

07 面向对象编程

面向对象编程——Object Oriented Programming，简称OOP，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

而面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

在Python中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

我们以一个例子来说明面向过程和面向对象在程序流程上的不同之处。

假设我们要处理学生的成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个dict表示：

```
std1 = { 'name': 'Michael', 'score': 98 }
std2 = { 'name': 'Bob', 'score': 81 }
```

而处理学生成绩可以通过函数实现，比如打印学生的成绩：

```
def print_score(std):
    print('%s: %s' % (std['name'], std['score']))
```

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是 `Student` 这种数据类型应该被视为一个对象，这个对象拥有 `name` 和 `score` 这两个属性（Property）。如果要打印一个学生的成绩，首先必须创建出这个学生对应的对象，然后，给对象发一个 `print_score` 消息，让对象自己把自己的数据打印出来。

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print('%s: %s' % (self.name, self.score))
```

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）。面向对象的程序写出来就像这样：

```
bart = Student('Bart Simpson', 59)
lisa = Student('Lisa Simpson', 87)
bart.print_score()
lisa.print_score()
```

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。Class是一种抽象概念，比如我们定义的Class——Student，是指学生这个概念，而实例（Instance）则是一个个具体的Student，比如，Bart Simpson和Lisa Simpson是两个具体的Student。

所以，面向对象的设计思想是抽象出Class，根据Class创建Instance。

面向对象的抽象程度又比函数要高，因为一个Class既包含数据，又包含操作数据的方法。

小结

- 数据封装、继承和多态是面向对象的三大特点，我们后面会详细讲解。

类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记类是抽象的模板，比如Student类，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

仍以Student类为例，在Python中，定义类是通过class关键字：

```
class Student(object):
    pass
```

class后面紧接着是类名，即Student，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来的，继承的概念我们后面再讲，通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类。

定义好了Student类，就可以根据Student类创建出Student的实例，创建实例是通过类名+()实现的：

```
>>> bart = Student()
>>> bart
<__main__.Student object at 0x10a67a590>
>>> Student
<class '__main__.Student'>
```

可以看到，变量bart指向的就是一个Student的实例，后面的0x10a67a590是内存地址，每个object的地址都不一样，而Student本身则是一个类。

可以自由地给一个实例变量绑定属性，比如，给实例bart绑定一个name属性：

```
>>> bart.name = 'Bart Simpson'
>>> bart.name
'Bart Simpson'
```

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的 `__init__` 方法，在创建实例的时候，就把 `name`，`score` 等属性绑上去：

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score
```

注意：特殊方法“**init**”前后分别有两个下划线！！

注意到 `__init__` 方法的第一个参数永远是 `self`，表示创建的实例本身，因此，在 `__init__` 方法内部，就可以把各种属性绑定到 `self`，因为 `self` 就指向创建的实例本身。

有了 `__init__` 方法，在创建实例的时候，就不能传入空的参数了，必须传入与 `__init__` 方法匹配的参数，但 `self` 不需要传，Python 解释器自己会把实例变量传进去：

```
>>> bart = Student('Bart Simpson', 59)
>>> bart.name
'Bart Simpson'
>>> bart.score
59
```

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量 `self`，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区别，所以，你仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

数据封装

面向对象编程的一个重要特点就是数据封装。在上面的 `Student` 类中，每个实例就拥有各自的 `name` 和 `score` 这些数据。我们可以通过函数来访问这些数据，比如打印一个学生的成绩：

```
>>> def print_score(std):
...     print('%s: %s' % (std.name, std.score))
...
>>> print_score(bart)
Bart Simpson: 59
```

但是，既然 `Student` 实例本身就拥有这些数据，要访问这些数据，就没有必要从外面的函数去访问，可以直接在 `Student` 类的内部定义访问数据的函数，这样，就把“数据”给封装起来了。这些封装数据的函数是和 `Student` 类本身是关联起来的，我们称之为类的方法：

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score

    def print_score(self):
        print('%s: %s' % (self.name, self.score))
```

要定义一个方法，除了第一个参数是 `self` 外，其他和普通函数一样。要调用一个方法，只需要在实例变量上直接调用，除了 `self` 不用传递，其他参数正常传入：

```
>>> bart.print_score()
Bart Simpson: 59
```

这样一来，我们从外部看 `Student` 类，就只需要知道，创建实例需要给出 `name` 和 `score`，而如何打印，都是在 `Student` 类的内部定义的，这些数据和逻辑被“封装”起来了，调用很容易，但却不用知道内部实现的细节。

封装的另一个好处是可以给 `Student` 类增加新的方法，比如 `get_grade`：

```
class Student(object):
    ...

    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'
```

同样的，`get_grade` 方法可以直接在实例变量上调用，不需要知道内部实现细节：

```
# -*- coding: utf-8 -*-
class Student(object):
    def __init__(self, name, score):
        self.name = name
        self.score = score

    def get_grade(self):
        if self.score >= 90:
            return 'A'
        elif self.score >= 60:
            return 'B'
        else:
            return 'C'
```

```
lisa = Student('Lisa', 99)
bart = Student('Bart', 59)
print(lisa.name, lisa.get_grade())
print(bart.name, bart.get_grade())
```

小结

- 类是创建实例的模板，而实例则是一个一个具体的对象，各个实例拥有的数据都互相独立，互不影响；
- 方法就是与实例绑定的函数，和普通函数不同，方法可以直接访问实例的数据；
- 通过在实例上调用方法，我们就直接操作了对象内部的数据，但无需知道方法内部的实现细节。

和静态语言不同，Python允许对实例变量绑定任何数据，也就是说，对于两个实例变量，虽然它们都是同一个类的不同实例，但拥有的变量名称都可能不同：

```
>>> bart = Student('Bart Simpson', 59)
>>> lisa = Student('Lisa Simpson', 87)
>>> bart.age = 8
>>> bart.age
8
>>> lisa.age
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'age'
```

访问限制

在Class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，这样，就隐藏了内部的复杂逻辑。

但是，从前面Student类的定义来看，外部代码还是可以自由地修改一个实例的name、score属性：

```
>>> bart = Student('Bart Simpson', 59)
>>> bart.score
59
>>> bart.score = 99
>>> bart.score
99
```

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线`__`，在Python中，实例的变量名如果以`__`开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问，所以，我们把Student类改一改：

```
class Student(object):

    def __init__(self, name, score):
        self.__name = name
        self.__score = score

    def print_score(self):
        print('%s: %s' % (self.__name, self.__score))
```

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问实例变量`__name`和实例变量`__score`了：

```
>>> bart = Student('Bart Simpson', 59)
>>> bart.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute '__name'
```

这样就确保了外部代码不能随意修改对象内部的状态，这样通过访问限制的保护，代码更加健壮。

但是如果外部代码要获取name和score怎么办？可以给Student类增加`get_name`和`get_score`这样的方法：

```
class Student(object):

    ...

    def get_name(self):
        return self.__name

    def get_score(self):
        return self.__score
```

如果又要允许外部代码修改score怎么办？可以再给Student类增加`set_score`方法：

```
class Student(object):
    ...

    def set_score(self, score):
        self.__score = score
```

你也许会问，原先那种直接通过 `bart.score = 99` 也可以修改啊，为什么要定义一个方法大费周折？因为在方法中，可以对参数做检查，避免传入无效的参数：

```
class Student(object):
    ...

    def set_score(self, score):
        if 0 <= score <= 100:
            self.__score = score
        else:
            raise ValueError('bad score')
```

需要注意的是，在Python中，变量名类似 `__xxx__` 的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用 `__name__`、`__score__` 这样的变量名。

有些时候，你会看到以一个下划线开头的实例变量名，比如 `_name`，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。

双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问 `__name` 是因为Python解释器对外把 `__name` 变量改成了 `_Student__name`，所以，仍然可以通过 `_Student__name` 来访问 `__name` 变量：

```
>>> bart._Student__name
'Bart Simpson'
```

但是强烈建议你不要这么干，因为不同版本的Python解释器可能会把 `__name` 改成不同的变量名。

总的来说就是，Python本身没有任何机制阻止你干坏事，一切全靠自觉。

最后注意下面的这种 *错误* 写法：

```
>>> bart = Student('Bart Simpson', 59)
>>> bart.get_name()
'Bart Simpson'
>>> bart.__name = 'New Name' # 设置__name变量！
>>> bart.__name
'New Name'
```

表面上看，外部代码“成功”地设置了 `__name` 变量，但实际上这个 `__name` 变量和 `class` 内部的 `__name` 变量 不是一个变量！内部的 `__name` 变量已经被Python解释器自动改成了 `_Student__name`，而外部代码给 `bart` 新增了一个 `__name` 变量。不信试试：

```
>>> bart.get_name() # get_name()内部返回self.__name
'Bart Simpson'
```

练习

请把下面的 `Student` 对象的 `gender` 字段对外隐藏起来，用 `get_gender()` 和 `set_gender()` 代替，并检查参数有效性：

```
# -*- coding: utf-8 -*-
class Student(object):
    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
```

```
# 测试：
bart = Student('Bart', 'male')
if bart.get_gender() != 'male':
    print('测试失败!')
else:
    bart.set_gender('female')
    if bart.get_gender() != 'female':
        print('测试失败!')
    else:
        print('测试成功!')
```

继承和多态

在OOP程序设计中，当我们定义一个 `class` 的时候，可以从某个现有的 `class` 继承，新的 `class` 称为子类（Subclass），而被继承的 `class` 称为基类、父类或超类（Base class、Super class）。

比如，我们已经编写了一个名为 `Animal` 的 `class`，有一个 `run()` 方法可以直接打印：

```
class Animal(object):
    def run(self):
        print('Animal is running...')
```

当我们需要编写 `Dog` 和 `Cat` 类时，就可以直接从 `Animal` 类继承：

```
class Dog(Animal):
    pass

class Cat(Animal):
    pass
```


对于 `Dog` 来说，`Animal` 就是它的父类，对于 `Animal` 来说，`Dog` 就是它的子类。
`Cat` 和 `Dog` 类似。

继承有什么好处？最大的好处是子类获得了父类的全部功能。由于 `Animal` 实现了 `run()` 方法，因此，`Dog` 和 `Cat` 作为它的子类，什么事也没干，就自动拥有了 `run()` 方法：

```
dog = Dog()
dog.run()

cat = Cat()
cat.run()
```

运行结果如下：

```
Animal is running...
Animal is running...
```

当然，也可以对子类增加一些方法，比如 `Dog` 类：

```
class Dog(Animal):

    def run(self):
        print('Dog is running...')

    def eat(self):
        print('Eating meat...')
```

继承的第二个好处需要我们对代码做一点改进。你看到了，无论是 `Dog` 还是 `Cat`，它们 `run()` 的时候，显示的都是 `Animal is running...`，符合逻辑的做法是分别显示 `Dog is running...` 和 `Cat is running...`，因此，对 `Dog` 和 `Cat` 类改进如下：

```
class Dog(Animal):

    def run(self):
        print('Dog is running...')

class Cat(Animal):

    def run(self):
        print('Cat is running...')
```

再次运行，结果如下：

```
Dog is running...
Cat is running...
```

当子类 and 父类都存在相同的 `run()` 方法时，我们说，子类的 `run()` 覆盖了父类的 `run()`，在代码运行的时候，总是会调用子类的 `run()`。这样，我们就获得了继承的另一个好处：多态。

要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个 `class` 的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和 Python 自带的数据类型，比如 `str`、`list`、`dict` 没什么两样：

```
a = list() # a是list类型
b = Animal() # b是Animal类型
c = Dog() # c是Dog类型
```

判断一个变量是否是某个类型可以用 `isinstance()` 判断：

```
>>> isinstance(a, list)
True
>>> isinstance(b, Animal)
True
>>> isinstance(c, Dog)
True
```

看来 `a`、`b`、`c` 确实对应着 `list`、`Animal`、`Dog` 这3种类型。

但是等等，试试：

```
>>> isinstance(c, Animal)
True
```

看来 `c` 不仅仅是 `Dog`，`c` 还是 `Animal`！

不过仔细想想，这是有道理的，因为 `Dog` 是从 `Animal` 继承下来的，当我们创建了一个 `Dog` 的实例 `c` 时，我们认为 `c` 的数据类型是 `Dog` 没错，但 `c` 同时也是 `Animal` 也没错，`Dog` 本来就是 `Animal` 的一种！

所以，在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行：

```
>>> b = Animal()
>>> isinstance(b, Dog)
False
```

`Dog` 可以看成 `Animal`，但 `Animal` 不可以看成 `Dog`。

要理解多态的好处，我们还需要再编写一个函数，这个函数接受一个 `Animal` 类型的变量：

```
def run_twice(animl):
    animl.run()
    animl.run()
```

当我们传入 `Animal` 的实例时，`run_twice()` 就打印出：

```
>>> run_twice(Animal())
Animal is running...
Animal is running...
```

当我们传入 `Dog` 的实例时，`run_twice()` 就打印出：

```
>>> run_twice(Dog())
Dog is running...
Dog is running...
```

当我们传入 `Cat` 的实例时，`run_twice()` 就打印出：

```
>>> run_twice(Cat())
Cat is running...
Cat is running...
```

看上去没啥意思，但是仔细想想，现在，如果我们再定义一个 `Tortoise` 类型，也从 `Animal` 派生：

```
class Tortoise(Animal):
    def run(self):
        print('Tortoise is running slowly...')
```

当我们调用 `run_twice()` 时，传入 `Tortoise` 的实例：

```
>>> run_twice(Tortoise())
Tortoise is running slowly...
Tortoise is running slowly...
```

你会发现，新增一个 `Animal` 的子类，不必对 `run_twice()` 做任何修改，实际上，任何依赖 `Animal` 作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

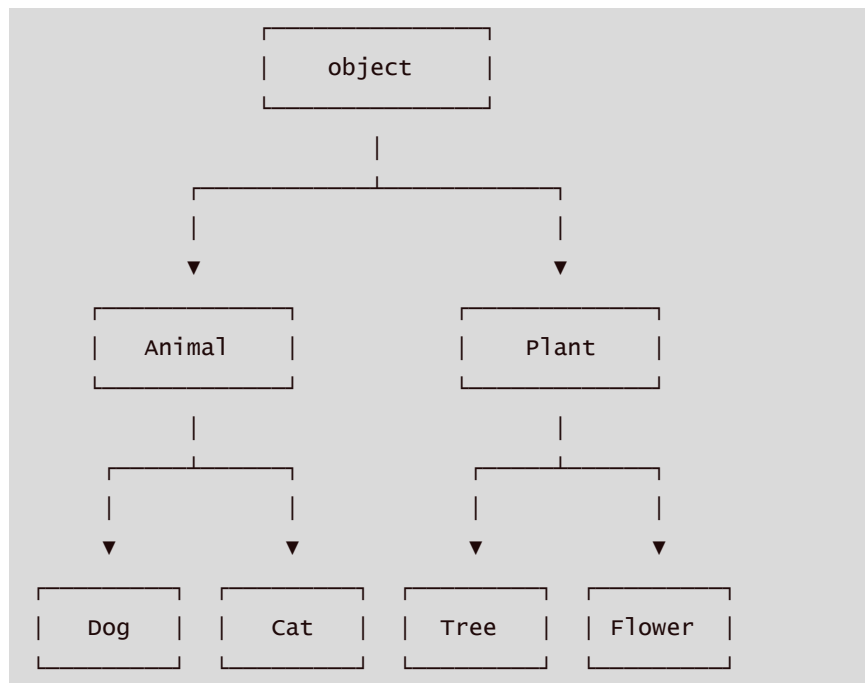
多态的好处就是，当我们需要传入 `Dog`、`Cat`、`Tortoise`.....时，我们只需要接收 `Animal` 类型就可以了，因为 `Dog`、`Cat`、`Tortoise`.....都是 `Animal` 类型，然后，按照 `Animal` 类型进行操作即可。由于 `Animal` 类型有 `run()` 方法，因此，传入的任意类型，只要是 `Animal` 类或者子类，就会自动调用实际类型的 `run()` 方法，这就是多态的意思：

对于一个变量，我们只需要知道它是 `Animal` 类型，无需确切地知道它的子类类型，就可以放心地调用 `run()` 方法，而具体调用的 `run()` 方法是作用在 `Animal`、`Dog`、`Cat` 还是 `Tortoise` 对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种 `Animal` 的子类时，只要确保 `run()` 方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则：

对扩展开放：允许新增 `Animal` 子类：

对修改封闭：不需要修改依赖 `Animal` 类型的 `run_twice()` 等函数。

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类 `object`，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：



静态语言 vs 动态语言

对于静态语言（例如 `Java`）来说，如果需要传入 `Animal` 类型，则传入的对象必须是 `Animal` 类型或者它的子类，否则，将无法调用 `run()` 方法。

对于 `Python` 这样的动态语言来说，则不一定需要传入 `Animal` 类型。我们只需要保证传入的对象有一个 `run()` 方法就可以了：

```
class Timer(object):
    def run(self):
        print('start...')
```

这就是动态语言的“鸭子类型”，它并不要求严格的继承体系，一个对象只要“看起来像鸭子，走起路来像鸭子”，那它就可以被看做是鸭子。

`Python` 的“file-like object”就是一种鸭子类型。对真正的文件对象，它有一个 `read()` 方法，返回其内容。但是，许多对象，只要有 `read()` 方法，都被视为“file-like object”。许多函数接收的参数就是“file-like object”，你不仅要传入真正的文件对象，完全可以传入任何实现了 `read()` 方法的对象。

小结

- 继承可以把父类的所有功能都直接拿过来，这样就不必重零做起，子类只需要新增自己特有的方法，也可以把父类不适合的方法覆盖重写。

- 动态语言的鸭子类型特点决定了继承不像静态语言那样是必须的。

获取对象信息

当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

使用type()

首先，我们来判断对象类型，使用 `type()` 函数：

基本类型都可以用 `type()` 判断：

```
>>> type(123)
<class 'int'>
>>> type('str')
<class 'str'>
>>> type(None)
<type(None) 'NoneType'>
```

如果一个变量指向函数或者类，也可以用 `type()` 判断：

```
>>> type(abs)
<class 'builtin_function_or_method'>
>>> type(a)
<class '__main__.Animal'>
```

但是 `type()` 函数返回的是什么呢？它返回对应的Class类型。如果我们要在 `if` 语句中判断，就需要比较两个变量的type类型是否相同：

```
>>> type(123)==type(456)
True
>>> type(123)==int
True
>>> type('abc')==type('123')
True
>>> type('abc')==str
True
>>> type('abc')==type(123)
False
```

判断基本数据类型可以直接写 `int`，`str` 等，但如果要判断一个对象是否是函数怎么办？可以使用 `types` 模块中定义的常量：

```
>>> import types
>>> def fn():
...     pass
...
>>> type(fn)==types.FunctionType
True
>>> type(abs)==types.BuiltinFunctionType
True
>>> type(lambda x: x)==types.LambdaType
True
>>> type((x for x in range(10)))==types.GeneratorType
True
```

使用isinstance()

对于class的继承关系来说，使用 `type()` 就很不方便。我们要判断class的类型，可以使用 `isinstance()` 函数。

我们回顾上次的例子，如果继承关系是：

```
object -> Animal -> Dog -> Husky
```

那么，`isinstance()` 就可以告诉我们，一个对象是否是某种类型。先创建3种类型的对象：

```
>>> a = Animal()
>>> d = Dog()
>>> h = Husky()
```

然后，判断：

```
>>> isinstance(h, Husky)
True
```

没有问题，因为 `h` 变量指向的就是 `Husky` 对象。

再判断：

```
>>> isinstance(h, Dog)
True
```

`h` 虽然自身是 `Husky` 类型，但由于 `Husky` 是从 `Dog` 继承下来的，所以，`h` 也还是 `Dog` 类型。换句话说，`isinstance()` 判断的是一个对象是否是该类型本身，或者位于该类型的父继承链上。

因此，我们可以确信，`h` 还是 `Animal` 类型：

```
>>> isinstance(h, Animal)
True
```

同理，实际类型是Dog的d也是Animal类型：

```
>>> isinstance(d, Dog) and isinstance(d, Animal)
True
```

但是，d不是Husky类型：

```
>>> isinstance(d, Husky)
False
```

能用type()判断的基本类型也可以用isinstance()判断：

```
>>> isinstance('a', str)
True
>>> isinstance(123, int)
True
>>> isinstance(b'a', bytes)
True
```

并且还可以判断一个变量是否是某些类型中的一种，比如下面的代码就可以判断是否是list或者tuple：

```
>>> isinstance([1, 2, 3], (list, tuple))
True
>>> isinstance((1, 2, 3), (list, tuple))
True
```

总是优先使用isinstance()判断类型，可以将指定类型及其子类“一网打尽”。

使用dir()

如果要获得一个对象的所有属性和方法，可以使用dir()函数，它返回一个包含字符串的list，比如，获得一个str对象的所有属性和方法：

```
>>> dir('ABC')
['__add__', '__class__', ..., '__subclasshook__',
'capitalize', 'casefold', ..., 'zfill']
```

类似__xxx__的属性和方法在Python中都是有特殊用途的，比如__len__方法返回长度。在Python中，如果你调用len()函数试图获取一个对象的长度，实际上，在len()函数内部，它自动去调用该对象的__len__()方法，所以，下面的代码是等价的：

```
>>> len('ABC')
3
>>> 'ABC'.__len__()
3
```

我们自己写的类，如果也想用 `len(myObj)` 的话，就自己写一个 `__len__()` 方法：

```
>>> class MyDog(object):
...     def __len__(self):
...         return 100
...
>>> dog = MyDog()
>>> len(dog)
100
```

剩下的都是普通属性或方法，比如 `lower()` 返回小写的字符串：

```
>>> 'ABC'.lower()
'abc'
```

仅仅把属性和方法列出来是不够的，配合 `getattr()`、`setattr()` 以及 `hasattr()`，我们可以直接操作一个对象的状态：

```
>>> class MyObject(object):
...     def __init__(self):
...         self.x = 9
...     def power(self):
...         return self.x * self.x
...
>>> obj = MyObject()
```

紧接着，可以测试该对象的属性：

```
>>> hasattr(obj, 'x') # 有属性'x'吗?
True
>>> obj.x
9
>>> hasattr(obj, 'y') # 有属性'y'吗?
False
>>> setattr(obj, 'y', 19) # 设置一个属性'y'
>>> hasattr(obj, 'y') # 有属性'y'吗?
True
>>> getattr(obj, 'y') # 获取属性'y'
19
>>> obj.y # 获取属性'y'
19
```

如果试图获取不存在的属性，会抛出 `AttributeError` 的错误：

```
>>> getattr(obj, 'z') # 获取属性'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyObject' object has no attribute 'z'
```


可以传入一个default参数，如果属性不存在，就返回默认值：

```
>>> getattr(obj, 'z', 404) # 获取属性'z'，如果不存在，返回默认值404
404
```

也可以获得对象的方法：

```
>>> hasattr(obj, 'power') # 有属性'power'吗?
True
>>> getattr(obj, 'power') # 获取属性'power'
<bound method MyObject.power of <__main__.MyObject object at 0x10077a6a0>>
>>> fn = getattr(obj, 'power') # 获取属性'power'并赋值到变量fn
fn
>>> fn # fn指向obj.power
<bound method MyObject.power of <__main__.MyObject object at 0x10077a6a0>>
>>> fn() # 调用fn()与调用obj.power()是一样的
81
```

小结

通过内置的一系列函数，我们可以对任意一个Python对象进行剖析，拿到其内部的数据。要注意的是，只有在不知道对象信息的时候，我们才会去获取对象信息。如果可以直接写：

```
sum = obj.x + obj.y
```

就不要写：

```
sum = getattr(obj, 'x') + getattr(obj, 'y')
```

一个正确的用法的例子如下：

```
def readImage(fp):
    if hasattr(fp, 'read'):
        return readData(fp)
    return None
```

假设我们希望通过文件流fp中读取图像，我们首先要判断该fp对象是否存在read方法，如果存在，则该对象是一个流，如果不存在，则无法读取。`hasattr()`就派上了用场。

请注意，在Python这类动态语言中，根据鸭子类型，有`read()`方法，不代表该fp对象就是一个文件流，它也可能是网络流，也可能是内存中的一个字节流，但只要`read()`方法返回的是有效的图像数据，就不影响读取图像的功能。

实例属性和类属性

由于Python是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过`self`变量：

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

s = Student('Bob')
s.score = 90
```

但是，如果`Student`类本身需要绑定一个属性呢？可以直接在`class`中定义属性，这种属性是类属性，归`Student`类所有：

```
class Student(object):
    name = 'Student'
```

当我们定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到。来测试一下：

```
>>> class Student(object):
...     name = 'Student'
...
>>> s = Student() # 创建实例s
>>> print(s.name) # 打印name属性，因为实例并没有name属性，所以会
继续查找class的name属性
Student
>>> print(Student.name) # 打印类的name属性
Student
>>> s.name = 'Michael' # 给实例绑定name属性
>>> print(s.name) # 由于实例属性优先级比类属性高，因此，它会屏蔽掉
类的name属性
Michael
>>> print(Student.name) # 但是类属性并未消失，用Student.name仍
然可以访问
Student
>>> del s.name # 如果删除实例的name属性
>>> print(s.name) # 再次调用s.name，由于实例的name属性没有找到，
类的name属性就显示出来了
Student
```

从上面的例子可以看出，在编写程序的时候，千万不要对实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性。

练习

为了统计学生人数，可以给Student类增加一个类属性，每创建一个实例，该属性自动增加：

```
# -*- coding: utf-8 -*-
class Student(object):
    count = 0

    def __init__(self, name):
        self.name = name

# 测试:
if Student.count != 0:
    print('测试失败!')
else:
    bart = Student('Bart')
    if Student.count != 1:
        print('测试失败!')
    else:
        lisa = Student('Bart')
        if Student.count != 2:
            print('测试失败!')
        else:
            print('Students:', Student.count)
            print('测试通过!')
```

小结

- 实例属性属于各个实例所有，互不干扰；
- 类属性属于类所有，所有实例共享一个属性；
- 不要对实例属性和类属性使用相同的名字，否则将产生难以发现的错误。