Python → Object-oriented programming → Inheritance

Theory: Inheritance

28 minutes 0 / 5 problems solved

Skip this topic Start practicing

2840 users solved this topic. Latest completion was about 1 hour ago.

One of the main principles of object-oriented programming is **inheritance.** In this topic, we'll focus on inheritance in Python: what it means and how it's done.

§1. What is inheritance?

Inheritance is a mechanism that allows classes to inherit methods or properties from other classes. Or, in other words, inheritance is a mechanism of deriving new classes from existing ones.

The purpose of inheritance is to reuse existing code. Often, objects of one class may resemble objects of another class, so instead of rewriting the same methods and attributes, we can make it so that a class inherits those methods and attributes from another class.

When we talk about inheritance, the terminology resembles biological inheritance: we have **child classes** (or **subclasses, derived classes)** that inherit methods or variables from **parent classes** (or **base classes, superclasses).** Child classes can also redefine methods of the parent class if necessary.

§2. Class object

Inheritance is very easy to implement in your programs. Any class can be a parent class, so all we need to do is to write in the definition of the child class the name of the parent class in **parentheses** after the child class:

```
# inheritance syntax
class ChildClass(ParentClass):
    # methods and attributes
    ...
```

The definition of the parent class should precede the definition of the child class, otherwise, you'll get a `NameError`! If a class has several subclasses, its definition should precede them all. The "sibling" classes can be defined in any order.

When we don't define a parent for our class, it doesn't mean that it doesn't have any! By default, all classes have the class `object` as their parent. In Python 3.x we don't need to explicitly indicate that, so the definitions below are equivalent:

```
# parent class is explicit
class SomeClass(object):
    # methods and attributes
    ...


# parent class is implicit
class SomeClass:
    # methods and attributes
    ...
```

Subclasses of `object` inherit its methods and attributes. So, all standard methods like `__init__` or `__repr__` are inherited from the class `object`. If we don't redefine those methods for our custom classes, we end up using their implementations defined for the `object`.

§3. Single inheritance

Current topic:

Inheritance ···

Topic depends on:

✕ Inheritance ···

✓ Magic methods 3★ ···

Topic is required for:

Method overriding ···

Django ORM ···

Table of contents:

Unlike some other programming languages, Python supports two forms of inheritance: **single** and **multiple. Single inheritance** is when a child class inherits from one parent class. **Multiple inheritance** is when a child class inherits from multiple parent classes. In this topic, we'll cover only single inheritance. Don't worry, though, you'll have a chance to learn about multiple inheritance in the next topics!

Let's consider an example of single inheritance.

```
# parent class
class Animal:
    def __init__(self, name):
        self.name = name

# child class
class Dog(Animal):
    pass
```

Here we have a base class `Animal` with the `__init__` method and a subclass `Dog` that inherits from the base class. The keyword `pass` allows us not to write anything in the definition of the child class.

Now that we've defined classes, we can create objects:

```
cow = Animal("Bessie")  # instance of Animal
corgi = Dog("Baxter")   # instance of Dog
```

We haven't defined the `__init__` for the class `Dog` but since it's a child of `Animal`, it inherited its `__init__`. So if we tried to declare an instance of the class `Dog` in a different way, we would get an error:

```
labrador = Dog()  # TypeError
```

§4. type() vs isinstance()

There are two main ways to check the type of an object: `type()` or `isinstance()` functions.

The `type()` function takes one argument, an object, and returns its type. The `isinstance()` function takes two arguments: an object and a class. It checks if the given object is an instance of the given class and returns a boolean value.

For built-in types, they work the same, but when inheritance is involved, their results are different. Let's check it out!

First, let's look at the `type()` function:

```
print(type(cow) == Animal)  # True
print(type(corgi) == Animal)  # False

print(type(cow) == Dog)     # False
print(type(corgi) == Dog)     # True
```

As you can see, this allows us to check for the immediate type of the object. Now, `isinstance()` works differently:

```
print(isinstance(cow, Animal))    # True
print(isinstance(corgi, Animal))  # True

print(isinstance(cow, Dog))    # False
print(isinstance(corgi, Dog))  # True
```

With this, we get `True` not only with the immediate type but also with the parent type and even with the parent of the parent type! This distinction is important to remember for future projects!

§5. issubclass()

While `isinstance()` checks the type of an instance of a class, another built-in function asks whether a given class is a subclass of another class:

```
print(issubclass(Dog, Animal))  # True
print(issubclass(Animal, Dog))  # False

print(issubclass(Dog, Dog))     # True
print(issubclass(corgi, Dog))   # TypeError
```

As shown, the `issubclass()` function returns `True` if the first class inherits from the second class, and `False` otherwise. Each class is considered a subclass of itself. Notice that the function can't work with instances of a class, both its arguments should be classes. However, you can use a tuple of classes to check if your class inherits from several parents.

```
print(issubclass(Dog, object))            # True
print(issubclass(Dog, (Animal, object)))  # True
```

The case with several classes might be somewhat misleading, though. The thing is that the function checks whether ***any*** element of the tuple is a parent. Say, we have defined a new class `Robot`:

```
class Robot:
    pass
```

Then `issubclass()` will return the following:

```
print(issubclass(Dog, Robot))            # False
print(issubclass(Dog, (Robot, Animal)))  # True
```

Even though `Dog` has nothing to do with `Robot`, in the last case, we got `True`. So keep this detail in mind when calling this function!

§6. Summary

As one of the pillars of OOP, inheritance is very important! In Python, declaring parent classes is quite simple and straightforward. In this topic, we've covered the basics of the inheritance in Python: how it's done, what is class `object`, how to define a single parent for the class, and then check the type of an object or a class without any mistakes.

Inheritance is what really makes classes so powerful and useful. It also allows programmers to stick to the DRY (Don't Repeat Yourself) principle and pushes them to think about the effectiveness and clarity of their classes.

Report a typo

241 users liked this theory. **2** didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Comments (1)   Hints (0)   Useful links (2)   Show discussion