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`, 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 `students` attribute; the number of students that exist is not a property of any given student but rather of all of them.

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 Professor and Student, 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 Student:
    students = 0 # this is a class attribute
    def __init__(self, name, ta):
        self.name = name # this is an instance attribute
        self.understanding = 0
        Student.students += 1
        print("There are now", Student.students, "students")
        ta.add_student(self)

        def visit_office_hours(self, staff):
            staff.assist(self)
            print("Thanks, " + staff.name)

class Professor:
    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

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.
What will the following lines output?

```python
>>> callahan = Professor("Callahan")
>>> elle = Student("Elle", callahan)

There are now 1 students

>>> elle.visit_office_hours(callahan)

Thanks, Callahan

>>> elle.visit_office_hours(Professor("Paulette"))

Thanks, Paulette

>>> elle.understanding

2

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

['Elle']

>>> x = Student("Vivian", Professor("Stromwell")).name

There are now 2 students

>>> x

'Vivian'

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

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.
['Elle']
1.2 In this question, we will implement a special version of a list called a MinList. A MinList acts similarly to a list in that you can append items and pop items from it, but it only can pop the smallest number.

Implement the class MinList such it contains the following methods:

1. `append(self, item)`: add an element to the MinList
2. `pop(self)`: remove and return the smallest element.

Each instance also contains an attribute `size` that represents how many elements it contains. Remember to update `size` in `append` and `pop`!

When you initialize a MinList, it will start out with no elements.

**Hint:** It might be helpful to actually include a Python list as an instance attribute for each MinList to keep track of what items we have.

```python
class MinList:
    """A list that can only pop the smallest element ""
    def __init__(self):
        self.items = []  # start out with no items
        self.size = 0

    def append(self, item):
        """Appends an item to the MinList
        >>> m = MinList()
        >>> m.append(4)
        >>> m.append(2)
        >>> m.size
        2
        ""
        self.items.append(item)
        self.size += 1

    def pop(self):
        """Removes and returns the smallest item from the MinList
        >>> m = MinList()
        >>> m.append(4)
        >>> m.append(1)
        >>> m.append(5)
        ""
```
>>> m.pop()
1
>>> m.size
2

smallest_item = \texttt{min}(self.items)
self.items.remove(smallest_item)
self.size -= 1
\texttt{return} smallest_item

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.
1.3 Tutorial:

We now want to write three different classes, Server, 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.

We suggest that you approach this problem by first filling out the Email class, then fill out the register_client method of Server, then implement the Client class, and lastly fill out the send method of the Server class.

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):
    self.msg = msg
    self.sender_name = sender_name
    self.recipient_name = recipient_name

class Server:
    """Each Server 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."
    """
    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 them to the clients instance attribute."
    """
    self.clients[client_name] = client
class Client:
    """Every Client has instance attributes name (which is used for addressing emails to the client), server (which is used to send emails out to other clients), and inbox (a list of all emails the client has received). """
    def __init__(self, server, name):
        self.inbox = []
        self.server = server
        self.name = name
        self.server.register_client(self, self.name)

    def compose(self, msg, recipient_name):
        """Send an email with the given message msg to the given recipient client. """
        email = Email(msg, self.name, recipient_name)
        self.server.send(email)

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

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

```python
class Dog:
    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:
    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:

```python
class Pet:
    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.

*Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.*
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):
        Pet.__init__(self, name, owner)
        self.lives = lives

    def talk(self):
        """Print out a cat's greeting."

        >>> Cat('Thomas', 'Tammy').talk()
        Thomas says meow!
        """

        print(self.name + ' says meow!')

    def lose_life(self):
        """Decrements a cat's life by 1. When lives reaches zero, 'is_alive'
        becomes False. If this is called after lives has reached zero, print out
        that the cat has no more lives to lose."
        """

        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 :(")
Video walkthrough

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.
Tutorial: More cats! Fill in this implementation of a class called NoisyCat, which is just like a normal Cat. However, NoisyCat talks a lot – twice as much as a regular Cat! Make sure to also fill in the __repr__ method for NoisyCat, so we know how to construct it! As a hint: You can use several string formatting methods to make this easier.

E.g.:

```python
>>> 'filling in {} spaces {} and {}'.format('blank', 'here', 'here')
'filling in blank spaces here and here'
```

```python
class _________________: # Fill me in!

class NoisyCat(Cat):

    """A Cat that repeats things twice."""
    def __init__(self, name, owner, lives=9):
        # Is this method necessary? Why or why not?

            Cat.__init__(self, name, owner, lives)

    No, this method is not necessary because NoisyCat already inherits Cat's __init__ method

    def talk(self):
        """Talks twice as much as a regular cat."

            >>> NoisyCat('Magic', 'James').talk()
            Magic says meow!
            Magic says meow!
            """

            Cat.talk(self)
            Cat.talk(self)

    def __repr__(self):
        """The interpreter-readable representation of a NoisyCat"
```
>>> muffin = NoisyCat('Muffin', 'Catherine')
>>> repr(muffin)
"NoisyCat('Muffin', 'Catherine')"
>>> muffin
NoisyCat('Muffin', 'Catherine')

return "NoisyCat({}, {}).format(repr(self.name), repr(self.owner))

Video walkthrough