Object oriented programing

Review by: Shahabedin Chatraee Azizabadi

Definition:

Object-oriented programming(OOP) can be seen as an approach for modeling concrete, real-world objects as well as relations between things like employers and employees, students and teachers, etc. OOP models real-world entities as software objects, which have some data associated with them and at the same time can perform certain functions.

Another common programming paradigm is *procedural programming* which structures a program like a recipe in that it provides a set of steps, in the form of functions and code blocks, which flow sequentially in order to complete a task.

The key takeaway is that objects are at the center of the object-oriented programming paradigm, not only representing the data, as in procedural programming, but in the overall structure of the program as well.

NOTE: Since Python is a multi-paradigm programming language, you can choose the paradigm that best suits the problem at hand, mix different paradigms in one program, and/or switch from one paradigm to another as your program evolves.

Classes in Python

Focusing first on the data, each thing or object is an instance of some *class*.

The primitive data structures available in Python, like numbers, strings, and lists are designed to represent simple things like the price of something, the name of a poem, and your favorite colors, respectively.

How would you know which element is supposed to be which? This lacks organization, and it's the exact need for *classes*. Classes are used to create new user-defined data structures that contain arbitrary information about something. In the case of an animal, we could create an Animal () class to track properties about the Animal like the name and age.

It's important to note that a class just provides structure—it's a blueprint for how something should be defined, but it doesn't actually provide any real content itself. It may help to think of a class as an idea for how something should be defined.

Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Python Objects (Instances)

While the class is the blueprint, an *instance* is a copy of the class with *actual* values, literally an object belonging to a specific class. It's not an idea anymore. Put another way, a class is like a form or questionnaire. It defines the needed information. After you fill out the form, your specific copy is an instance of the class; it contains actual information relevant to you.

You can fill out multiple copies to create many different instances, but without the form as a guide, you would be lost, not knowing what information is required. Thus, before you can create individual instances of an object, we must first specify what is needed by defining a class.

An object has two characteristics:

- attributes
- behavior

Defining class in python

You start with the class keyword to indicate that you are creating a class, then you add the name of the class.

Instance attributes

All classes create objects, and all objects contain characteristics called attributes (referred to as properties in the opening paragraph). Use the __init__() method to initialize (e.g., specify) an object's initial attributes by giving them their default value (or state). This method must have at least one argument as well as the self-variable, which refers to the object itself.

Remember: the class is just for defining the object, not actually creating *instances* of individual objects with specific names and ages, etc. Similarly, the self-variable is also an instance of the class. Since instances of a class have varying values we could state.

NOTE: You will never have to call the __init__() method; it gets called automatically when you create a new 'Class' instance.

Class Attributes

While instance attributes are specific to each object, class attributes are the same for all instances.

Instantiating Objects

Instantiating is a fancy term for creating a new, unique instance of a class.

These attributes are passed to the __init__ method, which gets called any time you create a new instance, attaching the name and age to the object. You might be wondering why we didn't have to pass in the self-argument.

This is Python magic; when you create a new instance of the class, Python automatically determines what self is and passes it to the __init__ method.

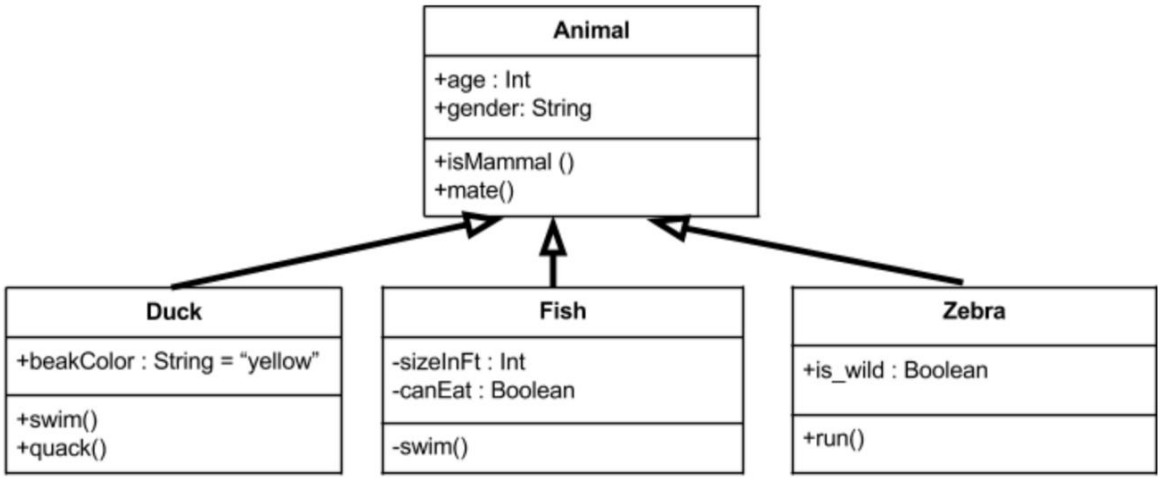
Instance Methods

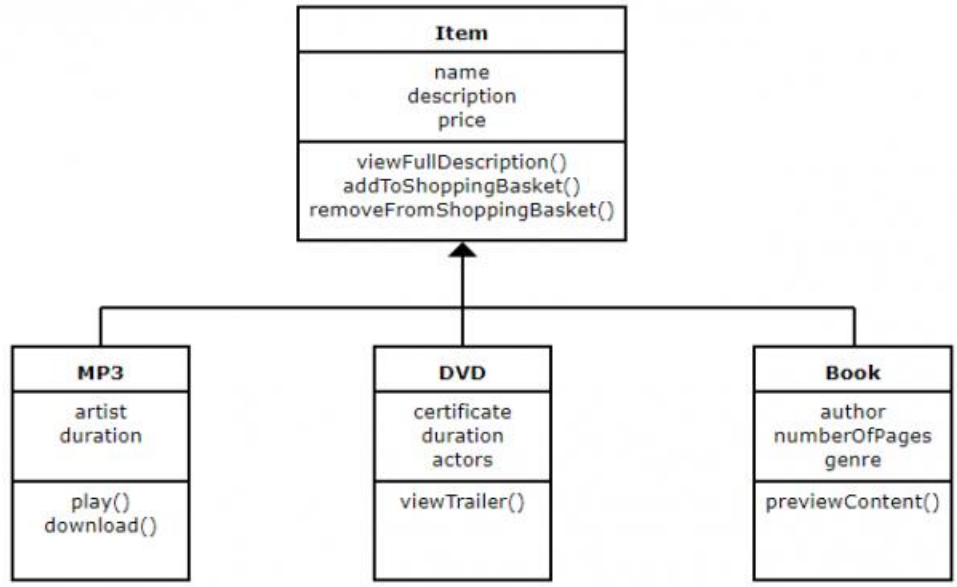
Instance methods are defined inside a class and are used to get the contents of an instance. They can also be used to perform operations with the attributes of our objects. Like the __init__ method, the first argument is always self.

Python Object Inheritance

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called *child classes*, and the classes that child classes are derived from are called *parent classes*.

It's important to note that child classes override *or* extend the functionality (e.g., attributes and behaviors) of parent classes. In other words, child classes inherit all of the parent's attributes and behaviors but can also specify different behavior to follow. The most basic type of class is an object, which generally all other classes inherit as their parent.





Parent vs. Child Classes

The isinstance() function is used to determine if an instance is also an instance of a certain parent class.

Overriding the Functionality of a Parent Class

Remember that child classes can also override attributes and behaviors from the parent class.

Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e. single "_ "or double "__ ".

Polymorphism

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However, we could use same method to color any shape. This concept is called Polymorphism.

***Super ()* function in class**

Python super function can refer to the superclass implicitly. So, the Python super () function makes our task more manageable.

While referring to the superclass from the base class, we don't need to write the name of the superclass explicitly.

```
class MyParentClass():
    def __init__(self):
        pass
class SubClass(MyParentClass):
    def __init__(self):
        super()
```

We can see that there's the base or parent class (also sometimes called the superclass) and derived class or subclass, but we still need to initialize the parent or base class within the subclass or derived or child. We can call the super() function to process more accessible. The goal of Super function is to provide a much more abstract and portable solution for initializing classes.

```
class Computer():
    def __init__(self, computer, ram, ssd):
        self.computer = computer
        self.ram = ram
        self.ssd = ssd
class Laptop(Computer):
```

```
def __init__(self, computer, ram, ssd, model):
    super().__init__(computer, ram, ssd)
    self.model = model
```