Lab 7: Object-Oriented Programming

lab07.zip (lab07.zip)

Due by 11:59pm on Wednesday, October 12.

Starter Files

Download lab07.zip (lab07.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

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.

Object-Oriented Programming

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:
    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.
- **data attributes**: a variable that belongs to the instance (also called instance variables). Think of a data attribute as a quality of the object: cars have *wheels* and *color*, so we have given our `Car` instance `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 bound to an instance. Think of a method as a "verb" of the class: cars can *drive* and also *pop their tires*, so we have given our `Car` instance the methods `drive` and `pop_tire`. We call methods using **dot notation**:

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

- **constructor**: Constructors build an instance of the class. The constructor for car objects is `Car(color)`. When Python calls that constructor, it immediately calls the `__init__` method. That's where we initialize the data attributes:

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

The constructor takes in one argument, `color`. As you can see, this 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.

To avoid redefining attributes and methods for similar classes, we can write a single **base class** from which the similar classes **inherit**. For example, we can write a class called **Pet** and define **Dog** as a **subclass** of **Pet**:

```
class Pet:

    def __init__(self, name, owner):
        self.is_alive = True    # It's alive!!!
        self.name = name
        self.owner = owner

    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")

    def talk(self):
        print(self.name)

class Dog(Pet):

    def talk(self):
        super().talk()
        print('This Dog says woof!')
```

Inheritance represents a hierarchical relationship between two or more classes where one class **is a** more specific version of the other: a dog **is a** pet (We use **is a** to describe this sort of relationship in OOP languages, and not to refer to the Python `is` operator).

Since `Dog` inherits from `Pet`, the `Dog` class will also inherit the `Pet` class's methods, so we don't have to redefine `__init__` or `eat`. We do want each `Dog` to `talk` in a `Dog`-specific way, so we can **override** the `talk` method.

We can use `super()` to refer to the superclass of `self`, and access any superclass methods as if we were an instance of the superclass. For example, `super().talk()` in the `Dog` class will call the `talk()` method from the `Pet` class, but passing the `Dog` instance as the `self`.

This is a little bit of a simplification, and if you're interested you can read more in the Python documentation (https://docs.python.org/3/library/functions.html#super) on `super`.

Required Questions

Getting Started Videos

What Would Python Display?

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: WWPD: 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`.

```python
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`.

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

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

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

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)
______
```

Coding Practice

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
        "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q Account
```

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.

> **Hint**: Don't forget that `FreeChecking` inherits from `Account`! Check the Inheritance section in Topics for a refresher.

```
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. Still counts as a free withdrawal even tl
    'Insufficient funds'
    >>> ch.withdraw(3)    # Second withdrawal is also free
    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

    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q FreeChecking
```

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 questions can be found in `classes.py`.
- The game loop can be found in `cardgame.py`, and is responsible for running the game. You won't need to open or read this file to receive full credit.
- 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

Here's how the game 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 cards' *power* values are then calculated and compared. 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)
```

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 = 2000 - 3000 = -1000
P2: 1500 - 1000 = 1500 - 1000 = 500
```

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. Note that when a card is played, the card is removed from the player's hand. This means that the card is no longer in the hand when the effect takes place. All effects are applied *before* power is calculated during that round:

- An `AICard` will allow you to add the top two cards of your deck to your hand via drawing.
- A `TutorCard` will add a copy of the first card in your hand to your hand, at the cost of automatically losing the current round.
- A `TACard` discards the card with the highest `power` in your hand, and adds the discarded card's attack and defense to the played `TACard`'s stats.

- An `InstructorCard` can survive multiple rounds, as long as it has a non-negative `attack` or `defense`. However, at the beginning of each round that it is played, its attack and defense are reduced by 1000 each.

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.

```
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, opponent_card):
        """
        Calculate power as:
        (player card's attack) - (opponent card's defense)
        >>> staff_member = Card('staff', 400, 300)
        >>> other_staff = Card('other', 300, 500)
        >>> staff_member.power(other_staff)
        -100
        >>> other_staff.power(staff_member)
        0
        >>> third_card = Card('third', 200, 400)
        >>> staff_member.power(third_card)
        0
        >>> third_card.power(staff_member)
        -100
        """
        "*** YOUR CODE HERE ***"
```

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.

> **Hint:** use methods from the `Deck` class wherever possible when attempting to draw from the `deck` when implementing `Player.__init__` and `Player.draw`.

```
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, 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__
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. Remember that by the time `effect` is called, the played card is no longer in the player's hand.

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: Resourceful Resources

In the `AICard` class, implement the `effect` method for AIs. An `AICard` will allow you to add the top two cards of your deck to your hand via `draw`ing from your deck.

> Once you have finished writing your code for this problem, set `implemented` to `True` so that the text is printed when playing an `AICard`! *This is specifically for the* `AICard`*!* For future questions, make sure to look at the problem description carefully to know when to reassign any pre-designated variables.

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

    def effect(self, opponent_card, player, opponent):
        """
        Add the top two cards of your deck to your hand via drawing.
        Once you have finished writing your code for this problem,
        set implemented to True so that the text is printed when
        playing an AICard.

        >>> from cards import *
        >>> player1, player2 = Player(standard_deck.copy(), 'p1'), Player(standard_decl
        >>> opponent_card = Card("other", 500, 500)
        >>> test_card = AICard("AI Card", 500, 500)
        >>> initial_deck_length = len(player1.deck.cards)
        >>> initial_hand_size = len(player1.hand)
        >>> test_card.effect(opponent_card, player1, player2)
        AI Card allows me to draw two cards!
        >>> initial_hand_size == len(player1.hand) - 2
        True
        >>> initial_deck_length == len(player1.deck.cards) + 2
        True
        """
        "*** YOUR CODE HERE ***"
        implemented = False
        # You should add your implementation above this.
        if implemented:
            print(f"{self.name} allows me to draw two cards!")
```

Use Ok to test your code:

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

Q7: Tutors: Sneaky Search

In the `TutorCard` class, implement the `effect` method for Tutors. A `TutorCard` will add a copy of the first card in your hand to your hand, at the cost of automatically losing the current round. Note that if there are no cards in hand, a `TutorCard` will not add any cards to the hand, but must still lose the round.

To implement the "losing" functionality, it is sufficient to override `Card`'s `power` method to return `-float('inf')` in the `TutorCard` class. In addition, be sure to add copies of cards, instead of the chosen card itself! Class methods may come in handy.

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

    def effect(self, opponent_card, player, opponent):
        """
        Add a copy of the first card in your hand
        to your hand, at the cost of losing the current
        round. If there are no cards in hand, this card does
        not add any cards, but still loses the round.  To
        implement the second part of this effect, a Tutor
        card's power should be less than all non-Tutor cards.

        >>> from cards import *
        >>> player1, player2 = Player(standard_deck.copy(), 'p1'), Player(standard_decl
        >>> opponent_card = Card("other", 500, 500)
        >>> test_card = TutorCard("Tutor Card", 10000, 10000)
        >>> player1.hand = [Card("card1", 0, 100), Card("card2", 100, 0)]
        >>> test_card.effect(opponent_card, player1, player2)
        Tutor Card allows me to add a copy of a card to my hand!
        >>> print(player1.hand)
        [card1: Staff, [0, 100], card2: Staff, [100, 0], card1: Staff, [0, 100]]
        >>> player1.hand[0] is player1.hand[2] # must add a copy!
        False
        >>> player1.hand = []
        >>> test_card.effect(opponent_card, player1, player2)
        >>> print(player1.hand) # must not add a card if not available
        []
        >>> test_card.power(opponent_card) < opponent_card.power(test_card)
        True
        """
        "*** YOUR CODE HERE ***"
        added = False
        # You should add your implementation above this.
        if added:
            print(f"{self.name} allows me to add a copy of a card to my hand!")

    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

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

Q8: TAs: Power Transfer

In the `TACard` class, implement the `effect` method for TAs. A `TACard` discards the card with the highest `power` in your hand, and adds the discarded card's attack and defense to the played `TACard` 's stats. **Discarding** a card removes the card from your `hand` . If there are no cards in hand, the `TACard` should not do anything for its effect.

```
class TACard(Card):
    cardtype = 'TA'

    def effect(self, opponent_card, player, opponent, arg=None):
        """
        Discard the card with the highest `power` in your hand,
        and add the discarded card's attack and defense
        to this card's own respective stats.

        >>> from cards import *
        >>> player1, player2 = Player(standard_deck.copy(), 'p1'), Player(standard_decl
        >>> opponent_card = Card("other", 500, 500)
        >>> test_card = TACard("TA Card", 500, 500)
        >>> player1.hand = []
        >>> test_card.effect(opponent_card, player1, player2) # if no cards in hand, n
        >>> print(test_card.attack, test_card.defense)
        500 500
        >>> player1.hand = [Card("card1", 0, 100), TutorCard("tutor", 10000, 10000), C
        >>> test_card.effect(opponent_card, player1, player2) # must use card's power
        TA Card discards card3 from my hand to increase its own power!
        >>> print(player1.hand)
        [card1: Staff, [0, 100], tutor: Tutor, [10000, 10000]]
        >>> print(test_card.attack, test_card.defense)
        600 500
        """
        "*** YOUR CODE HERE ***"
        best_card = None
        # You should add your implementation above this.
        if best_card:
            print(f"{self.name} discards {best_card.name} from my hand to increase its
```

Use Ok to test your code:

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

Q9: Instructors: Immovable

In the `InstructorCard` class, implement the `effect` method for Instructors. An `InstructorCard` can survive multiple rounds, as long as it has a non-negative `attack` or `defense` at the end of a round. However, at the beginning of each round that it is played (including the first time!), its attack and defense are permanently reduced by 1000 each.

> To implement the "survive" functionality, the `InstructorCard` should re-add itself to the player's hand.

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

    def effect(self, opponent_card, player, opponent, arg=None):
        """
        Survives multiple rounds, as long as it has a non-negative
        attack or defense at the end of a round. At the beginning of the round,
        its attack and defense are permanently reduced by 1000 each.
        If this card would survive, it is added back to the hand.

        >>> from cards import *
        >>> player1, player2 = Player(standard_deck.copy(), 'p1'), Player(standard_decl
        >>> opponent_card = Card("other", 500, 500)
        >>> test_card = InstructorCard("Instructor Card", 1000, 1000)
        >>> player1.hand = [Card("card1", 0, 100)]
        >>> test_card.effect(opponent_card, player1, player2)
        Instructor Card returns to my hand!
        >>> print(player1.hand) # survives with non-negative attack
        [card1: Staff, [0, 100], Instructor Card: Instructor, [0, 0]]
        >>> player1.hand = [Card("card1", 0, 100)]
        >>> test_card.effect(opponent_card, player1, player2)
        >>> print(player1.hand)
        [card1: Staff, [0, 100]]
        >>> print(test_card.attack, test_card.defense)
        -1000 -1000
        """
        "*** YOUR CODE HERE ***"
        re_add = False
        # You should add your implementation above this.
        if re_add:
            print(f"{self.name} returns to my hand!")
```

Use Ok to test your code:

```
python3 ok -q InstructorCard.effect
```

After you complete this problem, you'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!