OBJECT ORIENTED PROGRAMMING 7

DATA C88C

July 11, 2024

1 Object Oriented Programming

1.1 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.

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.

1.2 Methods

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 Classes

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.

1.4 Dot Notation

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'
```

1.5 ADTs and OOP

1. What is the relationship between a class and an ADT?

Solution: In general, we can think of an abstract data type as defined by some collection of selectors and constructors, together with some behavior conditions. As long as the behavior conditions are met (such as the division property above), these functions constitute a valid representation of the data type.

There are two different layers to the abstract data type:

1) The program layer, which uses the data, and
2) The concrete data representation that is independent of the programs that use the data. The only communication between the two layers is through selectors and constructors that implement the abstract data in terms of the concrete representation.

Classes are a way to implement an Abstract Data Type. But, ADTs can also be created using a collection of functions, as shown by the rational number example. (See Composing Programs 2.2)

2. What is the definition of a Class? What is the definition of an Instance?

Solution: Class: a template for all objects whose type is that class that defines attributes and methods that an object of this type has.
Instance: A specific object created from a class. Each instance shares class attributes and stores the same methods and attributes. But the values of the attributes are independent of other instances of the class. For example, all humans have two eyes but the color of their eyes may vary from person to person.

3. What is a Class Attribute? What is an Instance Attribute?

Solution: Class Attribute: A static value that can be accessed by any instance of the class and is shared among all instances of the class.
Instance Attribute: A field or property value associated with that specific instance of the object.

4. What Would Python Display?

```
class Foo():
    x = 'bam'
    def __init__(self, x):
        self.x = x
    def baz(self):
        return self.x

class Bar():
        x = 'boom'
        def __init__(self, x):
                Foo.__init__(self, 'er' + x)
        def baz(self):
                return Bar.x + Foo.baz(self)
>>> foo = Foo('boo')
>>> Foo.x
```

Solution: 'bam'

```
>>> foo.x
```

Solution: 'boo'

```
>>> foo.baz()
```

Solution: 'boo'

```
>>> Foo.baz()
```

Solution: Error

```
>>> Foo.baz(foo)
```

Solution: 'boo'

```
>>> bar = Bar('ang')
>>> Bar.x
```

Solution: 'boom'

```
>>> bar.x
```

Solution: 'erang'

```
>>> bar.baz()
```

Solution: 'boomerang'

1.6 OOP Questions

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.

5. 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
```

6. 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 each `Skittle` taken from the bag.

```
def take_all(self):
    """
    >>> my_bag = Bag()
    >>> for color in ['red', 'orange', 'yellow']:
    ...     my_bag.add_skittle(Skittle(color))
    >>> my_bag.print_bag()
    ['red', 'orange', 'yellow']
    >>> my_bag.take_all()
    red
    orange
    yellow
    >>> my_bag.print_bag()
    []
    """
```

Solution:

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

7. 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):
    """
    >>> my_bag = Bag()
    >>> for color in ['red', 'orange', 'yellow']:
    ...     my_bag.add_skittle(Skittle(color))
    >>> my_bag.print_bag()
    ['red', 'orange', 'yellow']
    >>> my_skittle = my_bag.take_color('red')
    >>> my_skittle.color
    red
    >>> my_bag.print_bag()
    ['orange', 'yellow']
    >>> none_skittle = my_bag.take_color('blue')
    >>> none_skittle
    >>> my_bag.print_bag()
    ['orange', 'yellow']
    """
```

Solution:

```
        for i in range(len(self.skittles)):
            curr_skittle = self.skittles[i].color
            if curr_skittle == color:
                return self.skittles.pop(i)
```