OBJECT ORIENTED PROGRAMMING 5

COMPUTER SCIENCE 61A

October 6, 2016

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** `CS61A_Student`. Each of you as individuals are an **instance** of this class. So, a student `Mitas` would be an instance of the class `CS61A_Student`.

Details that all CS61A 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 `CS61A_Student` is known as a **class attribute**. An example would be the `instructors` attribute; the instructor for 61A, Professor DeNero, is the same for every student in CS61A.

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 `CS61A_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 the same class
- **method:** an action (function) that all instances of a class may perform

1.1 Questions

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")
```

Solution:
```
Today we're learning about potions
Dumbledore is awesome!
```

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

Solution:
```
Today we're learning about herbology
Dumbledore is awesome!
```

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

Solution:
```
Thanks, Hagrid
```

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

Solution:
```
1
```

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

Solution:
```
2
```

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

Solution:
```
Today we're learning about transfiguration
Umbridge is awesome!
```

2. We now want to write three different classes, `Postman`, `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):
```

Solution:

```
        self.msg = msg
        self.sender_name = sender_name
        self.recipient_name = recipient_name
```

```
class Postman:
    """Each Postman 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.
        """
```

Solution:

```
        client = self.clients[email.recipient_name]
        client.receive(email)
```

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

Solution:

```
        self.clients[client_name] = client
```

```
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 = []
```

Solution:

```
        self.mailman = mailman
        self.name = name
        self.mailman.register_client(self, self.name)
```

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

Solution:

```
        email = Email(msg, self.name, recipient_name)
        self.mailman.send(email)
```

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

Solution:

```
        self.inbox.append(email)
```

2 Inheritance

Let's explore another powerful object-oriented programming tool: **inheritance**. Suppose we want to write `Dog` and `Cat` classes. Here's our first attempt:

```
class Dog(object):
    def __init__(self, name, owner, color):
        self.name = name
        self.owner = owner
        self.color = color
    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 the only difference between both the `Dog` and `Cat` classes are the `talk` method as well as the `color` and `lives` attributes. That's a lot of repeated code!

This is where inheritance comes in. In Python, a class can **inherit** the instance variables and methods of a another class without having to type them all out again. For example:

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

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

`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, even when you don't specify a class to inherit from. One common use of inheritance is to represent a hierarchical relationship between two or more classes where one class *is a* more specific version of the other class. For example, a dog *is a* pet.

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

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.

The line `Pet.__init__(self, name, owner)` in the `Dog` class is necessary for inheriting the instance attributes and methods from `Pet`. Notice that when we call `Pet.__init__`, we need to pass in `self` as a regular argument (that is, inside the parentheses, rather than by dot-notation) since `Pet` is a class, not an instance.

2.1 Questions

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):
```

Solution:

```
        Pet.__init__(self, name, owner)
        self.lives = lives
```

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

Solution:

```
        print('meow!')
```

```
    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.
        """
```

Solution:

```
        if self.lives > 0:
            self.lives -= 1
            if self.lives == 0:
                self.is_alive = False
        else:
            print("This cat has no more lives to lose :(")
```

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?
```

Solution:
```
Cat.__init__(self, name, owner, lives)
```

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

Solution:
```
Cat.talk(self)
Cat.talk(self)
```

3. What would Python print? (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()
```

Solution: `2`

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

Solution: `Error` (missing self argument)

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

Solution: `4`

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

Solution: `8`

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
```

Solution:

```
class Yolo:
    def __init__(self, motto):
        self.motto = motto
    def g(self, n):
        return self.motto + n
```