Python Metaclass

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

Introduction to the Python Metaclass

A metaclass is a class (https://www.pythontutorial.net/python-oop/python-class/) that creates other classes. By default, Python uses the `type (https://www.pythontutorial.net/python-oop/python-type-class/)` metaclass to create other classes.

For example, the following defines a `Person` class:

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

When Python executes the code, it uses the `type` metaclass to create the `Person` class. The reason is that the `Person` class uses the `type` metaclass by default.

The explicit `Person` class definition looks like this:

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

The `metaclass` argument allows you to specify which metaclass class to use to define the class. Therefore, you can create a custom metaclass that has its own logic to create other classes. By using a custom metaclass, you can inject functionality into the class creation process.

Python metaclass example

First, define a custom metaclass called `Human` that has the `freedom` attribute sets to `True` by default:

```
class Human(type):
    def __new__(mcs, name, bases, class_dict):
        class_ = super().__new__(mcs, name, bases, class_dict)
        class_.freedom = True
        return class_
```

Note that the `__new__` (https://www.pythontutorial.net/python-oop/python-__new__/) method returns a new class or a class object.

Second, define the `Person` class that uses the `Human` metaclass:

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

The `Person` class will have the `freedom` attribute as shown in the class variables (https://www.pythontutorial.net/python-oop/python-class-variables/) :

```
pprint(Person.__dict__)
```

Output:

```
mappingproxy({'__dict__': <attribute '__dict__' of 'Person' objects>,
              '__doc__': None,
              '__init__': <function Person.__init__ at 0x000001E716C71670>,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Person' objects>,
              'freedom': True})
```

Put it all together.

```
from pprint import pprint


class Human(type):
    def __new__(mcs, name, bases, class_dict):
        class_ = super().__new__(mcs, name, bases, class_dict)
        class_.freedom = True
        return class_


class Person(object, metaclass=Human):
    def __init__(self, name, age):
        self.name = name
        self.age = age


pprint(Person.__dict__)
```

Metaclass Parameters

To pass parameters to a metaclass, you use the keyword arguments. For example, the following redefine the Human metaclass that accepts keyword arguments, where each argument becomes a class variable:

```
class Human(type):
    def __new__(mcs, name, bases, class_dict, **kwargs):
        class_ = super().__new__(mcs, name, bases, class_dict)
        if kwargs:
            for name, value in kwargs.items():
                setattr(class_, name, value)
        return class_
```

The following uses the `Human` metaclass to create a `Person` class with the `country` and `freedom` class variables set to `USA` and `True` respectively:

```
class Person(object, metaclass=Human, country='USA', freedom=True):
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Here's Person class variables:

```
pprint(Person.__dict__)
```

Output:

```
mappingproxy({'__dict__': <attribute '__dict__' of 'Person' objects>,
              '__doc__': None,
              '__init__': <function Person.__init__ at 0x0000018A334235E0>,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Person' objects>,
              'country': 'USA',
              'freedom': True})
```

Put it all together.

```
from pprint import pprint


class Human(type):
    def __new__(mcs, name, bases, class_dict, **kwargs):
        class_ = super().__new__(mcs, name, bases, class_dict)
        if kwargs:
            for name, value in kwargs.items():
                setattr(class_, name, value)
        return class_


class Person(object, metaclass=Human, freedom=True, country='USA'):
    def __init__(self, name, age):
        self.name = name
        self.age = age


pprint(Person.__dict__)
```

When to use metaclasses

Here's the quote of Tim Peter (https://en.wikipedia.org/wiki/Tim_Peters_(software_engineer)) who wrote the Zen of Python:

> Metaclasses are deeper magic that 99% of users should never worry about it. If you wonder whether you need them, you don't (the people who actually need them to know with certainty that they need them and don't need an explanation about why).
>
> *Tim Peter*

In practice, you often don’t need to use metaclasses unless you maintain or develop the core of the large frameworks such as Django.

Summary

- A metaclass is a class that creates other classes.