INHERITANCE AND NONLOCAL 5

COMPUTER SCIENCE 61A

October 8, 2015

1 Object Oriented Programming

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

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 that all instances of a class may perform

2 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" # 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)

denero = Instructor("Professor DeNero")
class Student:
    instructor = denero

    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?

```
>>> alvin = TeachingAssistant("Alvin")
>>> michelle = Student("Michelle", alvin)
>>> michelle.attend_lecture("OOP")
```

Solution:
```
Today we're learning about OOP
Professor DeNero is awesome!
```

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

Solution:
```
Today we're learning about trees
Professor DeNero is awesome!
```

```
>>> kristin.visit_office_hours(TeachingAssistant("Luise"))
```

Solution:
```
Thanks, Luise
```

```
>>> michelle.understanding
```

Solution:
```
1
```

```
>>> alvin.students["Kristin"].understanding
```

Solution:
```
2
```

```
>>> Student.instructor = Instructor("Professor Hilfinger")
>>> Student.attend_lecture(michelle, "lists")
# Equivalent to michelle.attend_lecture("lists")
```

Solution:
```
Today we're learning about lists
Professor Hilfinger is awesome!
```

3 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.

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.

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.

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

```
>>> 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
```

Solution: `4`

```
>>> b.a
```

Solution: `3`

```
>>> f.garply()
```

Solution: `AttributeError: 'Foo'object has no attribute 'baz'`

```
>>> b.garply()
```

Solution: `3`

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

Solution: `9`

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

Solution: `16`

3.2 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.

```
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)
```

2. 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`

3. 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
```

4 Nonlocal

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`:

```
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.
- **Function parameters** cannot be overridden using `nonlocal` variables.

4.1 Some Common Misconceptions

1. What is wrong with the following code?

```
a = 5
def another_add_one():
    nonlocal a
    a += 1
another_add_one()
```

Solution: Nonlocal cannot be used to modify variables in the global frame.

2. What is wrong with the following code?

```
a = 5
def adder(x):
```

```
    def add(y):
        nonlocal x, y
        x += y
        return x
    return add
adder(2)(3)
```

Solution: Nonlocal cannot be used for y if there is no variable y defined in a parent frame. Here y is already a local variable.

4.2 Fill in the Blank

1. The bathtub below simulates an epic battle between Obi-Wan and Darth Vader over a populace of rubber duckies. Fill in the body of `ducky` so that all doctests pass.

```
def bathtub(n):
    """
    >>> annihilator = bathtub(500) # the force awakens...
    >>> darth_vader = annihilator(10)
    >>> darth_vader()
    490 rubber duckies left
    >>> obi_wan = annihilator(-20)
    >>> obi_wan()
    510 rubber duckies left
    >>> darth_vader()
    500 rubber duckies left
    """
    def ducky_annihilator(rate):
        def ducky():
```

Solution:

```
            nonlocal n
            n = n - rate
            print(n, 'rubber duckies left')
```

```
        return ducky
    return ducky_annihilator
```

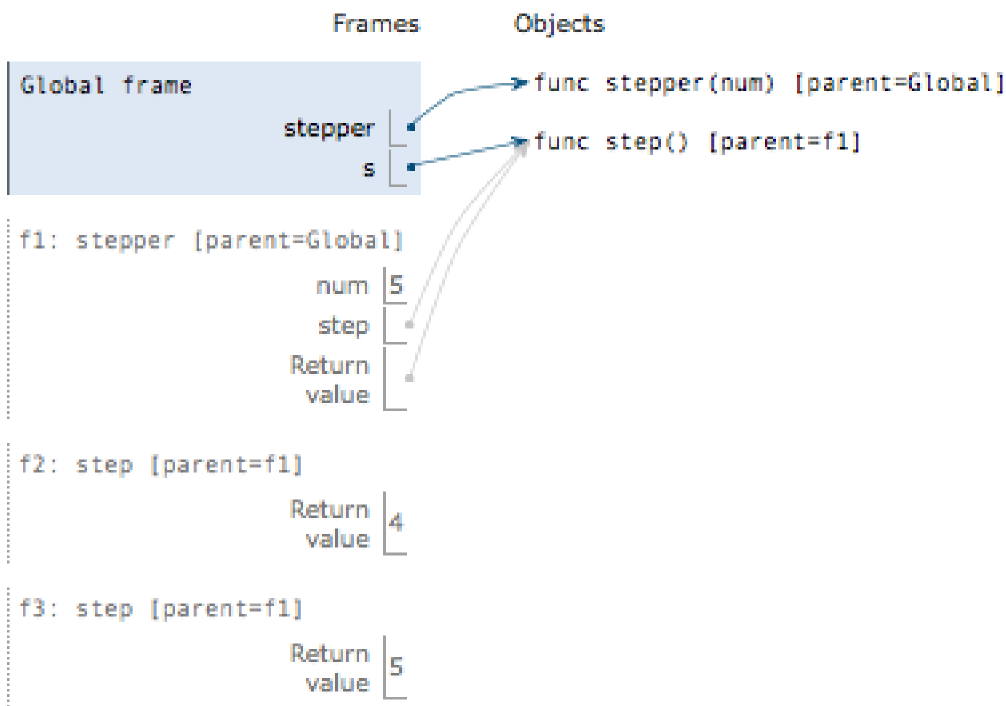
4.3 Environment Diagrams

1. Draw the environment diagram for the code below:

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

s = stepper(3)
s()
s()
```

Solution:



2. Given the definition of `make_shopkeeper` below, draw the environment diagram.

```
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)
```

Solution: