OBJECT ORIENTED PROGRAMMING 8

COMPUTER SCIENCE 61A

July 17, 2014

1 Overview

Object Oriented Programming (OOP) is useful programming paradigm that is a natural way of grouping characteristics and functionality. The beauty of object oriented programming lies in its reliance on **data abstraction**. Just look around you! The chalkboard, your pencil, and your backpack are all objects. We can use these objects without knowing how exactly they are made or what they are made of because we trust the abstraction. Think of objects as how you would an object in real life.

For our example, let's think of your laptop. First of all, it must have gotten its design from somewhere and that blueprint is called a **class**. The laptop itself is an **instance** of that class. If your friend has the same laptop as you, those laptops are just different instances of the same class.

Now, your laptop does stuff (e.g. turn on, display text, etc). Those are called **methods**. It also has properties (screen resolution, how much memory it has, and that scratch mark you hope no one else sees). Those are called attributes. If it's an attribute that's the same for all instances and shared by all instances, it's called a **class attribute** or **class variable**. So, if you were wondering how many instances of your laptop exists, that would be a class attribute because no matter which instance got asked that, it would be the same. If you were wondering how many scratches your laptop has, that's an **instance attribute** or **instance variable** because that number depends on each instance.

So, that's the vocabulary of OOP (yes, people say that – it's quite fun!). As a bonus warm-up, you should say it too.

1.1 Defining a Class

When defining a class, we use the following syntax:

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

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). When the `ParentClass` field is missing (an empty set of parens, or no parens), classes by default inherit from Python's built-in `object` class.

1.2 Defining a Method

To define a method, we write it almost exactly the same way as when we define functions. However, the first argument is **always** `self`, which we can use to refer to the instance we used to call the method. Without `self`, we don't know the context in which to evaluate the method.

```
class OurClass:
    def class_method(self, arg):
        """function body goes here"""
```

1.3 Using a Class or Its Attributes

Finally, to use a class or instance's attributes, we use "dot notation", which is aptly named for the use of the magic dot. The dot asks for the value of the attribute. So, if we have an attribute, `bar`, of a class or instance, `foo`, we access it by saying: "`foo.bar`" which says "Almighty `foo`, what is the value of the attribute `bar`?" Dot notation also is a way to bind attributes to instances or classes. You might be wondering what happens if we forget to add `self` in `self.bar = bar_name`. Then `bar` would be a local variable just as it would in any other function definition! The trusty old rules still apply.

```
class OurClass:
    bar = "Fruit Bar" #class attribute

    def __init__(self, bar_name):
        self.bar = bar_name #instance attribute
    def class_method(self, arg):
        """function body goes here"""
    def class_method2(self):
        """function body goes here"""
```

1.4 Skittles Example

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

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

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

In this example, we have the attribute `number_sold`, which is a class attribute. Also, you see this strange method called `__init__`. That is called when you make a new instance of the class. So, if you write `a = Bag()`, that makes a new instance of the Bag class (calling `__init__` to do so) and then returns `self`, which you can think of as a dictionary that holds all of the attributes of the object.

To make a new class attribute, you use the name of the class with dot notation. For example, `Bag.new_var = 10` makes a new class attribute `new_var` in the `Bag` class and assigns it the value of `10`. To make a new instance attribute, you use the name of the instance attribute: `a.new_var2 = 10`. Attribute lookup works similarly to environment diagrams. You look to see if some instance attribute has that name. If it doesn't, then you look up the name in the class attributes. If it is not defined in the current class, look in the parent class attributes. This step is continued recursively until you find the attribute.

2 Questions

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

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

Solution:

```
[]
```

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

Solution:

```
['blue']
```

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

Solution:

```
['blue', 'red', 'green', 'red']
```

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

Solution:

```
blue
```

```
>>> johns_bag.number_sold
```

Solution:

```
1
```

```
>>> Bag.number_sold
```

Solution:

```
1
```

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

Solution:

```
[]
```

```
>>> johns_bag.print_bag()
```

Solution:

```
['red', 'green', 'red']
```

```
>>> Bag.number_sold
```

Solution:

```
2
```

```
>>> soumyas_bag.number_sold
```

Solution:

2

```
>>> johns_bag.number_sold
```

Solution:

2

2. What type of attribute is `skittles`? What type of attribute is `number_sold`?

Solution: `skittles` - instance attribute

`number_sold` - class attribute

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`.

```
def take_color(self, color):
```

Solution:

```
        for i in range(len(self.skittles)):
            curr_skittle = self.skittles[i]
            if curr_skittle.color == color:
                self.skittles = self.skittles[:i] + \
                                self.skittles[(i + 1):]
                return curr_skittle
        return None # optional; not returning anything
                    # is the same as returning None
```

4. Write a new method for the `Bag` class called `take_all`, which takes all the `Skittles` in the current bag and prints the color of the `Skittle` every time one is taken from the bag.

```
def take_all(self):
```

Solution:

```
        for i in range(len(self.skittles)):
            print(self.take_skittle().color)
```

5. 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 Email:
    """Every email object has 3 instance attributes: the message, the
    sender (their name), and the addressee (the destination's name).
    """
    def __init__(self, msg, sender, addressee):
```

Solution:

```
        self.msg = msg
        self.sender = sender
        self.addressee = addressee
```

```
class Postman:
    """Each Postman has an instance attribute clients, which is a
    dictionary that associates client names with client objects.
    """
    def __init__(self):
        self.clients = dict()

    def send(self, email):
        """Take an email and put it in the inbox of the client it is
        addressed to."""
```

Solution:

```
        client = self.clients[email.addressee]
        client.receive(email)
```

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

Solution:

```
        self.clients[client_name] = client
```

```
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 = list()
```

Solution:

```
        self.mailman = mailman
        self.name = name
        self.mailman.register_client(self, self.name)
```

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

Solution:

```
        email = Email(msg, self.name, recipient)
        self.mailman.send(email)
```

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

Solution:

```
        self.inbox.append(email)
```

3 Inheritance & Interfaces

Inheritance is useful because many objects are just more specific versions of other classes. For example, an ostrich is a more specific version of a bird and a bird is a more specific version of an animal! The subclass inherits attributes from its parent class but it can override attributes, including methods. This allows programmers to write less code because only the differences between the specific and more general class need to be specified.

We say that all animals implement the interface that it can eat and make a sound to communicate. An interface is a collection of attributes and conditions that these attributes must satisfy. This is implemented in Python by sharing the same name amongst different classes. This is useful for creating abstraction by not depending on the implementation.

Inheritance and interfaces highlight two important object relationships. Inheritance demonstrates the is-a relationship. For example, every ostrich is a bird, but not every bird is an ostrich. Interfaces demonstrate the has-a relationship. For example, every animal has a `call` and `eat` method. Let's look at an example:

```
class Animal: # class
    """An Animal object that makes a sound and can eat."""
    population = 0 # class attribute
    eats = True # class attribute
    def __init__(self, sound):  # method
        self.sound = sound # instance attribute
        self.food_count = 0 # instance attribute
        Animal.population += 1 # class attribute
    def call(self): # method
        return self.sound # instance attribute
    def eat(self, food): # method
        self.food_count += 1 # instance attribute
        print("nom nom nom " + food)
```

Every `Animal` can eat and can use sound to communicate. However, we want to make a bird, so we define a `Bird` class!

```
class Bird(Animal):
    """A class that inherits from Animal and can fly."""
    fly = True
    feathers = True
    def __init__(self, color, tune="Chirp chirp."):
        Animal.__init__(self, tune) # call to parent class
        self.color = color
    def call(self):
        return self.sound + " Look at my " + self.color + " feathers!"
```

Notice that we override the `call` method to proclaim the beauty of our feathers. Methods may be overridden, but we can still access the original methods in the parent class, such as with `Animal.__init__(self, tune)`. We can also add new attributes that are specific to birds like feathers and the ability to fly! The `eat` method is not overridden, thus we call the original one defined in the `Animal` class.

An Exercise: Label the class definition below like the `Animal` class was labelled.

```
class Ostrich(Bird):
    """A class that inherits from Bird which inherits from Animal and
    cannot fly.
    """
    fly = False
    def __init__(self, color):
        Bird.__init__(self, color, "Hiss hiss.")
    def call(self):
        tune = Bird.call(self)
        return tune + " I wish I could fly."
```

We can also override class attributes! Notice that the `fly` attribute in the `Ostrich` class is set to `False` because ostriches cannot fly. Let's test your understanding with some questions:

```
>>> oscar = Ostrich("brown")
>>> olivia = Ostrich("grey")
>>> tweetie = Bird("yellow")
>>> oscar.call()
```

Solution:

```
'Hiss hiss. Look at my brown feathers! I wish I could fly.'
```

```
>>> olivia.eat("food")
```

Solution:

```
nom nom nom food
```

```
>>> oscar.food_count
```

Solution:

```
0
```

```
>>> oski = Animal("RAWR!")
>>> oski.eat("spirit")
```

Solution:

```
nom nom nom spirit
```

```
>>> oski.fly
```

Solution:

```
AttributeError: 'Animal' object has no attribute 'fly'
```

4 Dots, Methods and Currying... oh my!

In a class method, you probably noticed that the first argument is always this mysterious "`self`". And somehow, we never seem to have to pass in the argument "`self`" when we're calling it. This is the power of the magic dot. It tells us that "`self`" is the instance that's before the dot, and the method acts as a **bound method**. However, if it happens to be the name of the class, then the method is called as a **function** and `self` isn't an automatic argument.

As to what this has to do with currying.... try the questions and see. Let's start with the `Skittles` and `Bag` classes above.

1. Consider the following code and fill in what Python would print out.

```
>>> bag1 = Bag()
>>> def curried(f):
...     def outer(instance):
...         def inner(*args):
...             return f(instance, *args)
...         return inner
...     return outer
```

```
>>> add_binding = curried(Bag.add_skittle)
>>> bag1_add = add_binding(bag1)
>>> bag1.print_bag()
```

Solution:

```
[]
```

```
>>> bag1.add_skittle(Skittle("blue"))
>>> bag1.print_bag()
```

Solution:

```
['blue']
```

```
>>> bag1_add(Skittle("red"))
>>> bag1.add_skittle(Skittle("green"))
>>> bag1_add(Skittle("red"))
>>> bag1.print_bag()
```

Solution:

```
['blue', 'red', 'green', 'red']
```

```
>>> s = bag1.take_skittle()
>>> bag2 = Bag()
>>> bag2_add = add_binding(bag2)
>>> bag2.print_bag()
```

Solution:

```
[]
```

```
>>> bag2_add(Skittle("blue"))
>>> bag1.print_bag()
```

Solution:

```
['red', 'green', 'red']
```

```
>>> bag2.print_bag()
```

Solution:

```
['blue']
```

5 Summary

So, in summary, object oriented programming is just another helpful tool in abstraction.

The important terms are here again:

- **Class**: A basic skeleton that let's you create multiple instance objects that all behave similarly.
- **Attribute**: A property of the class and its instances. There are two types of attributes:
  - **Instance Attributes**: These are unique to every instance. Changes to one instance don't affect the rest.
  - **Class Attributes**: These are shared by every instance of a class. Changes affect all instances.
- **Method**: A class attribute that is a function. These have a special evaluation rule when accessed with a dot expression.
- **Instance**: An object created from a class.