# 目 录

# 1 面向对象编程

类 class 是一种抽象概念，比如学生 student 就是一个抽象概念。而实例 instance 是一个个具体的 student，比如 Bart Simpson 就是一个具体的 student。

面向对象编程的设计思想是抽象出 class，然后根据 class 创建 instance。

## 1.1 类和实例

在 python 中，定义类是通过 class 关键字，如下所示：

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

这里的 object 是 student 的父类，student 继承了 object 类。一般来说，如果没有合适的继承类，就使用 object 类。

创建实例的方式如下所示：

```
bart = student('Bart Simnpson', 59)
```

需要知道的是，在类中定义的函数的第一个参数永远是实例变量 self，并且调用类函数时，不用传递这个参数。除此之外，类函数和普通函数没有什么区别。

## 1.2 访问限制

如果想让类的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 __，这样就变成了私有变量，只有内部可以访问，外部不能访问，如下所示：

```
class student(object):
    def __init__(self, name, score):
        self.__name = name
        sekf.__score = score

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

需要知道的是，变量名类似 __xxx__ 的，是特殊变量。特殊变量可以直接访问，不是 private 变量。

有时候还可以看到以下划线开头的实例变量名，如 _name，这样的变量外部可以访问。但是按照约定俗成的规定，这种变量虽然可以被访问，但是请把它当作私有变量，不要随意访问。

私有变量其实也可以访问，student 类中的 __name 只是被 python 解释器变成了 _student__name，所以仍然可以通过 _student__name 访问 __name，如下所示：

```
bart = student('bart', 100)
bart._student__name
# 输出为bart
```

需要知道的是，不同版本的 python 解释器会把 __name 改为不同的变量名，不一定是 _student__name。

## 1.3 继承和多态

### 1.3.1 继承

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，而被继承的 class 称为基类、父类或超类。

需要注意的是，任何时候，如果没有合适的类可以继承，就继承 object 类。

例子如下：

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

class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

继承以后，子类获得父类的所有功能，像 Dog 和 Cat 类，就自动拥有了 run() 方法，如下所示：

```
dog = Dog()
dog.run()
# 输出结果为'Animal is running'

cat = Cat()
cat.run()
# 输出结果为'Animal is running'
```

### 1.3.2 多态

当子类和父类都存在相同的方法时，子类的方法将覆盖父类的芳法，在代码运行时，总是会调用子类的方法。

更进一步说，任何依赖 Animal 作为参数的函数，传入任意的类型，只要是 Animal 类或者它的子类，就会自动调用实际类型的方法，这就是多态。

如下所示：

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

class Dog(Animal):
    def run(self):
        print 'Dog is running'

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

# 一个依赖Animal作为参数的函数
def run_test(animal):
    animal.run()

# 运行时会调用实际类型的方法
run_test(Animal())
# 输出'Animal is running'
run_test(Dog())
# 输出'Dog is running'
run_test(Cat())
# 输出'Cat is running'
```

## 1.4 dir() 函数

使用 dir() 函数可以获得一个对象的所有属性和方法，如下所示：

```
dir('ABC')
```

## 1.5 使用 __slot__

在 python 中，如果创建一个 class 实例以后，我们可以给该实例绑定任何的属性和方法，如下所示：

```
class student(object):
    pass

s = student()
s.name = 'Michael' # 绑定一个属性

from types import MethodType
def set_age(self, age):
    self.age = age

s.set_age = MethodType(set_age, s, student) # 绑定一个方法
```

我们可以使用 __slots__ 限制 class 的属性，比如，只允许对 student 实例添加 name 和 age 属性：

```
class student(object):
    __slot__ = ('name', 'age')
```

此时，如果向 student 的实力绑定其他属性，就会出错：

```
s = student()
s.name = 'Michael'
s.age = 25
s.score = 100 # 报错
```

需要注意的是，__slots__ 只对当前类起作用，对继承的子类是不起作用的：

```
class GraduateStudent(student):
    pass

g = GraduateStudent()
g.score = 100 # 绑定一个属性，不会报错
```

## 1.6 property 类

property 类是 python 中的类，它的声明如下所示：

```
property(fget=None, fset=None, fdel=None, doc=None)
class property([fget,[,fset[,fdel[,doc]]]])
```

通过 prproperty 函数，可以将一个方法变成属性调用。这说起来有点抽象，可以看如下用法：

```
class C(object):
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property")

c = C()
c.x # 将调用c.getx()，输出_x
c.x = value # 将调用c.setx(value)，设置_x
del c.x # 将调用c.delx()
```

可以将 property 类和 @ 语法配合使用，如下所示：

```
class C(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        return self._x

c = C()
c.x # 输出_x的值
```

此时，类 C 中就有一个 property 类的实例 x，可以通过它直接获得 _x 的值。

为了进一步了解 property，我们来看一下它的实现代码：

```
class property(object):
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fget is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel, self.__doc__)

    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel, self.__doc__)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

当类中第一次使用 @property 以后，就返回了 property 类的实例 x，此时上述例子就相当于如下的类：

```
class C(object):
```

```
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    x = property(getx)
```

根据 property 类的实现代码可以知道，x 此时还有 setter、deleter 函数可以用来做装饰器，从而进一步增加 x 的功能，如下所示：

```
class C(object):
    def __init__(self):
        self._x = None

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

## 1.7 类中特殊的函数

### 1.7.1 __str__ 函数

当打印类时，__str__ 函数可以返回用户看到的字符串，如下所示：

```
class student(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'student object (name: %s)' % self.name

print student('Michael')
# 输出结果为'student object (name: Michael)'
```

### 1.7.2 __repr__ 函数

当直接调用类时，__repr__ 函数将返回程序开发者看到的字符串，如下所示：

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

```
4         def __repr__(self):
5             return 'student object (name: %s)' % self.name
6
7     s = student('Michael')
8     s # 输出结果为'student object (name: Michael)'
```

可以看出，__str__ 和 __repr__ 的区别在于，一个要用 print 打印类，一个直接调用类。

### 1.7.3 __iter__ 函数

当类用于 for 循环时，for 循环将调用一次类的 __iter__ 函数，用于初始化被循环的迭代器。因此，__iter__ 函数必须返回一个迭代器，用于 for 循环。

如下例所示：

```
1     # 本例中Fib类本身就是一个迭代器
2     class Fib(object):
3         def __init__(self):
4             self.a, self.b = 0, 1
5
6         def __iter__(self):
7             return self
8
9         def next(self):
10            self.a, self.b = self.b, self.a + self.b
11            if self.a > 10
12                raise StopIteration()
13            return self.a
14
15    for n in Fib():
16        print n
```

### 1.7.4 __getitem__ 函数

如果想让类通过 [] 接受参数，可以使用 __getitem__ 函数：

```
1     class Fib(object):
2         def __getitem__(self, n):
3             return n
4
5     f = Fib()
6     f[0] # 输出0
7     f[1] # 输出1
8     f[2] # 输出2
```

### 1.7.5 __getattr__ 函数

正常情况下，当我们调用类的方法或属性时，如果不存在，就会报错。但是如果定义了 __getattr__ 函数，那么被调用的方法或属性不存在时，将调用 __getattr__ 函数。

举个例子：

```
class student(object):
    def __init__(self):
        self.name = 'Michael'

    def __getattr__(self, attr):
        if attr == 'score':
            return 99
        if attr == 'age':
            return lambda: 25

s = student()
s.score # 返回属性，输出结果为99
s.age() # 返回方法，输出结果为25
```

需要知道的是，如果 __getattr__ 函数中没有特殊处理，将默认返回 None。

### 1.7.6 __call__ 函数

通过定义一个 __call__() 方法，就可以直接对实例进行调用。

举个例子：

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

    def __call__(self):
        print 'My name is %s.' % self.name

s = student('Michael')
s() # 输出结果为'My name is Michael.'
```

需要知道的是，这个 __call__ 函数是可以接受参数的。

## 1.8 type()

type() 函数可以查看一个类型或变量的类型，还可以创建 class。

type() 函数既可以返回一个对象的类型，又可以创建出新的类型。具体操作如下所示：

```
def fn(self, name='world'):
    print 'Hello, %s.' % name

Hello = type('Hello', (object,), dict(hello=fn)) # 创建Hello类
```

type() 函数依次传入 3 个参数：class 的名称、tuple 形式的父类集合、与函数绑定的 class 方法名称。

通过 type() 函数创建的类和直接写 class 是完全一样的，因为 python 解释器遇到 class 定义时，也是调用 type() 函数创建出 class。