towards
data science

Explaining Python Classes in a simple way

Understand the fundamentals of classes using examples

Eugenia Anello 1 day ago · 5 min read★

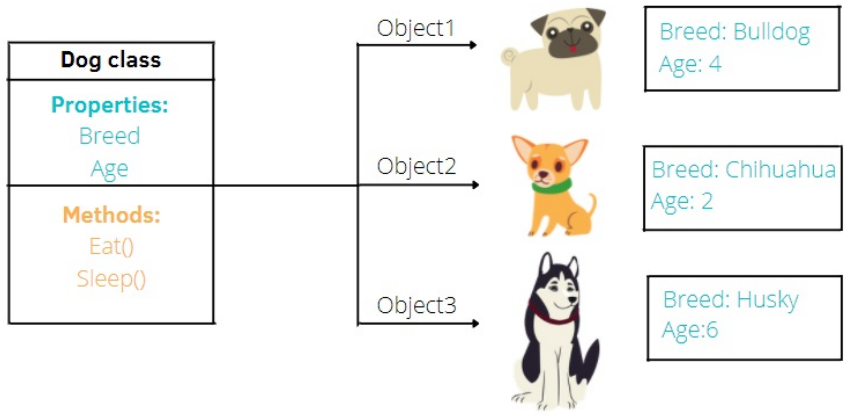

Illustration by Author

When I studied the Python Classes the first time, I found it really complicated and I didn't grasp the reason why they needed to be known. In a university lesson, the professor started to explain directly the syntax to build the classes without explaining the real sense of doing this particular topic and used very boring examples that made me lose in the way.

In this post, I want to explain Python Classes in a different way. I'll start explaining why the classes need to be known and show some of the many cases of already-built classes. Once the external context is clear, the next step is to show step by step how to define classes using some examples and illustrations.

Table of Contents:

. . .

1. Intro to Classes

When you started to study Python, you have surely met the sentence:

> *Python is an object-oriented programming language*

It means that programming in Python leads to being surrounded by objects everywhere. In the instant

we assign a value to a variable, we are creating an object. This **object** belongs to a particular **class** already pre-built, like numbers, strings, lists, dictionaries and so on. Depending on the class, the object will have different **properties** and **methods**.

Let's see some classic examples of objects and classes:

```
a = 28
print(type(a))
#<class 'int'>
```

We defined an object a, which belongs to the Integer class.

```
l = [1,2,3]
print(type(l))
#<class 'list'>
```

This time, the new object belongs to the List class. From the object, we can call methods, already pre-built in Python. To see all the methods allowed in the object, we use the dir function:

```
dir(l)
```

```

```

From the output, we can notice that the class can provide three types of methods:

- **Constructor method** `__init__`, called in this way because it's the method that constructs the object, setting its attributes.
- **Magical methods** are special methods with double underscores on both sides. For example, `__add__` and `__mul__` are used to respectively sum and multiplicate objects of the same class, while `__repr__` returns a representation of the object as a string.
- **Instance methods** are methods that belong to the object created. For example, `l.append(4)` adds an element at the end of the list.

2. Create a Class

Now, we will create an empty class and we'll progressively add parts of code within the tutorial.

```
class Dog:
   pass
```

We created a class with the name Dog, where `pass` is used to indicate that nothing is defined.

```
jack = Dog()
print(type(jack))
#<class '__main__.Dog'>
```

Once we defined the Dog class, we can create an **object of the class**, which is assigned to the variable jack. It's built using a similar notation we use to call a function: `Dog()`.

The object can also be called **instance**. Don't be confused if you find the words "instance" or "object" written somewhere, they always refer to the same thing.

Like before, we check the type of object. The output clearly points out that the object belongs to the Dog class.

3. Constructor method

In addition to the code shown previously, the constructor method `__init__` is used to set the attributes of the Dog class:

```
class Dog:
  def __init__(self,name,breed,age):
    self.Name = name
    self.Breed = breed
    self.Age = age
    print("Name: {}, Breed: {}, Age: {}".format(self.Name,

self.Breed,self.Age))
```

We can observe that the method has different arguments:

- `self` is a standard notation used as the first argument and refers to the object (that will be created later). It's also useful to access the attributes that belong to the class.
- `name`, `breed` and `age` are the remaining arguments. Each argument is used to store the specific **attribute's value** of the object.
- `Name`, `Breed` and `Age` are the defined **attributes**. They usually have the same name of the method's arguments, but I preferred to put the capital letter to distinguish better

```
jack = Dog('Jack','Husky',5)
#Name: Jack, Breed: Husky, Age: 5
print(jack)
#<__main__.Dog object at 0x000002551DCEFFD0>
```

We create again the object, but we also specify the values that correspond to the attributes. If you try to run the code, you'll obtain automatically the line of text shown in the second row of the grey window. It's a good way to check if the class defined is

working well.

4. Magical Method

There is also the possibility to print the same informations in a more sophisticated way. For this purpose, we use the magical method `__repr__`:

```
class Dog:
    def __init__(self,name,breed,age):
        self.Name = name
        self.Breed = breed
        self.Age = age
    def __repr__(self):
        return "Name: {}, Breed: {}, Age: {}".format(self.Name,
                                                     self.Breed,self.Age)
```

The method `__repr__` takes a unique parameter `self` from which it can access the attributes of the object.

```
jack = Dog('Jack','Husky',5)
print(jack)
#Name: Jack, Breed: Husky, Age: 5
```

If we display the new instance created, we can look at the attributes and their respective values.

5. Instance Method


Sleep or Awake? Photo by Paul Trienekens on Unsplash. Photo by Karl Anderson on Unsplash.

The instance methods are methods that belong to the class. As the magical methods, they take an input the parameter `self` to access the attributes of the class. Let's see an example:

```
class Dog:
    def __init__(self,name,breed,age,tired):
        self.Name = name
        self.Breed = breed
        self.Age = age
        self.Tired = tired
    def __repr__(self):
        return "Name: {}, Breed: {}, Age: {}".format(self.Name,
```

```
                                                self.Breed,self.Age)
    def Sleep(self):
        if self.Tired==True:
            return 'I will sleep'
        else:
            return "I don't want to sleep"
```

In the constructor method, we added a new argument `tired` and, consequently, a new attribute `Tired`. After, we define a new method called Sleep: if the attribute's value is equal to True, the dog will sleep, otherwise, it won't.

```
jack = Dog('Jack','Husky',5,tired=False)
print(jack.Sleep())
#I don't want to sleep
```

The dog is not tired, so it won't sleep.

. . .

Final thoughts:

In this post, I provided a fast summary of the Python classes. I hope you found it useful to consolidate your basis about the classes. There are other types of methods I didn't explain, Static and Class Methods, since I want to focus on the most common ones. Moreover, another interesting topic is Class Inheritance, which will be covered in the next post I will write. Thanks for reading. Have a nice day!

Programming Python Class Objects Methods