Object-Oriented Programming and Interfaces 6

Computer Science 61A

March 5, 2015

1 Introduction

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.

Your laptop performs many actions, e.g. turning on, displaying text, etc. Those are called **methods**. It also has properties, e.g. screen resolution, how much memory it has, 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, it's called a **class attribute**. 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** 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):
    """Class definition with 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 (i.e. just `class OurClass:`), classes 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 we always include is `self`, which we use to refer to the instance we used to call the method.

```
class OurClass(ParentClass):
    def method(self, arg):
        # body goes here
```

1.3 Using a Class and 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 an attribute, `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`?"

Typically, attributes are defined in the `__init__` function of a class:

```
class OurClass(ParentClass):
    bar = "Fruit Bar" # class attribute
    def __init__(self, bar_name):
        self.bar = bar_name # instance attribute
    def method(self, arg):
        # body goes here
```

Once an object is constructed, you can also access the attribute by using dot notation *outside* of the class definition:

```
>>> new_bar = OurClass('Crazy Bar')
>>> new_bar.bar
'Crazy Bar'
```

1.4 Class vs. Object

When discussing objects and classes, it is helpful to distinguish between the definition of a class and the instantiation of a class, or an object. The instantiation is referred to as an "object", whereas the definition is the "class". Following our example, `OurClass` is a class, while `new_bar` is an instantiation of that class, also referred to as an object.

2 Taste the Rainbow

As a starting example, consider the classes `Skittle` and `Bag`, which will be 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 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).
        """
        return self.skittles.pop(0)
```

```
    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: `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.

2.1 Questions

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

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

Solution:

```
[]
```

```
>>> for color in ['blue', 'red', 'green', 'red']:
...     johns_bag.add_skittle(Skittle(color))
>>> 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

2. 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:
                return self.skittles.pop(i)
        return None # optional; not returning anything
                    # is the same as returning None
```

3. Write a new method for the `Bag` class called `take_all`, which takes all the `Skittle`s in the current bag and prints the color of the each `Skittle` taken from the bag.

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

Solution:

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

3 Inheritance

Let's explore another powerful object-oriented programming tool: inheritance. Suppose we want to write `Dog` and `Cat` classes. Here's our first attempt:

```
class Dog(object):
    def __init__(self, name, owner, color):
        self.name = name
        self.owner = owner
        self.color = color
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        return self.name + " says woof!"

class Cat(object):
    def __init__(self, name, owner, lives=9):
        self.name = name
        self.owner = owner
        self.lives = lives
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        return self.name + " says meow!"
```

Notice that both the `Dog` and `Cat` classes have `name`, `owner`, `eat` method, and `talk` methods. That's a lot of effort for so much repeated code!

This is where **inheritance** comes in. In Python, a class can **inherit** the instance variables and methods of a another class without having to type them all out again. For example:

```
class Foo(object):
    # This is the superclass

class Bar(Foo):
    # This is the subclass
```

`Bar` inherits from `Foo`. We call `Foo` the **superclass** (the class that is being inherited) and `Bar` the **subclass** (the class that does the inheriting).

Notice that `Foo` also inherits from the `object` class. In Python, `object` is the top-level superclass; everything inherits from it, whether directly or through other superclasses. `object` provides basic functionality that is needed for other classes to work with Python.

3.1 When should we use inheritance?

One common use of inheritance is to represent a hierarchical relationship between two or more classes – one class is a more specific version of the other class. For example, dogs are a specific type of pet, and a pet is a specific type of animal.

Using inheritance, here is a second attempt at representing dogs.

```
class Pet(object):
    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('...')

class Dog(Pet):
    def __init__(self, name, owner, color):
        Pet.__init__(self, name, owner)
        self.color = color
    def talk(self):
        print('woof!')
```

Notice that, by using inheritance, we did not have to redefine `self.name`, `self.owner`, or the `eat` method. We did, however, redefine the `talk` method in the `Dog` class. In this case, we want `Dog`s to `talk` differently, so we **override** the method.

The line `Pet.__init__(self, name, owner)` in the `Dog` class is necessary for inheriting the instance attributes `self.is_alive`, `self.name`, and `self.owner`. Without this line, `Dog` will never inherit those instance attributes. Notice that when we call `Pet.__init__`, we need to pass in `self`, since `Pet` is a class, not an instance.

3.2 Questions

1. Implement the `Cat` class by inheriting from the `Pet` class. Make sure to use superclass methods wherever possible. In addition, add a `lose_life` method to the `Cat` class.

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

Solution:

```
        Pet.__init__(self, name, owner)
        self.lives = lives
```

```
    def talk(self):
        """A cat says meow! when asked to talk."""
```

Solution:

```
        print('meow!')
```

```
    def lose_life(self):
        """A cat can only lose a life if they have at least
        one life. When lives reach zero, the 'is_alive'
        variable becomes False.
        """
```

Solution:

```
        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 :(")
```

2. Assume these commands are entered in order. What would Python output?

```
>>> class Foo(object):
...     def __init__(self, a):
...         self.a = a
...     def garply(self):
...         return self.baz(self.a)
>>> class Bar(Foo):
...     a = 1
...     def baz(self, val):
...         return val
>>> f = Foo(4)
>>> b = Bar(3)
>>> f.a
```

Solution: `4`

```
>>> b.a
```

Solution: `3`

```
>>> f.garply()
```

Solution: `AttributeError: 'Foo'object has no attribute 'baz'`

```
>>> b.garply()
```

Solution: `3`

```
>>> b.a = 9
>>> b.garply()
```

Solution: `9`

```
>>> f.baz = lambda val: val * val
>>> f.garply()
```

Solution: `16`

3.3 Extra Questions

1. More cats! Fill in the methods for `NoisyCat`, which is just like a normal `Cat`. However, `NoisyCat` talks a lot, printing twice whatever a `Cat` says.

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

Solution:

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

```
    def talk(self):
        """Repeat what a Cat says twice."""
```

Solution:

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

4 Interfaces

In computer science, an **interface** is a shared set of attributes, along with a specification of the attributes' behavior. For example, an interface for vehicles might consist of the following methods:

- `def drive(self):` Drives the vehicle if it has stopped.
- `def stop(self):` Stops the vehicle if it is driving.

Data types can implement the same interface in different ways. For example, a `Car` class and a `Train` can both implement the interface described above, but the `Car` probably has a different mechanism for `drive` than the `Train`.

The power of interfaces is that other programs don't have to know *how* each data type implements the interface – only that they *have* implemented the interface. The following `travel` function can work with both `Car`s and `Train`s:

```
def travel(vehicle):
    while not at_destination():
```

```
        vehicle.drive()
    vehicle.stop()
```

4.1 Interfaces in Python

Python defines many interfaces that can be implemented by user-defined classes. For example, the interface for arithmetic consists of the following methods:

- `def __add__(self, other)`: Allows objects to do `self + other`.
- `def __sub__(self, other)`: Allows objects to do `self - other`.
- `def __mul__(self, other)`: Allows objects to do `self * other`.

In addition, there is also an interface for sequences:

- `def __len__(self)`: Allows objects to do `len(self)`.
- `def __getitem__(self, index)`: Allows objects to do `self[i]`.

4.2 Questions

Let's implement a `Vector` class that support basic operations on vectors. These include adding and subtracting vectors of the same length, multiplying a vector with a scalar, and taking the dot product of two vectors. The results of these operations are shown in the table below:

Operation	Result
`-Vector([1, 2, 3])`	`Vector([-1, -2, -3])`
`Vector([1, 2, 3]) + Vector([4, 5, 6])`	`Vector([5, 7, 9])`
`Vector([4, 5, 6]) - Vector([1, 2, 3])`	`Vector([3, 3, 3])`
`Vector([1, 2, 3]) * Vector([1, 2, 3])`	`14`
`Vector([1, 2, 3]) * 4`	`Vector([4, 8, 12])`
`10 * Vector([1, 2, 3])`	`Vector([10, 20, 30])`
`len(Vector([1, 2, 3]))`	`3`
`Vector([1, 2, 3])[1]`	`2`

We begin with an implementation of the `Vector` class:

```
class Vector:
    def __init__(self, vector):
        self.vector = vector

    def __neg__ (self)        : "*** YOUR CODE HERE ***"
    def __add__ (self, other): "*** YOUR CODE HERE ***"
    def __sub__ (self, other): return self.__add__(-other)
```

```
    def __mul__ (self, other): "*** YOUR CODE HERE ***"
    def __rmul__(self, other): return self.__mul__(other)
    def __len__(self)        : return len(self.vector)
    def __getitem__(self, n) : return self.vector[n]
```

1. Implement `__neg__`, which returns a new `Vector` that is the negation of the current vector, `self`. Try using list comprehensions.

```
def __neg__(self):
    return Vector(_____________________________________________)
```

Solution:

```
def __neg__(self):
    return Vector([-i for i in self.vector])
```

2. Implement `__add__`, which takes in two vectors of the same length and returns a new vector which is their sum. Try using list comprehensions.

```
def __add__(self, other):
    assert type(other) == Vector, "Invalid operation!"
    assert len(self) == len(other), "Invalid dimensions!"
```

Solution:

```
    return Vector([self[n] + other[n] for n in range(len(
        self))])
```

3. Implement `__mul__`, which takes in a value, and performs a scalar product if the value is a number, or a vector product if the value is another vector.

```
def __mul__(self, other):
    if type(other) == int or type(other) == float:
        "*** YOUR CODE HERE ***"
```

Solution:

```
        return Vector([v * other for v in self.vector])
```

```
    elif type(other) == Vector:
        "*** YOUR CODE HERE ***"
```

Solution:

```
        assert len(self) == len(other), "Invalid dimensions
           !"
        return sum([self[n] * other[n]
                    for n in range(len(self))])
```