Theory: Class vs instance

34 minutes 7 / 7 problems solved

By now, you already know that Python makes a distinction between **class attributes** and **instance attributes**. If you recall, class attributes are the ones shared by all instances of the class, while instance attributes are specific for each instance. Moreover, class attributes are defined within the class but outside any methods, and instance attributes are usually defined within methods, notably the `__init__` method.

Now let's go over the difference between class attributes and instance attributes in more detail.

Current topic:

✓ Class vs instance 3★

Topic depends on:

✓ String formatting Stage 1 7★

✓ Program with numbers Stage 1 17★

✓ Class instances Stage 1 3★

§1. Changing attributes

Suppose we have a class `Pet`:

```
class Pet:
    kind = "mammal"
    n_pets = 0  # number of pets
    pet_names = []  # list of names of all pets

    def __init__(self, spec, name):
        self.spec = spec
        self.name = name
        self.legs = 4
```

This class has three class attributes, `kind`, `n_pets`, and `pet_names`, as well as three instance attributes, defined in the `__init__` method, `spec`, `name`, and `legs`.

Let's create instances of that class and see how changing class or instance attributes works in Python:

```
tom = Pet("cat", "Tom")
avocado = Pet("dog", "Avocado")
ben = Pet("goldfish", "Benjamin")
```

We've created three instances of the class `Pet` that have the same class attributes and different instance attributes. Now, it would make sense to change the value of `n_pets` because we now have more than 0 pets. Since `n_pets` is an integer, which is an immutable type, we can change its value for the whole class only if we access it **directly** as a class attribute:

```
# access class attribute directly through the class
Pet.n_pets += 3

Pet.n_pets      # 3
tom.n_pets      # 3
avocado.n_pets  # 3
ben.n_pets      # 3
```

If we tried to change the value of `n_pets` via the instances it would not work as we wished:

```
# access class attribute through instances
tom.n_pets += 1
avocado.n_pets += 1
ben.n_pets += 1

Pet.n_pets      # 0
tom.n_pets      # 1
avocado.n_pets  # 1
ben.n_pets      # 1
```

Even though all instances have access to the class attribute, if those attributes are immutable, changing their value for one instance doesn't change them for the whole class.

The same would be with the attribute `kind`, since strings are also immutable in Python. If we change it for the object `ben` (since a goldfish is not a mammal), it would stay the same for other attributes (as it should):

```
ben.kind = "fish"

Pet.kind      # "mammal"
tom.kind      # "mammal"
avocado.kind  # "mammal"
ben.kind      # "fish"
```

In cases where there's a handful of unique objects that need to have a different value of the class variable, this is totally fine. However, if there're a lot of those objects, you should consider making this attribute an instance attribute!

The situation with the `pet_names` attribute is different. The `pet_names` attribute is a list and, therefore, mutable, so changes to it affect the whole class. See below:

```
tom.pet_names.append(tom.name)
avocado.pet_names.append(avocado.name)
ben.pet_names.append(ben.name)

Pet.pet_names      # ["Tom", "Avocado", "Benjamin"]
tom.pet_names      # ["Tom", "Avocado", "Benjamin"]
avocado.pet_names  # ["Tom", "Avocado", "Benjamin"]
ben.pet_names      # ["Tom", "Avocado", "Benjamin"]
```

If for some reason, we wanted the class attribute `pet_names` to store different values for different instances, we could do that by creating a new list instead of appending to the existing one:

```
tom.pet_names = ["Tom"]
avocado.pet_names = ["Avocado"]
ben.pet_names = ["Benjamin"]

Pet.pet_names      # []
tom.pet_names      # ["Tom"]
avocado.pet_names  # ["Avocado"]
ben.pet_names      # ["Benjamin"]
```

But this doesn't seem very convenient or necessary: after all, this is a class attribute and the idea behind it is that it stores values common to all the instances. So, again, if you want some attribute to store unique values, make it an instance attribute!

> Another way to look at this situation is in terms of (re)assignment. `=`, `+=`, and similar operators are **assignment** operators. When we try to modify the class attribute from the instance using these operators, we essentially create a new instance attribute for that particular object. This is why other instances and the class itself are unaffected by this change — because we assigned some value to a newly created "instance attribute". Adding a new element to the list with `append`, for example, is different because there is no reassignment happening, we just modify the existing list.

Like, for example, variable `legs`. It is an instance attribute, even though it is not explicitly passed as an argument of the `__init__` method. The default value is 4, but we can change it if we ever need to. That would be helpful for the object `ben` because the fish doesn't have legs (they do have fins, but let's save the question of whether a fin can be considered a leg in this context for another time). This is how we change the value of `legs` for the object `ben`:

```
ben.legs = 0
```

And that's it! There are basically no tricky moments with changing instance attributes because, again, they affect just one object.

§2. Adding attributes

In addition to changing attributes, we can also create attributes for the class or a particular instance. For example, we want to see the information about the species of all our pets. We then could write it in the class itself from the very beginning, or we could create a variable like this:

```
Pet.all_specs = [tom.spec, avocado.spec, ben.spec]

tom.all_specs      # ["cat", "dog", "goldfish"]
avocado.all_specs  # ["cat", "dog", "goldfish"]
ben.all_specs      # ["cat", "dog", "goldfish"]
```

Another thing we could do is to create an attribute for a specific instance. For example, we want to remember the breed of the dog called Avocado. Breeds are usually relevant in the context of dogs (cats have breeds too, but they are not that different) so it makes sense that we would want only our dog to have that information:

```
avocado.breed = "corgi"
```

Here we created an attribute `breed` for the object `avocado` and assigned a value `corgi` to it. Other instances of the class Pet as well as the class itself wouldn't have this attribute, so the following lines of code would cause an error:

```
Pet.breed  # AttributeError
tom.breed  # AttributeError
ben.breed  # AttributeError
```

§3. Summary

In this topic, we've shown the differences in using **class attributes** and **instance attributes**.

Class attributes are used to store information available for all instances of the class, but using them may be tricky if we don't take into account the type of the variable.

So, in what cases would we want to use class attributes? Well, firstly, if we want to define default values for all objects. Secondly, to store necessary class-specific constants (mathematical, for example). And lastly, to keep tabs on the data of all objects like in the example with `pet_names`. You may want to have easy access to particular information of every instance of your class and in that case, you could use a mutable class attribute.

> Remember that how the values of class attributes change depends on whether they are mutable or not. Take that into account when writing your program and operating the objects of the class!

Instance attributes, on the other hand, store information that is different for every instance, and it is obviously their main function. Changing and adding new instance attributes may only affect a single object, but still, pay attention to the alterations that you make.

We hope you now know the difference between the class and instance attributes and you'll use them successfully in your programs!

Comments (19) Hints (0) Useful links (1) Show discussion