Python __new__

Summary: in this tutorial, you'll learn about the Python __new__ method and understand how Python uses it to create a new object.

Introduction to the Python __new__ method

When you create an instance of a class (https://www.pythontutorial.net/python-oop/python-class/) , Python first calls the `__new__()` method to create the object and then calls the `__init__()` (https://www.pythontutorial.net/python-oop/python-__init__/) method to initialize the object's attributes.

The `__new__()` is a static method (https://www.pythontutorial.net/python-oop/python-static-methods/) of the `object` class. It has the following signature:

```
object.__new__(class, *args, **kwargs)
```

The first argument of the `__new__` method is the `class` of the new object that you want to create.

The `*args` and `**kwargs` parameters must match the parameters of the `__init__()` of the class. However, the `__new__()` method does use them.

The `__new__()` method should return a new object of the class. But it doesn't have to.

When you define a new class, that class implicitly inherits from the `object` class. It means that you can override the `__new__` static method and do something before and after creating a new instance of the class.

To create the object of a class, you call the `super().__new__()` method.

Technically, you can call the `object.__new__()` method to create an object manually. However, you need to call the `__init__()` yourself manually after. Python will not call the `__init__()` method automatically if you explicitly create a new object using the `object.__new__()` method.

Python __new__ method example

The following defines the `Person` class with the `name` attribute and create a new instance of the `Person` class:

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


person = Person('John')
```

In Python, a class is callable (https://www.pythontutorial.net/python-built-in-functions/python-callable/) . When you call the class to create a new object:

```
person = Person('John')
```

Python will call the `__new__()` and `__init__()` methods. It's equivalent to the following method calls:

```
person = object.__new__(Person, 'John')
person.__init__('John')
```

The following shows the contents of the `__dict__` of the `person` object after calling the `__new__()` and `__init__()` methods:

```
person = object.__new__(Person, 'John')
print(person.__dict__)

person.__init__('John')
print(person.__dict__)
```

Output:

```
{}
{'name': 'John'}
```

As you can see clearly from the output, after calling the `__new__()` method, the `person.__dict__` is empty. And after calling the `__init__()` method, the `person.__dict__` contains the `name` attribute with the value ' `John'` .

The following illustrates the sequence which Python calls the `__new__` and `__init__` method when you create a new object by calling the class:

```
class Person:
    def __new__(cls, name):
        print(f'Creating a new {cls.__name__} object...')
        obj = object.__new__(cls)
        return obj

    def __init__(self, name):
        print(f'Initializing the person object...')
        self.name = name


person = Person('John')
```

Output:

```
Creating a new Person object...
Initializing the person object...
```

In this example, Python calls the __new__ method and the __init__ method after that.

When using the __new__ method

The following example defines the `SquareNumber` class that inherits from the built-in int type:

```
class SquareNumber(int):
    def __new__(cls, value):
        return super().__new__(cls, value ** 2)



x = SquareNumber(3)
print(x)  # 9
```

In this example, the `__new__()` method of the `SquareNumber` class accepts an integer and returns the square number. `x` is an instance of the `SquareNumber` class and also an instance of the `int` built-in type:

```
print(isinstance(x, int)) # True
```

Note that you cannot do this by using the `__init__()` method because the `__init__()` method of the built-in `int` takes no argument. The following code will result in an error:

```
class SquareNumber(int):
    def __init__(self, value):
        super().__init__(value ** 2)
```

```
x = SquareNumber(3)
```

Error:

```
TypeError: object.__init__() takes exactly one argument (the instance to initiali
```

In practice, you use the `__new__()` method when you want to tweak the object at the instantiated time.

For example, the following defines the `Person` class and uses the `__new__` method to inject the `full_name` attribute to the Person's object:

```
class Person:
    def __new__(cls, first_name, last_name):
        # create a new object
        obj = super().__new__(cls)

        # initialize attributes
        obj.first_name = first_name
        obj.last_name = last_name

        # inject new attribute
        obj.full_name = f'{first_name} {last_name}'
        return obj


person = Person('John', 'Doe')
print(person.full_name)

print(person.__dict__)
```

Output:

```
John Doe
{'first_name': 'John', 'last_name': 'Doe', 'full_name': 'John Doe'}
```

Typically, when you override the `__new__()` method, you don't need to define the `__init__()` method because everything you can do in the `__init__()` method, you can do it in the `__new__()` method.

Summary

- The `__new__()` is a static method of the `object` class.
- When you create a new object by calling the class, Python calls the `__new__()` method to create the object first and then calls the `__init__()` method to initialize the object's attributes.
- Override the `__new__()` method if you want to tweak the object at creation time.