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. 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.

```python
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
```


What will the following lines output?

```python
>>> snape = Professor("Snape")
>>> harry = Student("Harry", snape)

>>> harry.visit_office_hours(snape)

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

>>> harry.understanding

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

>>> Student("Hermione", Professor("McGonagall")).name

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

```python
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 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. """

    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), 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 = []

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

    def receive(self, email):
        """Take an email and add it to the inbox of this client. """
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():
    def __init__(self, name, owner):
        self.is_alive = True
        self.name = name
        self.owner = owner
    def eat(self, thing):
        print(f'{self.name} ate a {str(thing)}!
    def talk(self):
        print(f'{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(f'{self.name} ate a {str(thing)}!
    def talk(self):
        print(f'{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():
    def __init__(self, name, owner):
        self.is_alive = True  # It's alive!!!
        self.name = name
        self.owner = owner
    def eat(self, thing):
        print(f'{self.name} ate a {str(thing)}!
    def talk(self):
        print(f'{self.name}')

class Dog(Pet):
    def talk(self):
        print(f'{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.

\textit{Hint:} You can call the \_\_init\_\_ method of Pet to set a cat’s name and owner.

\begin{verbatim}
class Cat(Pet):
    def \_\_init\_\_(self, name, owner, lives=9):
        pass

    def talk(self):
        """Print out a cat’s greeting."
        >>> Cat('Thomas', 'Tammy').talk()
        Thomas says meow!
        """

    def lose\_life(self):
        """Decrements a cat’s life by 1. When lives reaches zero, ‘is\_alive’
        becomes False."
        pass
\end{verbatim}

2.2 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!

\begin{verbatim}
class NoisyCat: # 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!
        """
\end{verbatim}
Extra Questions

2.3 (Summer 2013 Final) What would Python display?

```python
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)

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

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

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

class Foo:
```

3 Nonlocal

Until now, you’ve been able to access names in parent frames, but you have not been able to modify them. The nonlocal keyword can be used to modify a binding
in a parent frame. For example, consider `stepper`, which uses `nonlocal` to modify `num`:

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

>>> step1 = stepper(10)
>>> step1()  # Modifies and returns num
11
>>> step1()  # num is maintained across separate calls to step
12
>>> step2 = stepper(10)  # Each returned step function keeps its own state
>>> step2()
11
```

As illustrated in this example, `nonlocal` is useful for maintaining state across different calls to the same function.

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

- **Global names** cannot be modified using the `nonlocal` keyword.

- **Names in the current frame** cannot be overridden using the `nonlocal` keyword. This means we cannot have both a local and nonlocal binding with the same name in a single frame.

Because `nonlocal` lets you modify bindings in parent frames, we call functions that use it **mutable functions**.
Questions

3.1 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)
```
3.2 Write a function that takes in a number \( n \) and returns a one-argument function. The returned function takes in a function that is used to update \( n \). It should return the updated \( n \).

def memory(n):
    """
    >>> f = memory(10)
    >>> f(lambda x: x * 2)
    20
    >>> f(lambda x: x - 7)
    13
    >>> f(lambda x: x > 5)
    True
    """