Python type Class

Summary: in this tutorial, you'll learn about the Python type class and understand how Python uses the type class to create other classes.

Introduction to the Python type class

In Python, a class (https://www.pythontutorial.net/python-oop/python-class/) is an object of the class `type`. For example, the following defines the `Person` class with two methods (https://www.pythontutorial.net/python-oop/python-methods/) `__init__` and `greeting`:

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

    def greeting(self):
        return f'Hi, I am {self.name}. I am {self.age} year old.'
```

The `Person` class is an object of the class `type` as shown in the following statement:

```
print(type(Person))
```

Output:

```
<class 'type'>
```

The `Person` class is an instance of the `type` class:

```
print(isinstance(Person, type))
```

Python uses the type class to create other classes. The type class itself is a callable. The following shows one of the constructors of the type class:

```
type(name, bases, dict) -> a new type
```

The constructor has three parameters for creating a new class:

- `name` : is the name of the class e.g., Person
- `bases` is a tuple that contains the base classes of the new class. For example, the Person inherits from the object class, so the bases contains one class (object,)
- `dict` is the class namespace

Technically, you can use the `type` class to create a class dynamically. Before doing it, you need to understand how Python creates the classes.

When the Python interpreter encounters a class definition in the code, it will:

- First, extract the class body as string.
- Second, create a class dictionary for the class namespace.
- Third, execute the class body to fill up the class dictionary.
- Finally, create a new instance of type using the above `type()` constructor.

Let's emulate the steps above to create a `Person` class dynamically:

First, extract the class body:

```
class_body = """
def __init__(self, name, age):
    self.name = name
    self.age = age

def greeting(self):
    return f'Hi, I am {self.name}. I am {self.age} year old.'
"""
```

Second, create a class dictionary:

```
class_dict = {}
```

Third, execute the class body and fill up the class dictionary:

```
exec(class_body, globals(), class_dict)
```

The `exec()` function executes the class body and fills up the global and class namespaces.

Finally, create a new `Person` class using the `type` constructor:

```
Person = type('Person', (object,), class_dict)
```

Note that the `Person` is a class, which is also an object. The `Person` class inherits from the `object` class and has the namespace of the `class_dict` .

The following shows the type of the `Person` class which is the `type` class:

```
<class 'type'>
```

And it is an instance of the `type` class:

```
print(isinstance(Person, type))
```

Output:

```
True
```

The class_dict has the two functions `__init__` and `greeting` :

```
{'__init__': <function __init__ at 0x0000024E28465160>,
 'greeting': <function greeting at 0x0000024E28931550>}
```

The `Person.__dict__` also contains these functions :

```
mappingproxy({'__dict__': <attribute '__dict__' of 'Person' objects>,
              '__doc__': None,
              '__init__': <function __init__ at 0x00000165F7725160>,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Person' objects>,
              'greeting': <function greeting at 0x00000165F7F71550>})
```

In this example, Python creates the `type` class dynamically, which is the same as the one that you define statically in the code.

Because the `type` class creates other classes, we often refer to it as a **metaclass**. A metaclass is a class used to create other classes.

Summary

- In Python, a class is an instance of the type class.
- The type class creates other classs, therefore, it is called a metaclass.