CS61A Notes – Week 6: Object Oriented Programming

This week you were introduced to the programming paradigm known as Object Oriented Programming. If you've programmed in a language like Java or C++, this concept should already be familiar to you.

Object oriented programming is heavily based on the idea of data abstraction. In object oriented programming we have:

- Classes -- Blueprints for creating types of objects. These can be though of as the abstract data types in an object oriented program.
- Objects -- These are the actual pieces of data in the program. You sometimes hear of objects being called **instances**, which means that it is an object belonging to a certain class (we say things like "object X is an instance of the Y class").

Objects can be thought of as "smart data." Objects carry around **methods** which are functions that you can call to have the object perform an action. In addition to methods, an object has access to two types of values:

- Instance variables -- these are values that the object knows and can change.
- Class variables -- these are values that all objects of a class (data type) know and share. If any object of the class changes the class variable, all other objects in the class will see the same new value.

When defining a class, we use the following syntax:

```
class OurClass(ParentClass):
"""Definition of class here (methods and class variables)."""
```

Where OurClass is the name of the new class and ParentClass is the name of the class it inherits from (we'll talk more about inheritance later).

To define a method, we write it almost exactly the same way as when we define functions but we always include one argument before all other arguments, self, which is used to refer to the instance we used to call the method.

Finally, to use a class or instance's **attributes** (methods and variables), we use "dot notation." The way dot notation works is that we refer to the method or variable, bar, of a class or instance, foo, by saying: "foo.bar" which says "I want foo's attribute bar."

Example:

As a starting example, consider the classes Skittle and Bag, which is used to represent a single piece of Skittles candy and a bag of skittles respectively.

```
class Skittle(object):
"""A Skittle object has a color to describe it."""
    def __init__(self, color):
        self.color = color


class Bag(object):
"""A Bag is a collection of skittles. All bags share the number
```

```
of Bags ever made (sold) and each bag keeps track of its skittles
in a list.
"""
    number_sold = 0

    def __init__(self):
        self.skittles = []
        Bag.number_sold += 1

   def tag_line(self):
        """Print the Skittles tag line."""
        print("Taste the rainbow!")

    def print_bag(self):
        print([s.color for s in self.skittles])

    def take_skittle(self):
        """Take the first skittle in the bag (from the front of the
        skittles list).
        """
        skittle_to_eat = self.skittles[0]
        self.skittles = self.skittles[1:]
        return skittle_to_eat

    def add_skittle(self, s):
        """Add a skittle to the bag."""
        self.skittles.append(s)
```

QUESTIONS

1. What does Python print for each of the following:

```
>>> toms_bag = Bag()
>>> toms_bag.print_bag()
```

_____

```
>>> toms_bag.add_skittle(Skittle("blue"))
>>> toms_bag.print_bag()
```

_____

```
>>> toms_bag.add_skittle(Skittle("red"))
>>> toms_bag.add_skittle(Skittle("green"))
>>> toms_bag.add_skittle(Skittle("red"))
>>> toms_bag.print_bag()
```

_____

```
>>> s = toms_bag.take_skittle()
>>> print(s.color)
```

```
_____________________________

>>> toms_bag.number_sold

_____________________________

>>> Bag.number_sold

_____________________________

>>> akis_bag = Bag()
>>> akis_bag.print_bag()

_____________________________

>>> toms_bag.print_bag()

_____________________________

>>> Bag.number_sold

_____________________________

>>> akis_bag.number_sold

_____________________________

>>> toms_bag.number_sold

_____________________________
```

2. What type of variable is skittles? What type of variable is number_sold?

3. Write a new method for the Bag class called take_color, which takes a color and removes (and returns) a skittle of that color from the bag. If there is no skittle of that color, then it returns None.

4. We now want to write three different classes Postman, Client, and Email to simulate email. Fill in the definitions below to finish the implementation.

```
class Postman(object):
    """Each Postman has an instance variable 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."""
        """Your code here."""




    def register_client(self, client, client_name):
        """Takes a client object and client_name and adds it to the
        clients instance variable.
        """
        """Your code here."""




class Email(object):
    """Every email object has 3 instance variables: the message, the
    sender (their name), and the addressee (their name).
    """
    def __init__(self, msg, sender, addressee):
        """Your code here."""




class Client(object):
    """Every Client has instance variables 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 = []
        """Your code here."""

    def compose(self, msg, recipient):
        """Send an email with the given message msg to the given
        recipient."""
        """Your code here."""




    def receieve(self, email):
        """Take an email and add it to the inbox of this client."""
        """Your code here."""
```

Inheritance

Now consider writing Dog and Cat classes, you can imagine that they'd both have name, age, and owner instance variables, and also eat and talk methods. That's a lot of effort for writing the same code! This is where Inheritance steps in. In Python, you can create a class and have it inherit the instance variables and methods of a **parent** class without typing it all out again. All of our classes thus far have been inheriting from the **object** class. They are **children** of the object class. Object is the top-level, generic mack-daddy of all classes. It provides basic functionality for all objects, (it's subtle). This is an example of **Code reusability**, the idea that you shouldn't reinvent the wheel if at all possible.

When do you want to inherit? The rule-of-thumb is when there is a hierarchical relationship between two classes, where one is a type or sub-categorization of the other. This is commonly know as a "is a" relationship. A truck "is a" type of vehicle and thus could be a child class of a vehicle class. Make sure you don't get this confused with "has a" relationship. A truck has a color, and therefore color would be an instance variable of truck, not a child class.

Python has some particular syntax when it comes to inheritance. Take a look at this partial implementation of animals:

```
current_year = 2012

class Animal(object):
    def __init__(self):
        self.is_alive = True # It's alive!!!

class Pet(Animal):
```

```
    def __init__(self, name, year_of_birth, owner=None):
        Animal.__init__(self) # call the parent's constructor
        self.name = name
        self.age = current_year - year_of_birth
        self.owner = owner

    def eat(self, thing):
        print(self.name + " ate a " + thing + "!")
    def talk(self):
        print("...")

class Dog(Pet):
    def __init__(self, name, yob, owner, color):
        Pet.__init__(self, name, yob, owner)
        self.color = color

    def talk(self):
        print("Woof!")
```

What does the following code do?

```
>>> fido = Dog('Fido', 1993, 'Joe', 'golden')
>>> clifford = Dog('Clifford', 1963, 'Emily', 'red')
>>> fido.age
```

_____

```
>>> fido.talk()
```

_____

```
>>> fido.owner
```

_____

```
>>> clifford.owner
```

_____

```
>>> clifford.color
```

_____

```
>>>clifford.eat('bone')
```

_____

Now write a Cat class that inherits from Pet. Use parent methods wherever possible:

```
class Cat(Pet):
    def __init__(self, name, yob, owner, lives=9):




    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 0, the 'is_alive' variable becomes False
```

If you've defined Cat correctly, we should be able to build upon it!

```
class Schrodingers_Cat(Cat):
    def __init__(self):
        Cat.__init__(self, "Schrodinger", 1935, 9999)

    def peek(self):
        self.is_alive = not self.is_alive
        return self.is_alive
```