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 is 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 instructor for CS 61A, Professor DeNero, is 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. There are more questions on the next page.

```
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)
      if Student.instructor == dumbledore:
        print(Student.instructor.name + " is awesome!")
      else:
        print("I miss Dumbledore.")
      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")

>>> harry.visit_office_hours(TeachingAssistant("Hagrid"))

>>> harry.understanding

>>> [name for name in snape.students]

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

Equivalent to harry.attend_lecture("transfiguration")

1.2 We now want to write three different classes, Mailman, Client, and Email to simulate email. Fill in the definitions below to finish the implementation! There are more methods to fill out on the next page.

```
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):
```

class Mailman:

```
"""Each Mailman 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.
    """
```

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

```
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 = []
def compose(self, msg, recipient_name):
    """Send an email with the given message msg to the
    given recipient client.
    11 11 11
def receive(self, email):
    """Take an email and add it to the inbox of this
    client.
    11 11 11
```

2 Inheritance

Python classes can implement a useful abstraction technique known as **inheritance**. To illustrate this concept, consider the following Dog and Cat classes.

```
class Dog(object):
    def __init__(self, name, owner):
        self.is_alive = True
        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.is_alive = True
        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 because dogs and cats share a lot of similar qualities, there is a lot of repeated code! To avoid redefining attributes and methods for similar classes, we can write a single **superclass** from which the similar classes **inherit**. For example, we can write a class called Pet and redefine Dog as a **subclass** of 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 talk(self):
        print(self.name + ' says woof!')
```

Inheritance represents a hierarchical relationship between two or more classes where one class is a more specific version of the other, e.g. a dog is a pet. Because Dog inherits from Pet, we didn't have to redefine __init__ or eat. However, since we want Dog to talk in a way that is unique to dogs, we did override the talk method.

Questions

2.1 Below is a skeleton for the Cat class, which inherits from the Pet class. To complete the implementation, override the __init__ and talk methods and add a new

```
lose_life method.
    Hint: You can call the __init__ method of Pet to set a cat's name and owner.
    class Cat(Pet):
        def __init__(self, name, owner, lives=9):
        def talk(self):
            """ Print out a cat's greeting.
            >>> Cat('Thomas', 'Tammy').talk()
            Thomas says meow!
            11 11 11
        def lose_life(self):
            """Decrements a cat's life by 1. When lives reaches zero, 'is_alive'
            becomes False.
            .....
2.2 More cats! Fill in this implemention of a class called NoisyCat, which is just like a
    normal Cat. However, NoisyCat talks a lot - twice as much as a regular Cat!
    class _____: # Fill me in!
        """A Cat that repeats things twice."""
        def __init__(self, name, owner, lives=9):
            # Is this method necessary? Why or why not?
        def talk(self):
            """Talks twice as much as a regular cat.
            >>> NoisyCat('Magic', 'James').talk()
            Magic says meow!
            Magic says meow!
            .....
```

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

2.4 (Summer 2013 Final) Implement the Foo class so that the following interpreter session works as expected.

```
>>> x = Foo(1)
>>> x.g(3)
4
>>> x.g(5)
6
>>> x.bar = 5
>>> x.g(5)
10
```

class Foo:

>>> y.g(x, 2)