CS 61A Summer 2017 — Object Oriented Programming

Discussion 8: July 18, 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, Kevin and Stan, 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)
        print(Student.instructor.name + " is awesome!")
        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")
```

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.

For example: `Bar` inherits from `Foo`. We call `Foo` the **base class** (the class that is being inherited) and `Bar` the **subclass** (the class that does the inheriting). Notice that `Foo` also inherits from the `object` class. In Python, `object` is the top-level base class that provides basic functionality; everything inherits from it.

```
class Foo(object):
    # This is the base class

class Bar(Foo):
    # This is the subclass
```

```
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 What would Python display? (Summer 2013 Final)

```
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
```