -q cumulative_mul
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

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

Note: These questions use inheritance, which has not been covered in lecture yet. It will be covered on Wednesday, 10/9.

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,
        >>> 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
        """
        opponent.hand = opponent.hand[3:]
        for _ in range(3):
            opponent.draw()
        #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
```

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,
        >>> 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
        """
        other_card.attack, other_card.defense = other_card.defense, other_card.a
```

Use Ok to test your code:

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

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.
        >>> 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)
        for card in player.deck.cards:
            card.attack += other_card.attack
            card.defense += other_card.defense
        for card in opponent.deck.cards[:]:
            if card.attack == other_card.attack or card.defense == other_card.de
                opponent.deck.cards.remove(card)
        discarded = orig_opponent_deck_length - len(opponent.deck.cards)
        if discarded:
            #Uncomment the line below when you've finished implementing this met
            #print('{} cards were discarded from {}\'s deck!'.format(discarded,
            return
```

Use Ok to test your code:

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

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!

Q9: Nonlocal Environment Diagram

Draw the environment diagram that results from running the following code.

```
def moon(f):
    sun = 0
    moon = [sun]
    def run(x):
        nonlocal sun, moon
        def sun(sun):
            return [sun]
        y = f(x)
        moon.append(sun(y))
        return moon[0] and moon[1]
    return run

moon(lambda x: moon)(1)
```

After you've done it on your own, generate an environment diagram in python tutor (http://tutor.cs61a.org) to check your answer.