# Object Oriented Programming

## Abstract

Object-oriented Programming, or **OOP** for short, mean structuring programs so that properties and behaviors are bundled into individual objects.

For instance, an object could represent a person with a name, property, age, address, etc., with behaviors like walking, talking, breathing, and running. Or an email with properties like recipient list, subject, body, etc., and behaviors like adding attachments and sending.

Simply put, object-oriented programming is an approach for modeling concrete, real-world things like cars and relations between things like companies and employees, students and teachers, etc. OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.

## I. Classes

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 cost of something, the name of a poem, and your favorite colors, respectively. But what if you wanted to express something much more complicated?

For example, let's say you wanted to track the number of different animals. If you used a list, the first element could be the animal's name, while the second element could represent its age. How would you know which element is supposed to be which? What if you had 100 different animals? Are you certain each animal has both a name and an age, and so forth? What if you wanted to add other properties to these animals? 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` such as its name and age.

It's important to note that **a class provides structure**—it's a blueprint for how something should be defined, but it doesn't offer any actual content itself. The `Animal` class may specify that the name and age are necessary for determining an animal, but it will not state what a specific animal's name or age is. It may help to think of a class as an idea for how something should be defined.

## II. 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; it's a real animal, like a dog named Roger eight years old.

Put another way, a class is like a form or a questionnaire. It defines the needed information. After you fill out the form, your specific copy is an instance of the class; it contains basic 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, you must first specify what is needed by defining a class.

## III. Defining A Class

Defining a class is simple in Python:

```
class Dog():
    pass
```

- You start with the `class` keyword to indicate that you are creating a class,
- then add the name of the class (using *CamelCase* notation, starting with a capital letter.)

Also, we used the Python keyword `pass` here. This is very often used as a placeholder where code will eventually go. It allows us to run this code without throwing an error.

## IV. Creating An Object (Instance)

As we said, class is only a blueprint.

To create an object or an instance, you want to *call* the class of this class.

```
class Dog():
    pass

shelter_dog = Dog()
```

Here we created `shelter_dog`, an **object of the class** `Dog`.

## V. Attributes

Even in real life, all objects have attributes, and for example, dogs have **height**, **color** and **race**, we can implement this in our classes, so all the objects of the same class have the same attributes (though not the same values for those attributes).

Attributes are like variables; they can be any data type, the only difference is that they belong to an object.

To target an attribute, you need to **refer to the object** (`shelter_dog`), followed by a dot `.` and the name of the attribute. For example

```
shelter_dog.color
```

This will refer to the `color` attribute of the dog. Of course, when you target an object's attribute, you need to target an existing attribute, else you will have an error.

To define a new attribute (or modify an existing one), target it and assign it to his new value with the equal sign `=`.

```
class Dog():
    pass

shelter_dog = Dog()
shelter_dog.color = "Brown"
```

## VI. The `__init__` Method

When an object is created, **python automatically runs** the `__init__()` (it has to be called that) method of the class.

This method must have at least one argument, `self` (it doesn't have to be called `self,` but a python convention).

- **`self` refers to the object itself.**

```
class Dog():

    # Initializer / Instance Attributes
    def __init__(self):
        print("A new dog has been initialized !")

shelter_dog = Dog()
```

Although this function receives one argument (`self`), we don't need to pass it, **it will be passed automatically by python as the first argument.**

You can add arguments to `__init__`. Those arguments would be passed on the object creation (`shelter_dog = Dog()`).

For example:

```python
class Dog():
    # Initializer / Instance Attributes
    def __init__(self, name_of_the_dog):
        print("A new dog has been initialized !")
        print("His name is", name_of_the_dog)

shelter_dog = Dog(name_of_the_dog="Rex")
# or
shelter_dog = Dog("Rex")
```

Here we are passing only one argument (`"Rex"`), but in fact, two values are passed, the `name_of_the_dog` and `shelter_dog` itself, as `self`. In fact, this is what python is running:

```python
shelter_dog = Dog(shelter_dog, "Rex")
```

This argument is passed *implicitly*.

Most of the time, you want to initialize the attributes of an object on his creation (like a newborn dog would have some initial color, race, and height..).

To do so, you can pass arguments to the `__init__()` function and then initialize attributes in the function.

Remember that to assign a value to an attribute, we need to select the attribute with `object.attribute` and assign it with the `=` sign. In `__init__` function, the object is referred to as `self`, thus to define an attribute, you need `self.attribute = value`.

```python
class Dog():
    # Initializer / Instance Attributes
    def __init__(self, name_of_the_dog):
        print("A new dog has been initialized !")
        print("His name is", name_of_the_dog)
        self.name = name_of_the_dog

shelter_dog = Dog('Rex')
other_shelter_dog = Dog("Jimmy")
```

Here I have created two different `Dog` objects with two other names.

## Other Example

Create a class named Person, use the `__init__()` function to assign values for name and age:

```python
class Person():
  def __init__(self, name, age):
    self.name = name
    self.age = age

first_person = Person("John", 36)

print(first_person.name)
print(first_person.age)
```

# Instance Methods

Instance methods are defined inside a class and are used to describe a function that belongs to a class. For example, in real life, the “bark” function belongs to “Dog” class.

Instance methods can be used to perform operations with the attributes, get the contents of an instance, and many other things.

To define a method, use the `def` keyword inside the class, like we were doing with the `__init__` method. All instance methods need to receive `self` as the first argument; this allows us to play with the object inside the method.

Let’s define the bark method.

```
class Dog():
    # Initializer / Instance Attributes
    def __init__(self, name_of_the_dog):
        print("A new dog has been initialized !")
        print("His name is", name_of_the_dog)
        self.name = name_of_the_dog

    def bark(self):
        print("{} barks ! WAF".format(self.name))
```

To call an instance method, type the name of the instance (object) followed by a dot `.` and the function’s name. Let’s create a `Dog` object and call the `bark` function.

```
shelter_dog = Dog("Rex")
shelter_dog.bark()
```

The first line will output:

```
>> A new dog has been initialized
>> His name is Rex
```

And the second:

```
>> Rex barks ! WAF
```

After the ‘ self ‘ one, the methods can also receive arguments, which will be passed implicitly.

```
class Dog():
    # Initializer / Instance Attributes
    def __init__(self, name_of_the_dog):
        print("A new dog has been initialized !")
        print("His name is", name_of_the_dog)
        self.name = name_of_the_dog

    def bark(self):
        print("{} barks ! WAF".format(self.name))

    def walk(self, number_of_meters):
        print("{} walked {} meters".format(self.name, number_of_meters))

shelter_dog = Dog("Rex")
```

```python
shelter_dog.walk(10)
```

You can also use instance methods to modify object's attributes. Here is a function that change the dog's name:

```python
class Dog():
    # Initializer / Instance Attributes
    def __init__(self, name_of_the_dog):
        print("A new dog has been initialized !")
        print("His name is", name_of_the_dog)
        self.name = name_of_the_dog

    def bark(self):
        print("{} barks ! WAF".format(self.name))

    def walk(self, number_of_meters):
        print("{} walked {} meters".format(self.name, number_of_meters))

    def rename(self, new_name):
        self.name = new_name

shelter_dog = Dog("Rex")
shelter_dog.rename("Paul")
```

Remember that the `self` keyword refers to the object itself.

## Exercise

Analyse the code below. What will be the output ?

Explain the goal of the methods

Create a method that modifies the name of the person

```python
class Person():
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def show_details(self):
    print("Hello my name is " + self.name)

first_person = Person("John", 36)
first_person.show_details()
```

## Exercise

Analyse the code below. What will be the output ?

```
class Computer():

    def description(self, name):
        """
        This is a totally useless function
        """
        print("I am a computer, my name is", name)
        #Analyse the line below
        print(self)

mac_computer = Computer()
mac_computer.brand = "Apple"
print(mac_computer.brand)

dell_computer = Computer()

Computer.description(dell_computer, "Mark")
# IS THE SAME AS:
dell_computer.description("Mark")
```

## How To Define Classes In Python

To define a class you first need to know why it’s necessary for object-oriented programming. When working on complex programs, in particular, object-oriented programming lets you reuse code and write code that is more readable, which in turn makes it more maintainable.

## What Are Classes?

A class is a user-defined data type which includes local methods and properties.

Classes are an important concept in object-oriented programming.

One of the big differences between functions and classes is that a class contains both data (variables) called properties and methods (functions defined inside a class).

## Example

Now let’s define a class named `Shape`:

```
class Shape:
  sides = 4 #first property
  name = "Square" #second property
  def description(a): #method defined
    return ("A square with 4 sides")

s1 = Shape() #creating an object of Shape
print "Name of shape is:",(s1.name)
print "Number of sides are:",(s1.sides)
print s1.description()
```

## Explanation

The class `Shape` has the following properties:

- `sides`
- `name`

and the following method:

`description`

You might have noticed that the argument passed to the method is the word `self`, which is a reference to objects that are made based on this class. To reference the instance of the class, `self` will always be the first parameter.