Lab 7 Solutions lab07.zip (lab07.zip)

Solution Files

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.

Required Questions

Cars

These questions use inheritance. For an overview of inheritance, see the inheritance portion (http://composingprograms.com/pages/25-object-oriented-programming.html#inheritance) of Composing Programs.

Q1: Classy Cars

Below is the definition of a `Car` class that we will be using in the following WWPD questions.

> **Note:** The `Car` class definition can also be found in `car.py`.

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

For the later unlocking questions, we will be referencing the `MonsterTruck` class below.

> **Note**: The `MonsterTruck` class definition can also 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)
```

You can find the unlocking questions below.

> Use Ok to test your knowledge with the following "What Would Python Display?" questions:
>
> ```
> python3 ok -q wwpd-car -u
> ```
>
> **Important:** For all WWPD questions, type `Function` if you believe the answer is `<function...>`, `Error` if it errors, and `Nothing` if nothing is displayed.

```
>>> 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)
------
```

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

>>> Car.drive(deneros_car)
------

>>> MonsterTruck.drive(deneros_car)
------

>>> Car.rev(deneros_car)
------
```

Accounts

Let's say we'd like to model a bank account that can handle interactions such as depositing funds or gaining interest on current funds. In the following questions, we will be building off of the `Account` class. Here's our current definition of the class:

```
class Account:
    """An account has a balance and a holder.
    >>> a = Account('John')
    >>> a.deposit(10)
    10
    >>> a.balance
    10
    >>> a.interest
    0.02
    >>> a.time_to_retire(10.25) # 10 -> 10.2 -> 10.404
    2
    >>> a.balance               # balance should not change
    10
    >>> a.time_to_retire(11)    # 10 -> 10.2 -> ... -> 11.040808032
    5
    >>> a.time_to_retire(100)
    117
    """
    max_withdrawal = 10
    interest = 0.02

    def __init__(self, account_holder):
        self.balance = 0
        self.holder = account_holder

    def deposit(self, amount):
        self.balance = self.balance + amount
        return self.balance

    def withdraw(self, amount):
        if amount > self.balance:
            return "Insufficient funds"
        if amount > self.max_withdrawal:
            return "Can't withdraw that amount"
        self.balance = self.balance - amount
        return self.balance
```

Q2: Retirement

Add a `time_to_retire` method to the `Account` class. This method takes in an `amount` and returns how many years the holder would need to wait in order for the current `balance` to grow to at least `amount`, assuming that the bank adds `balance` times the `interest` rate to the total balance at the end of every year.

```
    def time_to_retire(self, amount):
        """Return the number of years until balance would grow to amount."""
        assert self.balance > 0 and amount > 0 and self.interest > 0
        future = self.balance
        years = 0
        while future < amount:
            future += self.interest * future
            years += 1
        return years
```

Use Ok to test your code:

```
python3 ok -q Account
```

We take of our current balance, and simulate the growth from interest over many years. We stop once we hit the target value.

Note that the problem solving procedure does not differ very much from an non OOP problem. The main difference here is make sure that we do not change the account balance while in the process of calculating the future balance. Therefore, something along these lines is necessary:

```
future = self.balance
```

Video walkthrough:

YouTube link (https://youtu.be/fQzeZcI-4a0)

Q3: FreeChecking

Implement the `FreeChecking` class, which is like the `Account` class from lecture except that it charges a withdraw fee after 2 withdrawals. If a withdrawal is unsuccessful, it still counts towards the number of free withdrawals remaining, but no fee for the withdrawal will be charged.

```
class FreeChecking(Account):
    """A bank account that charges for withdrawals, but the first two are free!
    >>> ch = FreeChecking('Jack')
    >>> ch.balance = 20
    >>> ch.withdraw(100)  # First one's free
    'Insufficient funds'
    >>> ch.withdraw(3)    # And the second
    17
    >>> ch.balance
    17
    >>> ch.withdraw(3)    # Ok, two free withdrawals is enough
    13
    >>> ch.withdraw(3)
    9
    >>> ch2 = FreeChecking('John')
    >>> ch2.balance = 10
    >>> ch2.withdraw(3) # No fee
    7
    >>> ch.withdraw(3)  # ch still charges a fee
    5
    >>> ch.withdraw(5)  # Not enough to cover fee + withdraw
    'Insufficient funds'
    """
    withdraw_fee = 1
    free_withdrawals = 2

    def __init__(self, account_holder):
        super().__init__(account_holder)
        self.withdrawals = 0

    def withdraw(self, amount):
        self.withdrawals += 1
        fee = 0
        if self.withdrawals > self.free_withdrawals:
            fee = self.withdraw_fee
        return super().withdraw(amount + fee)

    # Alternative solution where you don't need to include init.
    # Check out the video solution for more.
    def withdraw(self, amount):
        self.free_withdrawals -= 1
        if self.free_withdrawals >= 0:
            return super().withdraw(amount)
        return super().withdraw(amount + self.withdraw_fee)
```

Use Ok to test your code:

```
python3 ok -q FreeChecking
```

We can take advantage of inheritance to make sure we add just what we need to `withdraw`.

- For starters, a withdrawal with a fee is the same as the original withdraw amount plus the amount from the fee. We can therefore represent a `FreeChecking` withdraw as a "regular" `Account` withdraw in this way.
- On top of the note from before, we need to do a little bit of extra bookkeeping to make sure the first few withdrawals do not add the extra fee. We can either create a new instance attribute or modify an existing one.

Video walkthrough:

YouTube link (https://youtu.be/flIMJC2lY3M)

Magic: the Lambda-ing

In the next part of this lab, we will be implementing a card game! This game is inspired by the similarly named Magic: The Gathering (https://en.wikipedia.org/wiki/Magic:_The_Gathering).

Once you've implemented the game, you can start it by typing:

```
python3 cardgame.py
```

While playing the game, you can exit it 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. If you're familiar with the original game, you may notice the cards were not created with balance in mind, so feel free to modify the stats and add or 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 random 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 value, and a defense value. 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 attack and 1000 defense and Player 2 plays a card with 1500 attack and 3000 defense. 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 more interesting. A card can be of type AI, Tutor, TA, or Instructor, and each type has a different *effect* when they are played. All effects are applied before power is calculated during that round:

- An AI card will reduce the opponent card's attack by the opponent card's defense, and then double the opponent card's defense.
- A Tutor card will cause the opponent to discard and re-draw the first 3 cards in their hand.
- A TA card will swap the opponent card's attack and defense.
- An Instructor card will add 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!

Feel free to refer back to these series of rules later on, 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:

- a string as the `name` of the card
- an integer as the `attack` value of the card
- an integer as the `defense` value of the card

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. Refer to the Rules of the Game if you'd like 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, opponent_card):
        """
        Calculate power as:
        (player card's attack) - (opponent card's defense)/2
        >>> 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 - opponent_card.defense / 2
```

Use Ok to test your code:

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

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
        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, 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
        """
```

```
        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, you can implement the optional questions below.

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

To make the card game more interesting, let's add effects to our cards! We can do this by implementing an `effect` function for each card class, which takes in the opponent card, the current player, and the opponent player.

You can find the following questions in `classes.py`.

> **Important:** For the following sections, do **not** overwrite any lines already provided in the code.

Q6: AIs: Defenders

Implement the `effect` method for AIs, which reduces the opponent card's attack by the opponent card's defense, and then doubles the opponent card's defense.

> *Note:* The opponent card's resulting attack value cannot be negative.

```
class AICard(Card):
    cardtype = 'AI'

    def effect(self, opponent_card, player, opponent):
        """
        Reduce the opponent's card's attack by its defense,
        then double its defense.
        >>> from cards import *
        >>> player1, player2 = Player(player_deck, 'p1'), Player(opponent_deck, 'p2')
        >>> opponent_card = Card('other', 300, 600)
        >>> ai_test = AICard('AI', 500, 500)
        >>> ai_test.effect(opponent_card, player1, player2)
        >>> opponent_card.attack
        0
        >>> opponent_card.defense
        1200
        >>> opponent_card = Card('other', 600, 400)
        >>> ai_test = AICard('AI', 500, 500)
        >>> ai_test.effect(opponent_card, player1, player2)
        >>> opponent_card.attack
        200
        >>> opponent_card.defense
        800
        """
        opponent_card.attack = max(0, opponent_card.attack - opponent_card.defense)
        opponent_card.defense *= 2
```

Use Ok to test your code:

```
python3 ok -q AICard.effect
```

Q7: Tutors: Flummox

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. You can assume that at least 3 cards in the opponent's hand and at least 3 cards in the opponent's deck.

```
class TutorCard(Card):
    cardtype = 'Tutor'

    def effect(self, opponent_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')
        >>> opponent_card = Card('other', 500, 500)
        >>> tutor_test = TutorCard('Tutor', 500, 500)
        >>> initial_deck_length = len(player2.deck.cards)
        >>> tutor_test.effect(opponent_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()
        # You should add your implementation above this.
        print('{} discarded and re-drew 3 cards!'.format(opponent.name))
```

Use Ok to test your code:

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

Q8: TAs: Shift

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, opponent_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')
        >>> opponent_card = Card('other', 300, 600)
        >>> ta_test = TACard('TA', 500, 500)
        >>> ta_test.effect(opponent_card, player1, player2)
        >>> opponent_card.attack
        600
        >>> opponent_card.defense
        300
        """
        opponent_card.attack, opponent_card.defense = opponent_card.defense, opponent_carc
```

Use Ok to test your code:

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

Q9: The Instructor Arrives

A new challenger has appeared! Implement the `effect` method for the Instructors, who add 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:* If you mutate a list while iterating through it, you may run into trouble. Try iterating through a copy of the list instead. You can use slicing to make a copy of a list:
>
> ```
> >>> original = [1, 2, 3, 4]
> >>> copy = original[:]
> >>> copy
> [1, 2, 3, 4]
> >>> copy is original
> False
> ```

```
class InstructorCard(Card):
    cardtype = 'Instructor'

    def effect(self, opponent_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)
        >>> instructor_test = InstructorCard('Instructor', 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(), 'p
        >>> instructor_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 += opponent_card.attack
            card.defense += opponent_card.defense
        for card in opponent.deck.cards[:]:
            if card.attack == opponent_card.attack or card.defense == opponent_card.defens
                opponent.deck.cards.remove(card)
        # You should add your implementation above this.
        discarded = orig_opponent_deck_length - len(opponent.deck.cards)
        if discarded:
            print('{} cards were discarded from {}\'s deck!'.format(discarded, opponent.na
            return
```

Use Ok to test your code:

```
python3 ok -q InstructorCard.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!