1 Object Oriented Programming Review

This week, 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 your CS 61A classmates and yourself. You are all objects of the class CS61A student. Each of you as individuals is an instance of this class.

Some details that all CS61A students have, such as name, year in school, and major, are called attributes. Every student has these attributes, but their values differ from student to student. An attribute that has the same value for every single student is known as a class attribute. An example would be the instructors attribute; the instructors for 61A, Robert and Albert, are the same for every single student in 61A. However, not all students have the same TA, so that would be an instance attribute.

Actions that all students are able to perform include doing homework, attending lecture, and going to office hours. These actions would be methods of the CS61A student class.

2 Questions

1. Below we have defined the classes Instructor, Student, and TeachingAssistant, implementing some of what was described above.
```python
class Instructor:
    who = 0

    def __init__(self, name):
        self.name = name

    def lecture(self, topic):
        print("Today we're learning about " + topic)
        Instructor.who = 1 - Instructor.who

albert, robert = Instructor("Albert"), Instructor("Robert")

class Student:
    instructors = albert, robert

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

    def attend_lecture(self, topic):
        current_lecturer = Student.instructors[Instructor.who]
        current_lecturer.lecture(topic)
        print(current_lecturer.name + " is a great lecturer!")
        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?

```python
>>> jessica = TeachingAssistant("Jessica")
>>> john = Student("John", jessica)
>>> paul = Student("Paul", jessica)
>>> john.attend_lecture("OOP")

>>> paul.attend_lecture("trees")

>>> paul.visit_office_hours(TeachingAssistant("Tammy"))

>>> john.understanding

>>> jessica.students["Paul"].understanding

>>> Instructor.who = 1
>>> Student.attend_lecture(john, "lists")

>>> denero = Instructor("DeNero")
>>> denero.who = 1
>>> Instructor.who
```
Let's explore another powerful object-oriented programming tool: inheritance. Suppose we want to write Dog and Cat classes. Here's our first attempt:

```python
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 both the only difference between Dog and Cat classes is the talk method and the color and lives attributes. That's a lot of effort for so much 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:

```python
class Foo(object):
    # This is the superclass

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

Bar inherits from Foo. We call Foo the superclass (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 superclass that provides basic functionality; everything inherits from it. 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, since Pet is a class, not an instance.
3.1 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.

   ```python
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. Assume these commands are entered in order. What would Python output?

   ```python
>>> class Foo(object):
...     ...     def __init__(self, a):
...         self.a = a
...     ...     def garply(self):
...         ...         return self.baz(self.a)
>>> class Bar(Foo):
...     ...     a = 1
...     ...     def baz(self, val):
...         ...         return val
>>> f = Bar(3)
>>> >>>
```
>>> b.a

>>> f.garply()

>>> b.garply()

>>> b.a = 9
>>> b.garply()

>>> f.baz = lambda val: val * val
>>> f.garply()

3.2 Extra Questions

1. 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):
       
       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. Fill in the classes `Emotion`, `Joy`, and `Sadness` below so that you get the following output from the Python interpreter.

```python
>>> Emotion.num
0
>>> joy = Joy()
>>> sadness = Sadness()
>>> Emotion.num  # number of Emotion instances created
2
>>> joy.power
5
>>> joy.catchphrase()  # Print Joy's catchphrase
Think positive thoughts
>>> sadness.catchphrase()  # Print Sad's catchphrase
I'm positive you will get lost
>>> sadness.power
5
>>> joy.feeling(sadness)  # When both Emotions have same power value, print "Together"
Together
>>> sadness.feeling(joy)
Together
>>> joy.power = 7
>>> joy.feeling(sadness)  # Print the catchphrase of the more powerful feeling before the less powerful feeling
Think positive thoughts
I'm positive you will get lost
>>> sadness.feeling(joy)
Think positive thoughts
I'm positive you will get lost
```
class Emotion(_______):

def __init__(self):

def feeling(self, other):

class Joy(_______):

def catchphrase(self):

class Sadness(_______):

def catchphrase(self):