Object Oriented Programming

Last 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 the class \texttt{CS61A\_Student}. Each of you as individuals are an instance of this class. So, a student Sumukh would be an instance of the class \texttt{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 \texttt{CS61A\_Student} is known as a class attribute. An example would be the instructors attribute; the instructor for 61A, Professor Hilfinger, is the same for every student in CS61A. However, the TA attribute isn’t shared among all students since students will not necessarily have the same TA, so that would be an instance attribute.

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 \texttt{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. 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" # 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)

hilfinger = Instructor("Professor Hilfinger")

class Student:
    instructor = hilfinger

    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, student):
        self.students[student.name] = student

    def assist(self, self, student):
        student.understanding += 1
What will the following lines output?

```python
>>> soumik = TeachingAssistant("Soumik")
>>> kelly = Student("Kelly", soumik)
>>> kelly.attend_lecture("OOP")

>>> kristin = Student("Kristin", soumik)
>>> kristin.attend_lecture("trees")

>>> kristin.visit_office_hours(TeachingAssistant("James"))

>>> kelly.understanding

>>> soumik.students["Kristin"].understanding

>>> Student.instructor = Instructor("Professor DeNero")
>>> Student.attend_lecture(kelly, "lists")
# Equivalent to kelly.attend_lecture("lists")
```
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 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:

```python
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.

```python
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.

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

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 = Foo(4)
>>> b = Bar(3)
>>> f.a
```


2.2 Extra Questions: You Oughta Like Objects

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.
   ```python
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. What would Python print? (Summer 2013 Final)
   ```python
class A:
    def f(self):
        return 2
    def g(self, obj, x):
        if x == 0:
            return 2
```
```
return A.f(obj)
return obj.f() + self.g(self, x - 1)
```

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

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

>>> B.f()

>>> x.g(x, 1)

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

3. Implement the Yolo class so that the following interpreter session works as expected.
(Summer 2013 Final)
```python
>>> x = Yolo(1)
>>> x.g(3)
4
>>> x.g(5)
6
>>> x.motto = 5
>>> x.g(5)
10
```
Until now, you’ve been able to access variables in parent frames, but you have not been able to modify them. The `nonlocal` keyword can be used to modify a variable in the parent frame outside the current frame. For example, consider `stepper`, which uses `nonlocal` to modify `num`:

```python
def stepper(num):
    def step():
        nonlocal num  # declares num as a nonlocal variable
        num = num + 1  # modifies num in the stepper frame
        return num
    return step
```

However, there are two important caveats with `nonlocal` variables:

- **Global variables** cannot be modified using the `nonlocal` keyword.
- **Variables in the current frame** cannot be overridden using the `nonlocal` keyword.

### 3.1 Some Common Misconceptions

1. What is wrong with the following code?
   ```python
   a = 5
   def another_add_one():
       nonlocal a
       a += 1
   another_add_one()
   ```

2. What is wrong with the following code?
   ```python
   def adder(x):
       def add(y):
           nonlocal x, y
           x += y
           return x
       return add
   adder(2)(3)
   ```
3.2 Fill in the Blank

1. The bathtub below simulates an epic battle between Finn and Kylo Ren over a populace of rubber duckies. Fill in the body of `duddy` so that all doctests pass.

```python
def bathtub(n):
    
    >> annhilator = bathtub(500)  # the force awakens...
    >> kylo_ren = annhilator(10)
    >> kylo_ren()
    490 rubber duckies left
    >> finn = annhilator(-20)
    >> finn()
    510 rubber duckies left
    >> kylo_ren()
    500 rubber duckies left

    def ducky_annihilator(rate):
        def ducky():

            return ducky

    return ducky_annihilator
```
3.3 Environment Diagrams

1. Draw the environment diagram for the code below:

```python
def stepper(num):
    def step():
        nonlocal num
        num = num + 1
        return num
    return step

s = stepper(3)
s()
s()```
2. Given the definition of `make_shopkeeper` below, draw the environment diagram.

```python
def make_shopkeeper(total_gold):
    def buy(cost):
        nonlocal total_gold
        if total_gold < cost:
            return 'Go farm some more champions'
        total_gold = total_gold - cost
        return total_gold
    return buy

infinity_edge, zeal, gold = 3800, 1100, 3800
shopkeeper = make_shopkeeper(gold - 1000)
shopkeeper(zeal)
shopkeeper(infinity_edge)
```