楼+之Python实战第10期 (/courses/1190)

面向对象编程

简介

面向对象编程（Object Oriented Programming，简称 OOP，面向对象程序设计）是一种程序设计思想。用面向过程的思想设计程序时，程序是一条条指令的顺序执行，当指令变得多起来时，它们可以被分隔成我们先前实验中讲解过的函数。而面向对象编程则是将对象视为程序的组成单元，程序的执行通过调用对象提供的接口完成。

面向对象的概念不容易通过理论讲解来理解，后续项目实战中我们会大量用到面向对象的思想，本节内容为后续实战做一定的铺垫，不会涉及太深入的内容。

面向对象的 4 个核心概念：

- 抽象
- 封装
- 继承
- 多态

下面我们通过例子和代码来理解这四个概念以及如何在 Python 中运用它们。

知识点

- 面向对象编程思想
- 抽象
- 封装、类与实例
- 继承与方法重写
- 多态
- 私有属性和方法
- 类方法与静态方法
- property 装饰器
- 类中的特殊方法

从面向过程说起

有这样一句话：

> 有一条叫旺财的狗和一只叫 Kitty 的猫在叫， 旺财发出 wang wang wang... 的叫声，Kitty 发出 miu miu miu... 的叫声。

楼+之Python实战第10期 (/courses/1190)

如果要把这句话转化为程序语言，使用面向过程的方法，可能会写出下面的代码，注意这部分示例不可以执行，只是讲解面向过程的编程方法：

```
main() {
    dog_name = '旺财';
    dog_sound = 'wang wang wang...';
    cat_name = 'Kitty';
    cat_sound = 'miu miu miu...';
    print(dag_name + ' is making sound ' + dog_sound);
    print(cat_name + ' is making sound ' + cat_sound);
}
```

执行结果会是：

```
旺财 is making sound wang wang wang...
Kitty is making sound miu miu miu...
```

这基本上就是面向过程的代码风格。

抽象

如何以面向对象的思想来写上面的程序呢？首先要做抽象。抽象就是对特定实例抽取共同的特征及行为形成一种抽象类型的过程。

在上面的程序中，“旺财”是狗的一种，它有一个名字，它可以“wang wang wang...” 的叫，我们可以抽象出这样一种类型，狗：

```
dog {
  name   (特征)
  sound() (行为)
}
```

同理，对于 Kitty，可以抽象为猫，猫也有名字，不过它的行为（叫声）和狗不同：

```
cat {
  name  (特征)
  sound() (行为)
}
```

特征和行为在程序语言中通常被称为属性（Attribute）和方法（Method）。

类与对象

楼+之Python实战第10期 (/courses/1190)

在自然界中给一切物体起名就采用了面向对象这种方式。比如说：所有的鸟都可以统称为“鸟类”，在“鸟类”这个总称下面又具体区分为不同的鸟名称，像“黄鹂”、“喜鹊”、“百灵鸟”等等。这些不同的鸟有共同的特征：两足、恒温、卵生、身披羽毛等等，但是它们也有区别于彼此的属性或是行为特征：羽毛的颜色、搭巢的方式不同等等。

在复杂的程序中，都推荐采用面向对象的方式编写程序，主要是为了能够快速的使用代码构建所需模型，完成快速开发。

面向对象的两个基本概念是类（Class）和实例（Instance），类是抽象的模板，而实例是根据类创建出来的一个个具体的“对象”。一个类中既会定义属性，也会定义方法。

下面的一些例子可以让大家更好地区分类和实例：

- 类：鸟； 实例对象：黄鹂、喜鹊、百灵鸟
- 类：苹果； 实例对象：红富士苹果、青苹果、红心苹果
- 类：狗； 实例对象：哈士奇、拉布拉多犬、贵宾犬

那么一个类是如何构成的呢？一般关注 3 个部分：

- 类的名称：简称类名
- 类的属性：它是一组数据，这组数据标识了这个类默认包含的数据
- 类的方法：它是一些函数，这些函数定义了这个类可以对数据进行哪些操作

比如下面就是一些类的设计：

- 类名：狗（Dog）； 属性：名字、性别、种类、毛的花色； 方法：吃东西、跑、跳、摇尾巴
- 类名：枪； 属性：种类、长度、重量； 方法：射击

创建类和实例对象

类的命名规则遵循大驼峰命名法：每个单词的第一个字母都大写，私有类使用一个下划线开头。

使用 `class` 关键字加上类的名称可以创建一个类，这个类的属性和方法需要写在冒号 `:` 下方的缩进代码块中。

```
class 类名:
    代码块（在这里定义类的属性和方法）
```

创建对象的方式为：

```
对象名 = 类名()
```

来看一个简单的例子：创建狗类 Dog 以及实例对象 dog ，在 Python 3 交互式命令行中输入：

楼+之Python实战第10期 (/courses/1190)

```
>>> class Dog:   # 注：object 类是所有自定义类的父类，用 python2 创建类时需要继承父类 object ，在 python3 中可以不写，它会默认继承此类
...     pass
...
>>> dog = Dog()   # 创建类的实例对象
>>> print(dog)    # 验证创建实例是否成功，打印类的实例 dog
<__main__.Dog object at 0x7ffff686d1d0>   # 打印实例的默认格式，这表示 Dog 类的 __main__ 模块中有一个实例对象，这个实例对象在计算机中的内存地址为 0x7ffff6869b38
>>> Dog
<class '__main__.Dog'>   # 这说明 Dog 是一个类
>>> dog.name = '旺财'   # 给实例对象绑定 name 属性
>>> dog.name    # 访问 dog 实例对象的 name 属性
'旺财'
>>>
```

实例方法

在 Python 中只要新建一个类就会自动创建它的内置类方法或属性，可以通过 dir(类名) 查看：

```
>>> class Dog:
...     pass
...
>>> dir(Dog)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
>>>
```

在上面的这些内置方法中，最常用的是 __init__ 方法，它可以对实例进行初始化设置（类的内置方法前后各有两个下划线 _ ，简称：“双下划线” ），设置一些默认的值，通过这个类创建的实例对象将默认拥有这些值（属性），在 /home/shiyanlou/Dog.py 文件中添加如下代码：

```
class Dog:
    def __init__(self):
        self.name = '旺财'

dog = Dog()
print(dog.name)
```

执行文件：

```
$ python3 Dog.py    # 执行文件
旺财
```

楼+之Python实战第10期 (/courses/1190)

在上面的例子中， `__init__` 方法的第一个参数 `self` 表示创建的实例对象本身，也就是对象它自己，当创建一个实例对象时，`__init__` 方法会被默认调用，Python 解释器会把这个对象的引用作为第一个参数传递给 `self` ，不需要开发者传递。

对象也可以从外面传递数据给类，用来绑定数据到对应的属性上。修改 `Dog.py` 文件如下所示：

```
class Dog:
    def __init__(self, name, age):  # 创建类的实例时必须传入 name 参数
        self.name = name      # 设置实例的属性
        self.age = age

dog = Dog('旺财', 2)               # 创建类的实例
print(dog)                     # 打印类的实例
print(dog.name)                # 打印实例的属性 name
print(dog.age)
```

执行文件：

```
$ python3 Dog.py   # 执行文件
<__main__.Dog object at 0x7ffff6869b38>    # 实例的默认格式
旺财   # 实例的 name 属性
2    # 实例的 age 属性
```

从上面可以看出，类中的函数定义方法和普通的函数定义大致相同，唯一的区别是类中的函数必须有一个 `self` 参数，且该参数必须放在第一个位置，用来表示实例化对象的引用。

如上所示，当我们打印类的实例时，默认的格式就是由一对尖括号包起来的， `at` 后面是内存地址，每个实例的地址都不同。但是这样很乱，不太容易看清这是哪个实例。我们可以使用类的另一个内置方法 `__repr__` 来格式化实例的打印格式，修改 `Dog.py` 文件如下：

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

    def __repr__(self):
        return 'Dog: {}'.format(self.name)  # 自定义打印格式

dog = Dog('旺财', 2)     # 创建类的实例
print(dog)             # 打印类的实例，执行 print ，在 Dog 类中会自动调用 __repr__ 方法
print(dog.name)        # 打印实例的属性 name
```

执行文件：

```
$ python3 Dog.py
Dog: 旺财          # 自定义打印格式的效果
旺财
```

练习题：按要求实现类

楼+之Python实战第10期 (/courses/1190)

在 /home/shiyanlou 目录下创建代码文件 classtest.py ：

```
$ cd /home/shiyanlou/
$ touch classtest.py
```

classtest.py 中包含一个 UserData 类，该类可以保存用户的 id 和名字，并且可以使用 print 按照一定的格式直接打印输出。

向 classtest.py 文件写入以下代码内容：

```
#!/usr/bin/env python3

class UserData:
    TODO

if __name__ == '__main__':
    user1 = UserData(101, 'Jack')
    user2 = UserData(102, 'Louplus')
    print(user1)
    print(user2)
```

预期输出的是：

```
$ python3 /home/shiyanlou/classtest.py
ID:101 Name:Jack
ID:102 Name:Louplus
```

后续题目都将会不断完善 classtest.py ，请完成本题目后不要删除文件。

请按照题目要求完善 UserData 类，程序完成后，点击 下一步 ，系统将自动检测完成结果。

封装

在面向对象的语言中，封装就是用类将数据和基于数据的操作封装在一起，隐藏内部数据，对外提供公共的访问接口。

将上面“抽象”那一节讲的内容转化为 Python 程序：

```
class Dog(object):
    def __init__(self, name):
        # 不同于 Java 或者 C++，Python 没有特定的关键字声明私有属性
        # Python 的私有属性用一个或两个下划线开头表示
        # 一个下划线表示外部调用者不应该直接调用这个属性，但还是可以调用到
        # 两个下划线外部就不能直接调用到了
        self._name = name
    def get_name(self):
        return self._name
    def set_name(self, value):
        self._name = value
    def bark(self):
        print(self.get_name() + 'is making sound wang wang wang...')

class Cat(object):
    def __init__(self, name):
        self._name = name
    def get_name(self):
        return self._name
    def set_name(self, value):
        self._name = value
    def mew(self):
        print(self.get_name() + 'is making sound miu miu miu...')
```

类是一个抽象的概念，而实例是一个具体的对象。比如说狗是一个抽象的概念，因为狗有很多种，而那个正在 wang wang 叫的叫旺财的狗是一个实例。

面向对象风格的主程序就变成这样：

```
# 在 Python 实例化一个对象
dog = Dog('旺财')
cat = Cat('Kitty')
dog.bark()
cat.mew()
```

`Dog` 类中的 `bark` 方法实现了狗叫的信息输出，但使用这个方法需要先用 `Dog()` 创建一个对象。

隐藏数据访问有什么好处呢？最大的好处就是提供访问控制。比如在 `Cat` 类中，用户输入的名字可能有小写，有大写，而我们希望对外提供的名词都是首字母大写，其余字母小写，那么我们就可以在 `get_name` 方法中做访问控制：

```
def get_name(self):
    return self._name.lower().capitalize()
```

封装成类操作视频：

楼+之Python实战第10期 (/courses/1190)

0:00 / 2:27

注意：视频中有一部分描述不准确，在 Python 的类中， `__init__()` 函数是在对象创建中执行的，并不是用来创建对象的必备函数，创建对象的实际函数是 `__new__()` ，而 `__new__()` 是继承自 object 类所具备的函数，此处可以不必重新实现，并且在 `__new__()` 中甚至可以指定是否执行 `__init__()` 。

继承与方法重写

继承分为两种：单继承和多继承。单继承是指子类只继承于一个父类，相应的多继承是指子类继承于多个父类。

单继承与方法重写

将“旺财”抽象为狗，“Kitty”抽象为猫，这是显而易见的，以至于你可能都没察觉到已经完成了一层抽象。对狗和猫还可以做进一步的抽象：它们都是动物，它们都有一个名字，它们都能发出叫声，只是叫声不同。这样我们就形成了一个抽象的父类：

```
class Animal(object):
    def __init__(self, name):   # 初始化设置名字
        self._name = name       # 设置为私有属性
    def get_name(self):        # 获取名字，私有属性不可以直接被访问，因此设置该方法获取属性值
        return self._name
    def set_name(self, value):   # 设置私有属性值
        self._name = value
    def make_sound(self):
        pass
```

因为每一种动物发出的声音不同，所以在抽象类中并没有实现 `make_sound` 方法，使用 pass 直接略过这个函数，不做任何事情。

狗和猫都是动物，它们之间就形成了一种继承关系，用 Python 语言表示出来就是：

楼+之Python实战第10期 (/courses/1190)

```
class Dog(Animal):   # 继承于父类 Animal
    def make_sound(self):  # 重写父类的 make_sound 方法
        print(self.get_name() + ' is making sound wang wang wang...')


class Cat(Animal):   # 继承于父类 Animal
    def make_sound(self):  # 重写父类的 make_sound 方法
        print(self.get_name() + ' is making sound miu miu miu...')
```

Dog 和 Cat 继承了父类 Animal 的初始化方法， `get_name` 和 `set_name` 方法，并重写了父类的 `make_sound` 方法。所谓重写父类的方法就是指：在子类中定义和父类同名的方法，那么子类中的该方法就会覆盖掉父类中对应的方法。

现在主程序变成：

```
dog = Dog('旺财')
cat = Cat('Kitty')
dog.make_sound()
cat.make_sound()
```

继承操作视频：

0:00 / 2:49

在继承中，如果子类想要使用父类 `__init__` 中的属性，可以使用 `super()` 调用。

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

class Dog(Animal):
    def __init__(self,name,age):
        # 这是写法1
        super().__init__(name)
        # 这是写法2
        # super(Dog,self).__init__(name)
        # 这是写法3
        # Animal.__init__(self,name)
        self.age = age
```

运行代码：

```
>>> dog = Dog('旺财')
>>> dog.name
'旺财'
```


多继承

多继承是指一个子类继承了多个父类，因此这个子类也相应的拥有了所有父类的属性，可以调用所有父类中的方法。比如在 `/home/shiyanlou/mixin.py` 文件中写入如下的代码：

```
class A:
    def __init__(self):
        self.name = 'xiaoming'
    def testA(self):
        print('----testA----')

class B:
    def __init__(self):
        self.age = 8
    def testB(self):
        print('----testB----')

class Person(A,B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
    def testPerson(self):
        print('----testPerson----')

person = Person()
print(person.name)
print(person.age)
person.testA()
person.testB()
person.testPerson()
```

执行代码，结果如下：

```
$ python3 mixin.py
xiaoming
8
----testA----
----testB----
----testPerson----
```

在上面的代码中，如果父类 A 和父类 B 有相同的方法，那么子类去调用的时候的调用顺序是什么样的呢？修改上面的代码为如下所示：

```
    def test(self):
        print('----testA----')

class B:
    def test(self):
        print('----testB----')

class Person(A,B):
    pass

person = Person()
person.test()
print(Person.mro())   # 类名.mro() 方法可以查看该类的继承顺序，也就是调用方法的解析顺序
```

执行代码，结果如下：

```
$ python3 mixin.py
----testA----
[<class '__main__.Person'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```

通过打印的结果可以看出调用方法的解析顺序依次是：Person、A、B、object

但是由于在 Python 中多重继承定义相同的方法名容易出现调用次序不确定的问题，造成程序的可理解性变差，所以推荐在设计类时使用 `Mixin` 的设计思想。简而言之就是：保持继承的单一性原则，一个子类只继承一个主要的类，其它功能拆分出来，单独的一个功能命名为一个 `类Mixin` 。如下所示：

```
class People():
    pass

class Coder(People):
    pass

class WritableMixin():   # 把会写作的能力独立出来作为一个类
    pass

class somebody(Coder, WritableMixin):
    pass
```

练习题：继承类并重写方法

在 `/home/shiyanlou/classtest.py` 中实现一个新类 NewUser，继承 UserData 类，并且额外提供新的接口：

1. `get_name(self)` ：返回 NewUser 对象的名称
2. `set_name(self, value)` ：设置 NewUser 对象的名称为 value

同时当我们使用 `print()` 打印 NewUser 对象的时候，输出的格式与 UserData 对象有了区别。

楼+之Python实战第10期 (/courses/1190)

向 `classtest.py` 文件写入以下代码内容：

```
#!/usr/bin/env python3

class UserData:
    # 之前题目中已经完成，不需修改

# TODO：增加 NewUser 的定义

if __name__ == '__main__':
    user1 = NewUser(101, 'Jack')
    user1.set_name('Jackie')
    user2 = NewUser(102, 'Louplus')
    print(user1)
    print(user2)
```

预期输出的是：

```
$ python3 /home/shiyanlou/classtest.py
ID:101 Name:Jackie
ID:102 Name:Louplus
```

请按照题目要求完善 NewUser 类，程序完成后，点击 下一步 ，系统将自动检测完成结果。

多态

简单的说，多态就是使用同一方法对不同对象可以产生不同的结果。

下面我们通过一个例子来说明，为了方便说明，假设我们在使用的是一门类似 Java 的强类型语言，使用变量前必须声明类型。

假设现在又来了一只叫“来福”的狗和一只叫“Betty”的猫，它们也在叫，那么我们可能会写出这样的代码：

```
# 伪代码
Dog dog1 = new Dog('旺财');
Cat cat1 = new Cat('Kitty');
Dog dog2 = new Dog('来福');
Dog cat2 = new Cat('Betty');
dog1.make_sound();
cat1.make_sound();
dog2.make_sound();
cat2.make_sound();
```

Dog 和 Cat 都继承自 Animal，并且实现了自己的 make_sound 方法，那么借助强类型语言父类引用可以指向子类对象这一特性，我们可以写出多态的代码：

楼+之Python实战第10期 (/courses/1190)

```
# 伪代码
Set animals = [new Dog('旺财'), new Cat('Kitty'), new Dog('来福'), new Cat('Betty')];
Animal animal;
for (i = 0; i <= animals.lenth(); i++) {
    # 父类引用指向子类对象
    animal = animals[i];
      # 多态
    animal.make_sound();
}
```

在 Python 这种动态类型的语言中可能没有那么明显的体现多态的威力，因为在 Python 中，你可以用任意变量指向任意类型的值。上面过程用 Python 来写的话会非常简单：

```
animals = [Dog('旺财'), Cat('Kitty'), Dog('来福'), Cat('Betty')]
for animal in animals:
    animal.make_sound()
```

可以看到不管 animal 具体是 Dog 还是 Cat，都可以在 for 循环中执行 make_sound 这个方法。这就是面向对象的多态性特征。

多态示例操作视频：

- 拓展阅读：浅析 Python 的类、继承和多态 (https://segmentfault.com/a/1190000010046025)

私有属性和方法

在 Java 和 C++ 中，可以用 `private` 和 `protected` 关键字修饰属性和方法，它们控制属性和方法能否被外部或者子类访问，在 Python 中约定在属性方法名前添加 `__` （两个下划线 `_` ）来拒绝外部的访问。

楼+之Python实战第10期 (/courses/1190)

```
>>> class Shiyanlou:
...   __private_name = 'shiyanlou'
...   def __get_private_name(self):
...     return self.__private_name
...

>>> s = Shiyanlou()
>>> s.__private_name
Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

AttributeError: 'Shiyanlou' object has no attribute '__private_name'


>>> s.__get_private_name()

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

AttributeError: 'Shiyanlou' object has no attribute '__get_private_name'
```

为什么说是“约定”，因为 Python 中没有绝对的私有，即使是 `__` 两个下划线来约束，也是可以通过 `obj._Classname__privateAttributeOrMethod` 来访问：

```
>>> s._Shiyanlou__private_name

'shiyanlou'
>>> s._Shiyanlou__get_private_name()

'shiyanlou'
```

所以说 `__` 只是约定，告诉外部使用者不要直接使用这个属性和方法。虽然可以通过 `对象名._class__attr` 的方式获取，但强烈不推荐使用这种方法获得属性/方法值。

两个下划线是设置私有属性/方法的标准样式， 还有一种设置私有属性/方法的样式，就是在属性/方法名字前加一个下划线 _attr 这样的私有属性/方法约定俗成地视为不能直接访问，尽管它仍然可以被直接访问。

- 拓展阅读：在类中封装属性名 (https://python3-cookbook.readthedocs.io/zh_CN/latest/c08/p05_encapsulating_names_in_class.html?highlight=%E7%A7%81%E6%9C%89%E5%B1%9E%E6%80%A7)

类属性、类方法与静态方法

本节主要介绍除了实例方法以外的另外两种方法，分别是类方法和静态方法。

类属性和类方法

类属性和类方法是可以直接从类访问，不需要实例化对象就能访问（但是实例化对象是可以访问的）。

类属性是类对象的属性，通过类对象定义的实例对象都可以拥有这个属性，但是在类外，只有公有的类属性才可以被直接被访问。

假设上面例子中的动物它们都是 Jack 养的，那么就可以在 `Animal` 类中用一个类属性表示，一般声明在 `__init__` 前面：

```
class Animal(object):
    owner = 'jack'
    __owner_age = 32
    def __init__(self, name):
        self._name = name
```

现在可以通过 Animal 或者子类直接访问：

```
>>> Animal.owner  # 通过类本身访问公有的类属性
'jack'
>>> Cat.owner   # 通过子类访问公有的类属性
'jack'
>>> animal = Animal('dog')   # 实例化对象
>>> animal.owner   # 通过实例化对象访问公有的类属性
'jack'
>>> Animal.__owner_age   # 通过类本身不能访问私有的类属性
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Animal' has no attribute '__owner_age'
```

类方法和类属性类似，它也可以通过类名直接访问，类方法用装饰器 `@classmethod` 装饰，类方法中可以访问类属性，下面添加了一个类方法 `get_owner` ：

```
class Animal(object):
    owner = 'jack'
    def __init__(self, name):
        self._name = name
    @classmethod
    def get_owner(cls):
        return cls.owner
```

注意类方法的第一个参数传入的是类对象，而不是实例对象，所以是 `cls` 。约定俗成的，类方法中要指代类对象本身都使用 `cls` 。

通过类方法获取 owner:

```
>>> Animal.get_owner()   # 通过类本身访问类方法
'jack'
>>> Cat.get_owner()   # 通过子类访问类方法
'jack'
>>> animal = Animal('dog')   # 实例化对象
>>> animal.get_owner()   # 通过实例化对象访问类方法
'jack'
```

类属性和类方法操作视频：

注意：视频中讲的“静态变量”就是指的类属性。

类方法的一个用处就是可以通过类方法来修改类属性，比如看下面的代码：

```
class Animal(object):
    owner = 'jack'
    def __init__(self, name):
        self._name = name
    @classmethod
    def get_owner(cls):
        return cls.owner
    @classmethod
    def set_owner(cls,name):
        cls.owner = name
```

执行代码，修改类属性：

```
>>> Animal.get_owner()
'jack'
>>> Animal.set_owner('rose')   # 修改类属性
>>> Animal.get_owner()
'rose'
```

静态方法

静态方法用装饰器 `@staticmethod` 装饰，和 `@classmethod` 有点类似。静态方法在运行时不需要实例的参与，它被放在类下面只是因为它和类有一点关系，但并不像类方法那样需要传递一个 `cls` 参数。

静态方法的应用场景是当一个函数完全可以放到类外面单独实现的时候，如果这个函数和类还有一点联系，放入类中能更好的组织代码逻辑，那么可以考虑使用类中的静态方法。

比如说，Animal 下面有一个方法，主人 Jack 可以调用它来购买小动物的食物：

```
class Animal(object):
    owner = 'jack'
    def __init__(self, name):
        self._name = name

    @staticmethod
    def order_animal_food():
        print('ording...')
        print('ok')
```

调用静态方法：

```
>>> Animal.order_animal_food()
ording...
ok
```

类中只有 3 种方法，分别是：实例方法、类方法和静态方法，简单总结一下这 3 者的区别：

区别	第一个参数强制为	可以引用
实例方法	实例对象 self	类属性、实例属性、类方法、实例方法
类方法	类对象 cls	类属性、类方法
静态方法	无	类属性

练习题：类方法和静态方法

首先，为 `/home/shiyanlou/classtest.py` 中的 NewUser 增加一个类属性：

```
group = 'shiyanlou-louplus'
```

继续为 `/home/shiyanlou/classtest.py` 中的 NewUser 增加新的方法：

1. 增加类方法 `get_group` ：返回 NewUser 的类属性 group 的值
2. 增加静态方法 `format_userdata(id, name)` ：直接按照一定格式打印传入的用户数据

向 `classtest.py` 文件中， `__main__` 中的测试用例：

楼+之Python实战第10期 (/courses/1190)

```
#!/usr/bin/env python3

class UserData:
    # 之前题目中已经完成，不需修改

# TODO: 向 NewUser 中添加方法和变量

if __name__ == '__main__':
    print(NewUser.get_group())
    print(NewUser.format_userdata(109,'Lucy'))
```

预期输出的是：

```
$ python3 /home/shiyanlou/classtest.py
shiyanlou-louplus
Lucy's id is 109
```

请按照题目要求完善 NewUser 类，程序完成后，点击 下一步 ，系统将自动检测完成结果。

property 装饰器

来看之前的 Animal 类，在其中定义 age 属性:

```
>>> class Animal:
...     def __init__(self):
...         self.age = 3
...
>>> cat = Animal()
>>> cat.age   # 获取 age 属性值
3
>>> cat.age = 'a'   # 改变 age 属性值
>>> cat.age
'a'
>>>
```

在上面的例子中，可以看到如果 age 属性值可以被公开访问，用户赋值为字符串，很明显这不符合实际情况。因此可以在类中设置 get_age 和 set_age 方法进行取值和修改值的操作，如下所示：

楼+之Python实战第10期 (/courses/1190)

```
>>> class Animal:
...     def __init__(self):
...         self.__age = 3   # 把 age 设置为私有属性
...     def get_age(self):    # get_age 方法获取属性值
...         return self.__age
...     def set_age(self, value):     # set_age 方法修改属性值，在其中对值做判断
...         if isinstance(value,int):
...             self.__age = value
...         else:
...             raise ValueError
...
>>> cat = Animal()
>>> cat.get_age()
3
>>> cat.set_age('a')   # 如果赋值非整数就会报错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 10, in set_age
ValueError
>>> cat.set_age(5)
>>> cat.get_age()
5
>>>
```

上面的函数定义如果使用 `@property` 装饰器会更加符合 Python 的风格。

`@property` 装饰器可以将一个方法变成一个属性来使用，通过 `@property` 装饰器可以获得和修改对象的某一个属性。

使用 `@property` 装饰器的方法如下：

- 只有 `@property` 表示只读
- 同时有 `@property` 和 `@*.setter` 表示可读可写
- 同时有 `@property` 、 `@*.setter` 、和 `@*.deleter` 表示可读可写可删除
- `@property` 必须定义在 `@*.setter` 的前面
- 类必须继承 object 父类，否则 `@property` 不会生效

用 `@property` 改写的 Animal 类：

楼+之Python实战第10期 (/courses/1190)

```
>>> class Animal(object):
...     def __init__(self):
...         self.__age = 3
...     @property
...     def age(self):
...         return self.__age
...     @age.setter
...     def age(self,value):
...         if isinstance(value, int):
...             self.__age = value
...         else:
...             raise ValueError
...     @age.deleter
...     def age(self):
...         print('delect age')
...         del self.__age
...
>>> cat = Animal()   # 创建实例
>>> cat.age   # 获取值
3
>>> cat.age = 'a'   # 赋予值为字符串，报错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in age
ValueError
>>> cat.age = 6   # 赋予值为整数，成功
>>> cat.age
6
>>> del cat.age   # 删除 age 属性
delect age
>>> cat.age
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in age
AttributeError: 'Animal' object has no attribute '_Animal__age'
>>>
```

这样我们就能以访问属性的方式获取和修改 `age` 属性了。

从这个简单的例子中我们可以发现 `age` 由一个函数转变为一个属性，并且通过增加一个 setter 函数的方式来支持 `age` 的设置。通过 property 和 setter ，可以有效地实现 `get_age`（获取对象的属性）和 `set_age`（设置对象的属性）这两个操作，而不需要直接将内部的 `__age` 属性暴露出来，同时可以在 setter 函数中对设置的参数进行检查，避免了直接对 `__age` 内部属性进行赋值的潜在风险。

练习题：应用 property

完善 `/home/shiyanlou/classtest.py` 中的 NewUser 类，符合以下需求：

1. 使用 property 和 setter 来替代 `get_name` 和 `set_name`

2. 当设置对象名称的时候需要规范新的对象名称必须超过3个字符，小于3个字符的要报错，打印 ERROR

楼+之Python实战第10期 (/courses/1190)

向 classtest.py 文件写入以下代码内容：

```
#!/usr/bin/env python3

class UserData(object):
    def __init__(self,id,name):
      self.id = id
      self._name = name

# TODO: 按题目要求修改 NewUser 的定义

if __name__ == '__main__':
    user1 = NewUser(101, 'Jack')
    user1.name = 'Lou'
    user1.name = 'Jackie'
    user2 = NewUser(102, 'Louplus')
    print(user1.name)
    print(user2.name)
```

预期输出的是：

```
$ python3 /home/shiyanlou/classtest.py
ERROR
Jackie
Louplus
```

请按照题目要求完善 NewUser 类，程序完成后，点击 下一步 ，系统将自动检测完成结果。

类中的特殊方法（魔法方法）

在 Python 中有一些特殊的方法，它们是 Python 内置的方法，通常以双下划线来命名，比如 __init__ 、 __repr__ 等等，在类中使用它们时往往较少的代码就可以发挥很大的作用，提高开发效率，因此在 Python 中称这些方法为“魔法方法”。在这里主要介绍 Python 类中常用的魔法方法。

__new__ 和 __del__

在 Python 中最常使用的是 __init__ 方法，它可以用于新建实例对象的时候给对象绑定属性，但是在新建对象的时候第一个调用的不是 __init__ 方法，而是 __new__(cls, [...]) 方法，这两个方法的区别在于：

- __init__ 方法是在实例对象创建完成后调用的，主要用于设置实例对象的初始值，它的第一个参数为 self，可以不需要返回值

- `__new__` 方法是在实例对象被创建之前调用的，主要用于创建实例对象并返回实例对象，它的第一个参数为 cls，它只会取 cls 参数，其余参数都传给了 `__init__` 方法，必须要有返回值可以是 `super().__new__(cls)` 或是 `object.__new__(cls)`

与之对应的，有创建实例就有销毁实例，在执行 `del 实例对象` 的时候真正调用的是类中的 `__del__` 方法，它定义了当对象被销毁时需要执行的行为。

看下面的这个例子：

```
>>> class Animal(object):
...     def __new__(cls,name):
...         print("__new__")
...         return super(Animal,cls).__new__(cls)
...     def __init__(self,name):
...         print("__init__")
...         self.name = name
...     def __del__(self):
...         print("__del__")
...
>>> cat = Animal('tom')
__new__   # 创建对象时先调用 __new__ 方法
__init__   # 然后调用 __init__ 方法
>>> cat.name
'tom'
>>> del cat
__del__   # 删除对象时调用的是 __del__ 方法
>>>
```

__slots__

由于 Python 是动态语言，因此定义一个实例对象后可以绑定任意的属性，如果需要限制绑定属性类别，可以使用 `__slots__` 变量，可以绑定的属性值以元组的形式赋予给它。需要注意的是：`__slots__` 中定义的属性只在定义的类中有效，在被继承的子类中是无效的，如果想要在子类中限制属性则需要重新定义。

楼+之Python实战第10期 (/courses/1190)

```
>>> class Animal(object):
...     __slots__ = ('name','age')  # 限定 Animal 类只能定义 name 和 age 属性
...
>>> dog = Animal()
>>> dog.name = 'wangcai'
>>> dog.age = 2
>>> dog.gender = 'male'  # dog 是 Animal 的实例化对象，无法定义 gender 属性
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Animal' object has no attribute 'gender'
>>> class Cat(Animal):   # Cat 类继承于 Animal 类
...     __slots__ = ('address')   # 限定 Cat 类定义 address 属性，但是由于 Cat 继承于 Animal ，所以可以定义的属性有：name 、age、address
...
>>> cat = Cat()
>>> cat.gender = 'male'   # 无法定义 gender 属性
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Cat' object has no attribute 'gender'
>>> cat.address = 'chengdu'
>>> cat.name = 'tom'   # 可以定义 name 属性
>>>
```

__getattribute__ 和 __getattr__

在 Python 类中，当访问实例对象属性的时候，其实默认会调用 `__getattribute__` 方法，如果没有该属性就会报 AttributeError 错误，这个时候可以考虑使用 `__getattr__` 方法进行自定义。两者的区别主要如下：

区别	**__getattribute__(self,attr)**	**__getattr__**
调用时间	调用实例对象属性时触发 (无论属性是否存在)	获取不存在的实例对象属性时触发
参数	self,attr(实例对象属性)	self,attr(实例对象属性)
返回值	必须有	必须有
作用	控制访问权限	访问不存在属性时进行自定义

需要注意的是： `__getattribute__` 方法会比 `__getattr__` 方法优先调用，如果调用 `__getattribute__` 方法有返回值就不会再调用 `__getattr__` 方法了。

```
class Animal(object):
    def __init__(self,name,gender):
        self.name = name
        self.gender = gender
    def __getattribute__(self,attr):
        print('调用了 __getattribute__ 方法，访问{}属性'.format(attr))
        if attr in ('name','gender'):
            return object.__getattribute__(self,attr)
        else:
            print('交给__getattr__处理...')
            return object.__getattr__(self,attr)
    def __getattr__(self,attr):
        print('调用了 __getattr__ 方法，访问{}属性'.format(attr))
        if attr == 'age':
            return 3
        else:
            return '__getattr__ 方法中未找到该属性'
```

执行结果如下：

```
>>> dog = Animal('wangcai','male')
>>> dog.name
调用了 __getattribute__ 方法，访问name属性
'wangcai'
>>> dog.gender
调用了 __getattribute__ 方法，访问gender属性
'male'
>>> dog.age
调用了 __getattribute__ 方法，访问age属性
交给__getattr__处理...
调用了 __getattr__ 方法，访问age属性
3
>>> dog.address
调用了 __getattribute__ 方法，访问address属性
交给__getattr__处理...
调用了 __getattr__ 方法，访问address属性
'__getattr__ 方法中未找到该属性'
```

__call__

在 Python 中一切皆对象，对象分为可调用的和不可调用的，凡是可以通过一对括号（）来调用的都是可调用对象，比如函数、类等，可以通过 `callable()` 函数来判断一个对象是否是可调用对象。

```
>>> class Animal(object):
...     pass
...
>>> dog = Animal()
>>> callable(dog)
False   # 说明是不可调用对象
>>>
```

通常情况下实例对象都是不可调用对象，但是如果在类中使用了 `__call__` 方法就可以将实例对象转换为可调用对象：

```
>>> class Animal(object):
...     def __call__(self):
...         print('__call__')
...
>>> dog = Animal()
>>> dog()
__call__
>>> callable(dog)
True
>>>
```

其中一个使用场景就是把类作为一个装饰器来使用，下面这个例子可以记录 num_counter 函数被调用的次数：

```
>>> class Counter(object):
...     def __init__(self,func):   # 把类用作装饰器时，init 只能接受被装饰的函数这一个参数
...         self.func = func
...         self.count = 0
...     def __call__(self,*args,**kwargs):   # 让对象可调用
...         self.count += 1
...         return self.func(*args,**kwargs)
...
>>> @Counter
... def num_counter():
...     pass
...
>>> for i in range(20):
...     num_counter()
...
>>> num_counter.count
20
>>>
```

__iter__ 和 __next__

使用 `__iter__` 方法就可以让类成为一个可迭代对象，如果使用 for 循环遍历类对象还需要在类中定义 `__next__` 方法，在 `__next__` 方法中可以定义取值的规则，当超出取值规则会抛出 StopIteration 异常从而退出当前循环。

```
>>> class Test(object):
...     def __init__(self,data=0):
...         self.data = data
...     def __iter__(self):
...         return self
...     def __next__(self):
...         if self.data > 5:
...             raise StopIteration
...         else:
...             self.data += 1
...             return self.data
...
>>> for i in Test():
...     print(i)
...
1
2
3
4
5
6
>>>
```

练习题：应用 __call__

继续为 `/home/shiyanlou/classtest.py` 中的 NewUser 增加新的方法实现：

- 使用 NewUser 类生成的实例对象可调用

向 `classtest.py` 文件中，`__main__` 中的测试用例：

```
#!/usr/bin/env python3

class UserData:
    # 之前题目中已经完成，不需修改

# TODO：向 NewUser 中添加方法

if __name__ == '__main__':
    user = NewUser(101, 'Jack')
    user()
```

预期输出的是：

```
$ python3 /home/shiyanlou/classtest.py
Jack's id is 101
```

请按照题目要求完善 NewUser 类，程序完成后，点击 下一步 ，系统将自动检测完成结果。

总结

楼+之Python实战第10期 (/courses/1190)

本节实验通过几个例子学习了面向对象的 4 个核心概念：

- 抽象
- 封装
- 继承
- 多态

同时，也学习了 Python 面向对象编程中类及对象的使用方法，以及经常在开发中会用到的私有属性和方法、静态变量和类方法、property及静态方法的用法。面向对象的编程方法在后续的项目实战中会大量用到。

拓展阅读

学完这节课后，一些同学会对函数名称前后出现的下划线困扰。你可以阅读：

- 《Underscores in Python（译文：关于 Python 中的下划线）》(https://segmentfault.com/a/1190000002611411)

另外，进一步加深对 Python 中出现的类、方法等概念的理解，可以阅读：

- 《Python 官方文档（中文）- 类》(http://www.pythondoc.com/pythontutorial3/classes.html#)
- 《Python 官方文档（中文）- 继承》(http://www.pythondoc.com/pythontutorial3/classes.html#tut-inheritance)

上一节：挑战：完善工资计算器 (/courses/1190/labs/8522/document)

下一节：文件处理 (/courses/1190/labs/8524/document)