面向对象编程

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

面向过程编程(Procedure Oriented Programming, DOP)，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，DOP把函数细分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。
- OOP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OOP中的类（Class）的概念。

我们以一个例子来说明DOP和OOP在流程上的不同之处。

假设我们要处理学生的成绩表。

1. 如果使用DOP，为表示一个学生的成绩，可以用一个dict表示：

In [2]:
```
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：

In [6]:
```
def print_score(std):
    print('%s: %s' % (std['name'], std['score']))

print_score(std1)
print_score(std2)
```

```
Michael: 98
Bob: 81
```

1. 如果采用OOP，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，它拥有name和score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，让对象把自己的数据打印出来。

In [2]:
```
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：

In [3]:
```
bart = Student('Bart Simpson', 59)
lisa = Student('Lisa Simpson', 87)
bart.print_score()
lisa.print_score()
```

```
Bart Simpson: 59
Lisa Simpson: 87
```

OOP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，也就是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，OOP的设计思想是抽象出Class，根据Class创建Instance。

OOP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。

面向对象的三大特点：

- 数据封装
- 继承
- 多态

类和实例

面向对象最重要的概念就是

- 类 (Class)
- 实例 (Instance)

必须牢记：

- 类是抽象的模板，比如Student类;
- 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：

In [2]:
```
class Student(object):
    pass
```

定义类，用关键字class，步骤如下：

- class后面紧接着是类名，即Student。

  类名通常以大写开头

- 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。

创建实例通过“类名+()”来实现。

In [3]:
```
bart = Student()
print(bart)
print(Student)
```

```
<__main__.Student object at 0x7f24d4119518>
<class '__main__.Student'>
```

- 变量bart指向的就是一个Student的实例，后面的0x7f5bb854c110是内存地址，每个object的地址都不一样；
- 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：

In [9]:
```
bart.name = 'Bart Simpson'
print(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，创建实例时，就可以把name，score等属性绑上去：

In [45]:
```
class Student(object):

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

注意：特殊方法“**__init__**”前后有两个下划线！！！

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，创建实例时，

- 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。

In [46]:
```
bart = Student('Bart Simpson', 59)
print(bart.name)
print(bart.score)
```

```
Bart Simpson
59
```

和普通的函数相比，在类中定义的函数只有一点不同，即：

第一个参数永远是**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

OOP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：

In [47]:
```
def print_score(std):
    print('%s: %s' % (std.name, std.score))

print_score(bart)
```

```
Bart Simpson: 59
```

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。

In [ ]:
```
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：

In [28]:
```
bart = Student('Bart Simpson', 59)
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；
- 至于如何打印，在Student类中定义。

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：

In [ ]:
```
class Student(object):
    ...

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

In [30]:
```
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))

    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：

In [32]:
```
bart = Student('Bart Simpson', 59)
bart.get_grade()
```

Out[32]:
```
'C'
```

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：

In [33]:
```
bart = Student('Bart Simpson', 59)
lisa = Student('Lisa Simpson', 87)
bart.age = 8
bart.age
8
lisa.age
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-33-d95089d1f6f0> in <module>()
      4 bart.age
      5 8
----> 6 lisa.age

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：

In [35]:
```
bart = Student('Bart Simpson', 98)
print(bart.score)
bart.score = 59
print(bart.score)
```

```
98
59
```

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：

In [10]:
```
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：

In [13]:
```
bart = Student('Bart Simpson', 98)
print(bart.__name)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-13-fdb65aeaa12d> in <module>()
      1 bart = Student('Bart Simpson', 98)
----> 2 print(bart.__name)

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。

In [6]:
```
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))
    def get_name(self):
        return self.__name
    def get_score(self):
        return self.__score
```

In [7]:
```
bart = Student('Bart Simpson', 98)
print(bart.get_name())
print(bart.get_score())
```

```
Bart Simpson
98
```

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：

In [8]:
```
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))
    def get_name(self):
        return self.__name
    def get_score(self):
        return self.__score
    def set_score(self, score):
        self.__score = score
```

In [9]:
```
bart = Student('Bart Simpson', 98)
print(bart.get_score())
bart.set_score(100)
print(bart.get_score())
```

```
98
100
```

你也许会问，原先那种直接通过bart.score = 59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。

In [14]:
```
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))
    def get_name(self):
        return self.__name
    def get_score(self):
        return self.__score
    def set_score(self, score):
        if 0 <= score <= 100:
            self.__score = score
        else:
            raise ValueError('bad score')
```

In [15]:
```
bart = Student('Bart Simpson', 98)
bart.set_score(101)
print(bart.get_score())
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-15-d7d30a7adbba> in <module>()
      1 bart = Student('Bart Simpson', 98)
----> 2 bart.set_score(101)
      3 print(bart.get_score())

<ipython-input-14-eea158d9d6b7> in set_score(self, score)
     13             self.__score = score
     14         else:
---> 15             raise ValueError('bad score')

ValueError: bad score
```

一些有意思的变量形式

1. __xxx：以两个下划线开头，会被限制访问。

1. __xxx__：以两个下划线开头，两个下划线结尾，是特殊变量

   特殊变量是可以直接访问的，不是private变量。所以，访问限制时，不能用__name__、__score__这样的变量名。
2. _xxx：以一个下划线开头

   这样的实例变量外部可以访问。但是，按照约定俗成的规定，当你看到这样的变量时，它会向你倾诉：“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”。

__xxx真的不能从外部访问呢？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：

In [46]:
```
bart._Student__name
```

Out[46]: '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：

In [47]:
```
bart = Student('Bart Simpson', 98)
bart.get_name()
bart.__name = 'New Name' # 设置__name变量！
bart.__name
```

Out[47]: '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。

In [48]:
```
bart.get_name() # get_name()内部返回self.__name
```

Out[48]: 'Bart Simpson'

继承和多态

在OOP中，定义一个class时，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，假设已经定义了一个名为Animal的class，有一个run()方法可以直接打印：

In [18]:
```
class Animal(object):
    def run(self):
        print('Animal is running...')
```

定义Dog和Cat类时，就可以直接从Animal类继承：

In [19]:
```
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Animial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：

In [20]:
```
dog = Dog()
dog.run()

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

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

当然，也可以对子类增加一些方法，比如Dog类：

In [21]:
```
class Dog(Animal):

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

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

要了解继承的第二个好处，我们先对代码做一点改进。

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...

而符合逻辑的做法应该分别显示

Dog is running...

Cat is running...

对Dog和Cat类改进如下：

In [22]:
```
class Dog(Animal):
    def run(self):
        print('Dog is running...')

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

In [19]:
```
dog = Dog()
dog.run()

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

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

当子类和父类都存在相同的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：

In [23]:
```
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

In [24]:
```
print(isinstance(a, list))
print(isinstance(b, Animal))
print(isinstance(c, Dog))
```

```
True
True
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：

In [25]:
```
print(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。

当创建了一个Dog的实例时，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：`

In [63]:
```
b = Animal()
print(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：

In [64]:
```
def run_twice(animal):
    animal.run()
    animal.run()
```

当我们传入Animal的实例时，run_twice()就打印出：

In [65]:
```
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：

In [66]:
```
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：

In [67]:
```
run_twice(Cat())
```

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

看上去没啥意思 :(

但如果我们再定义一个Tortoise类型，也从Animal派生：

In [68]:
```
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：

In [69]:
```
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：

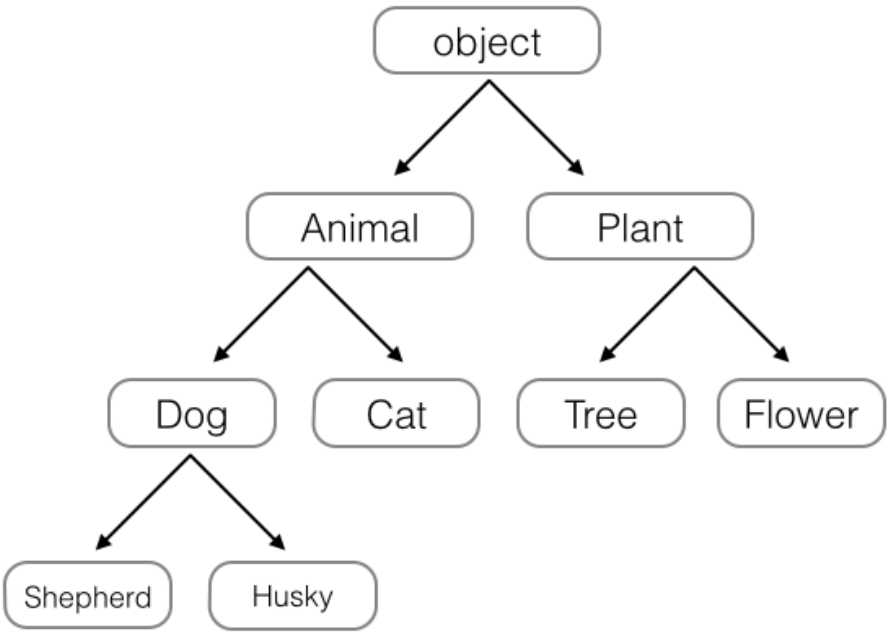
调用方只管调用，不管细节

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：

In [80]:
```
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“，你不一定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()**

判断对象类型，请使用**type()**：

- 基本类型都可以用type()判断：

In [83]:
```
print(type(123))
print(type('str'))
print(type(None))
```

```
<type 'int'>
<type 'str'>
<type 'NoneType'>
```

- 如果一个变量指向函数或者类，也可以用type()判断：

In [87]:
```
print(type(abs))
a = Animal()
print(type(a))
```

```
<type 'builtin_function_or_method'>
<class '__main__.Animal'>
```

type()返回的是什么类型呢？

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：

In [88]:
```
print(type(123)==type(456))
print(type(123)==int)
print(type('abc')==type('123'))
print(type('abc')==str)
print(type('abc')==type(123))
```

```
True
True
True
True
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。

In [20]:
```
import types
def fn():
    pass

print(type(fn)==types.FunctionType)
print(type(abs)==types.BuiltinFunctionType)
print(type(lambda x: x)==types.LambdaType)
print(type((x for x in range(10)))==types.GeneratorType)
```

```
True
True
True
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()**。

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：

In [21]:
```
class Animal(object):
    pass

class Dog(Animal):
    pass

class Husky(Dog):
    pass
```

In [22]:
```
a = Animal()
d = Dog()
h = Husky()

print(isinstance(h, Husky))
print(isinstance(h, Dog))
print(isinstance(h, Animal))
```

```
True
True
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。 当然，h还是Animal类型。

In [99]:
```
print(isinstance(d, Dog) and isinstance(d, Animal))
print(isinstance(d, Husky))
```

```
True
False
```

能用type()判断的基本类型也可以用isinstance()判断：

In [101]:
```
print(isinstance('a', str))
print(isinstance(123, int))
print(isinstance(b'a', bytes))
```

```
True
True
True
```

isinstance() 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：

In [103]:
```
print(isinstance([1, 2, 3], (list, tuple)))
print(isinstance((1, 2, 3), (list, tuple)))
```

```
True
True
```

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：

In [105]:
```
dir('ABC')
```

```
Out[105]: ['__add__',
           '__class__',
           '__contains__',
           '__delattr__',
           '__doc__',
           '__eq__',
           '__format__',
           '__ge__',
           '__getattribute__',
           '__getitem__',
           '__getnewargs__',
           '__getslice__',
           '__gt__',
           '__hash__',
           '__init__',
           '__le__',
           '__len__',
           '__lt__',
           '__mod__',
           '__mul__',
           '__ne__',
           '__new__',
           '__reduce__',
           '__reduce_ex__',
           '__repr__',
           '__rmod__',
           '__rmul__',
           '__setattr__',
           '__sizeof__',
           '__str__',
           '__subclasshook__',
           '_formatter_field_name_split',
           '_formatter_parser',
           'capitalize',
           'center',
           'count',
           'decode',
           'encode',
           'endswith',
           'expandtabs',
           'find',
           'format',
           'index',
           'isalnum',
           'isalpha',
           'isdigit',
           'islower',
           'isspace',
           'istitle',
           'isupper',
           'join',
           'ljust',
           'lower',
           'lstrip',
           'partition',
           'replace',
           'rfind',
           'rindex',
           'rjust',
           'rpartition',
           'rsplit',
           'rstrip',
           'split',
           'splitlines',
           'startswith',
           'strip',
           'swapcase',
           'title',
```

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：

In [106]:
```
print(len('ABC'))
print('ABC'.__len__())
```

```
3
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：

In [110]:
```
class MyDog(object):
    def __len__(self):
        return 100

dog = MyDog()
print(len(dog))
print(dog.__len__())
```

```
100
100
```

剩下的都是普通属性或方法，比如lower()返回小写的字符串：

In [111]:
```
'ABC'.lower()
```

Out[111]:
```
'abc'
```

仅仅把属性和方法列出来是不够的，配合

- getattr()
- setattr()
- hasattr()

我们可以直接操作一个对象的状态。

In [30]:
```
class MyObject(object):

    def __init__(self):
        self.x = 9

    def power(self):
        return self.x * self.x
```

In [34]:
```
obj = MyObject()
print( hasattr(obj, 'x') )       # 有属性'x'吗？
print( getattr(obj, 'x') )       # 获取属性'x'
print( obj.x )                   # 获取属性'x'
```

```
True
9
9
```

如果试图获取不存在的属性，会抛出AttributeError的错误：

In [37]:
```
print( getattr(obj, 'z') )          # 获取属性'z'
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-37-9740960fd131> in <module>()
----> 1 print( getattr(obj, 'z') )          # 获取属性'z'

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：

In [38]:
```
print( 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：

In [39]:
```
print( hasattr(obj, 'power') )# 有属性'power'吗？
print( getattr(obj, 'power') )# 获取属性'power'
fn = getattr(obj, 'power')  # 获取属性'power'并赋值到变量fn
print( fn() )                 # 调用fn()与调用obj.power()是一样的
```

```
True
<bound method MyObject.power of <__main__.MyObject object at 0x7f24d40d24a8>>
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：

In [132]:
```
sum = obj.x + obj.y
```

就不要写：

In [133]:
```
sum = getattr(obj, 'x') + getattr(obj, 'y')
```

一个正确的用法的例子如下

In [41]:
```
def readImage(fp):

    if hasattr(fp, 'read'):
        return readData(fp)

    return None
```

假设我们希望从文件流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：

In [42]:
```
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：

In [43]:
```
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：

In [149]:
```
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
print Student.name   # 打印类的 name 属性
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
del s.name           # 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
Student
Michael
Student
```

从上面的例子可以看出，编写程序时，

- 千万不要把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。