OBJECT ORIENTED PROGRAMMING 7

COMPUTER SCIENCE 61A

March 6, 2013

1 Overview

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 (OOP) is heavily based on the idea of data 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 (turn on, display text, etc). Those are called **methods**. It also has properties (screen resolution, how much memory it has, that scratch mark you hope no one else sees). Those are called **variables**. If it's a variable that's the same for all instances, it's called a **class variable**. So, if you were wondering how many instances of your laptop exists, that would be a class variable 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 variable** because that number depends on each instance. Variables and methods are called **attributes**.

So, that's the vocabulary of OOP (yes, people say that- it's quite fun!). As a bonus warmup, 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 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).

1.2 Defining a Method

To define a method, we write it almost exactly the same way as when we define functions but the first argument we always include is `self`, which we use to refer to the instance we used to call the method.

```
class OurClass(ParentClass):
    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 the class for the value of the attribute. So, if we have a method or variable, `bar`, of a class or instance, `foo`, we access it by saying: "`foo.bar`" which says "Almighty `foo` class, what is the value of the attribute `bar`?"

```
class OurClass(ParentClass):
    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 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(tuple(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 += (s,)
```

In this example, we have the variable `number_sold`, which is a class variable. 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 the `self` variable, which you can think of as a dictionary that holds all of the attributes of the object.

To make a new class variable, you use the name of the class with dot notation: `Bag.new_var = 10` makes a new class variable `new_var` in the Bag class and assigns it the value of `10`. To make a new instance variable, you use the name of the instance variable: `a.new_var2 = 10`. Variable lookup works similarly to environment diagrams. You look to see if the instance variable has the variable name. If it doesn't, then you look in the list of class variables.

2 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. However, if it happens to be the name of the class, then it's the method itself 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')
```

3 Questions

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

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

Solution:

```
()
```

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

Solution:

```
('blue',)
```

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

Solution:

```
('blue', 'red', 'green', 'red')
```

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

Solution:

```
blue
```

```
>>> amirs_bag.number_sold
```

Solution:

```
1
```

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

Solution:

```
1
```

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

Solution:

```
()
```

```
>>> amirs_bag.print_bag()
```

Solution:

```
('red', 'green', 'red')
```

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

Solution:

```
2
```

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

Solution:

```
2
```

```
>>> amirs_bag.number_sold
```

Solution:

2

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

Solution:

`skittles` - instance variable

`number_sold` - class variable

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].color
            if curr_skittle == 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 a skittle 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(object):
    """Every email object has 3 instance variables: 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(object):
    """Each Postman has an instance variable 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 variable.
        """
```

Solution:

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

```
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 = 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 += email
```