CS 61A Fall 2017

Object Oriented Programming

Discussion 5: October 4, 2017

1 Object Oriented Programming

In a previous lecture, you were introduced to the programming paradigm known as Object-Oriented Programming (OOP). OOP allows us to treat data as objects - like we do in real life.

For example, consider the **class** `Student`. Each of you as individuals are an **instance** of this class. So, a student `Angela` would be an instance of the class `Student`.

Details that all CS 61A students have, such as `name`, `year`, and `major`, are called **instance attributes**. Every student has these attributes, but their values differ from student to student. An attribute that is shared among all instances of `Student` is known as a **class attribute**. An example would be the `instructors` attribute; the instructors for CS 61A, DeNero and Hilfinger, are the same for every student in CS 61A.

All students are able to do homework, attend lecture, and go to office hours. When functions belong to a specific object, they are said to be **methods**. In this case, these actions would be bound methods of `Student` objects.

Here is a recap of what we discussed above:

- **class:** a template for creating objects
- **instance:** a single object created from a class
- **instance attribute:** a property of an object, specific to an instance
- **class attribute:** a property of an object, shared by all instances of a class
- **method:** an action (function) that all instances of a class may perform

Questions

1.1 Below we have defined the classes `Instructor`, `Student`, and `TeachingAssistant`, implementing some of what was described above. Remember that we pass the `self` argument implicitly to instance methods when using dot-notation.

```
class Instructor:
    degree = "PhD (Magic)" # this is a class attribute
    def __init__(self, name):
        self.name = name # this is an instance attribute

    def lecture(self, topic):
        print("Today we're learning about " + topic)

dumbledore = Instructor("Dumbledore")
class Student:
    instructor = dumbledore

    def __init__(self, name, ta):
        self.name = name
        self.understanding = 0
        ta.add_student(self)

    def attend_lecture(self, topic):
      Student.instructor.lecture(topic)
      if Student.instructor == dumbledore:
        print(Student.instructor.name + " is awesome!")
      else:
        print("I miss Dumbledore.")
      self.understanding += 1

    def visit_office_hours(self, staff):
        staff.assist(self)
        print("Thanks, " + staff.name)

class TeachingAssistant:
    def __init__(self, name):
        self.name = name
        self.students = {}

    def add_student(self, student):
        self.students[student.name] = student

    def assist(self, student):
        student.understanding += 1
```

What will the following lines output?

```
>>> snape = TeachingAssistant("Snape")
>>> harry = Student("Harry", snape)
>>> harry.attend_lecture("potions")
```

```
>>> hermione = Student("Hermione", snape)
>>> hermione.attend_lecture("herbology")
```

```
>>> hermione.visit_office_hours(TeachingAssistant("Hagrid"))
```

```
>>> harry.understanding
```

```
>>> snape.students["Hermione"].understanding
```

```
>>> Student.instructor = Instructor("Umbridge")
>>> Student.attend_lecture(harry, "transfiguration")
# Equivalent to harry.attend_lecture("transfiguration")
```

1.2 We now want to write three different classes, `Mailman`, `Client`, and `Email` to simulate email. Fill in the definitions below to finish the implementation!

```
class Email:
    """Every email object has 3 instance attributes: the
    message, the sender name, and the recipient name.
    """
    def __init__(self, msg, sender_name, recipient_name):









class Mailman:
    """Each Mailman has an instance attribute clients, which
    is a dictionary that associates client names with
    client objects.
    """
    def __init__(self):
        self.clients = {}

    def send(self, email):
        """Take an email and put it in the inbox of the client
        it is addressed to.
        """







    def register_client(self, client, client_name):
        """Takes a client object and client_name and adds it
        to the clients instance attribute.
        """
```

```
class Client:
    """Every Client has instance attributes name (which is
    used for addressing emails to the client), mailman
    (which is used to send emails out to other clients), and
    inbox (a list of all emails the client has received).
    """
    def __init__(self, mailman, name):
        self.inbox = []







    def compose(self, msg, recipient_name):
        """Send an email with the given message msg to the
        given recipient client.
        """







    def receive(self, email):
        """Take an email and add it to the inbox of this
        client.
        """
```

2 Inheritance

Let's explore another tool: **inheritance**. Suppose we want the `Dog` and `Cat` classes.

```
class Dog(object):
    def __init__(self, name, owner):
        self.name = name
        self.owner = owner
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says woof!")

class Cat(object):
    def __init__(self, name, owner, lives=9):
        self.name = name
        self.owner = owner
        self.lives = lives
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print(self.name + " says meow!")
```

Notice that there's a lot of repeated code! This is where inheritance comes in. In Python, a class can **inherit** the instance variables and methods of another class.

```
class Pet(object):
    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 __init__(self, name, owner):
        Pet.__init__(self, name, owner)
    def talk(self):
        print(self.name + ' says woof!')
```

Inheritance often represents a hierarchical relationship between two or more classes where one class *is a* more specific version of the other. For example, a dog *is a* pet. By making `Dog` a subclass of `Pet`, we did not have to redefine `self.name`, `self.owner`, or `eat`. However, since we want `Dog` to `talk` differently, we did redefine, or **override**, the `talk` method.

Questions

2.1 Implement the `Cat` class by inheriting from the `Pet` class. Make sure to use superclass methods wherever possible. In addition, add a `lose_life` method to the `Cat` class.

```
class Cat(Pet):
    def __init__(self, name, owner, lives=9):




    def talk(self):
        """A cat says meow! when asked to talk."""




    def lose_life(self):
        """A cat can only lose a life if they have at
        least one life. When lives reaches zero, 'is_alive'
        becomes False.
        """
```

2.2 More cats! Fill in the methods for `NoisyCat`, which is just like a normal `Cat`. However, `NoisyCat` talks a lot, printing twice whatever a `Cat` says.

```
class NoisyCat(Cat):
    """A Cat that repeats things twice."""
    def __init__(self, name, owner, lives=9):
        # Is this method necessary? Why or why not?



    def talk(self):
        """Repeat what a Cat says twice."""
```

2.3 (Summer 2013 Final) What would Python display?

```
class A:
    def f(self):
        return 2
    def g(self, obj, x):
        if x == 0:
            return A.f(obj)
        return obj.f() + self.g(self, x - 1)

class B(A):
    def f(self):
        return 4

>>> x, y = A(), B()
>>> x.f()
```

```
>>> B.f()
```

```
>>> x.g(x, 1)
```

```
>>> y.g(x, 2)
```

2.4 Implement the `Yolo` class so that the following interpreter session works as expected. (Summer 2013 Final)

```
>>> x = Yolo(1)
>>> x.g(3)
4
>>> x.g(5)
6
>>> x.motto = 5
>>> x.g(5)
10
```

2.5 (Summer 2015 Final) The TAs are building a social networking website called CS61A+. The TAs plan to represent the network in a class called `Network` that supports the following method:

- `add_friend(user1, user2)` adds `user1` and `user2` to each other's friends lists. If `user1` or `user2` are not in the `Network`, add them to the dictionary of friends.

Help the TAs implement these two methods to make their social networking website popular!

```
class Network:
    """
    >>> cs61a_plus = Network()
    >>> cs61a_plus.add_friend('Robert', 'Jeffrey')
    >>> cs61a_plus.friends['Robert']
    ['Jeffrey']
    >>> cs61a_plus.friends['Jeffrey']
    ['Robert']
    >>> cs61a_plus.add_friend('Jessica', 'Robert')
    >>> cs61a_plus.friends['Robert']
    ['Jeffrey', 'Jessica']
    """
    def __init__(self):
        self.friends = {}       # Maps users to a list of their friends

    def add_friend(self, user1, user2):


        if __________________________________________________:


             __________________________________________________


        if __________________________________________________:


             __________________________________________________


        _____________________________________________________


        _____________________________________________________
```

CS61A+ turns out to be unpopular. To attract more users, the TAs want to implement a feature that checks if two users have at most `n` degrees of separation. Consider the following CS61A+ `Network`:

```
self.friends = {
    'Robert':  ['Jeffrey', 'Jessica'],
    'Jeffrey': ['Robert', 'Jessica', 'Yulin'],
    'Jessica': ['Robert', 'Jeffrey', 'Yulin'],
    'Yulin':   ['Jeffrey', 'Jessica'],
    'Albert':  []
}
```

- There is 1 degree of separation between Robert and Jeffrey, because they are direct friends.
- There are 2 degrees of separation between Robert and Yulin (Robert → Jessica → Yulin)
- The degree of separation between Albert and anyone else is undefined, since Albert has no friends.

```
class Network:
    # Code from previous question

    def degrees(self, user1, user2, n):
        """In these doctests, assume cs61a_plus is a Network with the
        dictionary of friends described in the example.

        >>> cs61a_plus.degrees('Robert', 'Yulin', 2)    # Exactly 2 degrees
        True
        >>> cs61a_plus.degrees('Robert', 'Jessica', 2)  # Less than 2 degrees
        True
        >>> cs61a_plus.degrees('Yulin', 'Robert', 1)    # More than 1 degree
        False
        >>> cs61a_plus.degrees('Albert', 'Jessica', 10) # No friends!
        False
        """
        if ______________________________________:
            return True

        elif ____________________________________:
            return False

        for friend in _______________________________:

            if ______________________________________:
                return True

        return ______________________________________
```