08 面向对象高级编程

数据封装、继承和多态只是面向对象程序设计中最基础的3个概念。在Python中，面向对象还有很多高级特性，允许我们写出非常强大的功能。

我们会讨论多重继承、定制类、元类等概念。

使用`__slots__`

正常情况下，当我们定义了一个class，创建了一个class的实例后，我们可以给该实例绑定任何属性和方法，这就是动态语言的灵活性。先定义class：

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

然后，尝试给实例绑定一个属性：

```
>>> s = Student()
>>> s.name = 'Michael' # 动态给实例绑定一个属性
>>> print(s.name)
Michael
```

还可以尝试给实例绑定一个方法：

```
>>> def set_age(self, age): # 定义一个函数作为实例方法
...     self.age = age
...
>>> from types import MethodType
>>> s.set_age = MethodType(set_age, s) # 给实例绑定一个方法
>>> s.set_age(25) # 调用实例方法
>>> s.age # 测试结果
25
```

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

```
>>> s2 = Student() # 创建新的实例
>>> s2.set_age(25) # 尝试调用方法
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute
'set_age'
```

为了给所有实例都绑定方法，可以给class绑定方法：

```
>>> def set_score(self, score):
...     self.score = score
...
>>> Student.set_score = set_score
```

给class绑定方法后，所有实例均可调用：

```
>>> s.set_score(100)
>>> s.score
100
>>> s2.set_score(99)
>>> s2.score
99
```

通常情况下，上面的`set_score`方法可以直接定义在class中，但动态绑定允许我们在程序运行的过程中动态给class加上功能，这在静态语言中很难实现。

使用slots

但是，如果我们想要限制实例的属性怎么办？比如，只允许对Student实例添加`name`和`age`属性。

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的`__slots__`变量，来限制该class实例能添加的属性：

```
class Student(object):
    __slots__ = ('name', 'age') # 用tuple定义允许绑定的属性名称
```

然后，我们试试：

```
>>> s = Student() # 创建新的实例
>>> s.name = 'Michael' # 绑定属性'name'
>>> s.age = 25 # 绑定属性'age'
>>> s.score = 99 # 绑定属性'score'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'score'
```

由于`'score'`没有被放到`__slots__`中，所以不能绑定`score`属性，试图绑定`score`将得到`AttributeError`的错误。

使用`__slots__`要注意，`__slots__`定义的属性仅对当前类实例起作用，对继承的子类是不起作用的：

```
>>> class GraduateStudent(Student):
...     pass
...
>>> g = GraduateStudent()
>>> g.score = 9999
```

除非在子类中也定义`__slots__`，这样，子类实例允许定义的属性就是自身的`__slots__`加上父类的`__slots__`。

使用`@property`

在绑定属性时，如果我们直接把属性暴露出去，虽然写起来很简单，但是，没办法检查参数，导致可以把成绩随便改：

```
s = Student()
s.score = 9999
```

这显然不合逻辑。为了限制score的范围，可以通过一个`set_score()`方法来设置成绩，再通过一个`get_score()`来获取成绩，这样，在`set_score()`方法里，就可以检查参数：

```
class Student(object):

    def get_score(self):
         return self._score

    def set_score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~
100!')
        self._score = value
```

现在，对任意的Student实例进行操作，就不能随心所欲地设置score了：

```
>>> s = Student()
>>> s.set_score(60) # ok!
>>> s.get_score()
60
>>> s.set_score(9999)
Traceback (most recent call last):
  ...
ValueError: score must between 0 ~ 100!
```

但是，上面的调用方法又略显复杂，没有直接用属性这么直接简单。

有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？对于追求完美的Python程序员来说，这是必须要做到的！

还记得装饰器（decorator）可以给函数动态加上功能吗？对于类的方法，装饰器一样起作用。Python内置的`@property`装饰器就是负责把一个方法变成属性调用的：

```
class Student(object):

    @property
    def score(self):
        return self._score

    @score.setter
    def score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~
100!')
        self._score = value
```

`@property`的实现比较复杂，我们先考察如何使用。把一个getter方法变成属性，只需要加上`@property`就可以了，此时，`@property`本身又创建了另一个装饰器`@score.setter`，负责把一个setter方法变成属性赋值，于是，我们就拥有一个可控的属性操作：

```
>>> s = Student()
>>> s.score = 60 # OK，实际转化为s.set_score(60)
>>> s.score # OK，实际转化为s.get_score()
60
>>> s.score = 9999
Traceback (most recent call last):
  ...
ValueError: score must between 0 ~ 100!
```

注意到这个神奇的`@property`，我们在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过getter和setter方法来实现的。

还可以定义只读属性，只定义getter方法，不定义setter方法就是一个只读属性：

```
class Student(object):

    @property
    def birth(self):
        return self._birth

    @birth.setter
    def birth(self, value):
        self._birth = value

    @property
    def age(self):
        return 2015 - self._birth
```

上面的`birth`是可读写属性，而`age`就是一个*只读*属性，因为`age`可以根据`birth`和当前时间计算出来。

小结

`@property`广泛应用在类的定义中，可以让调用者写出简短的代码，同时保证对参数进行必要的检查，这样，程序运行时就减少了出错的可能性。

练习

请利用`@property`给一个`Screen`对象加上`width`和`height`属性，以及一个只读属性`resolution`：

```
# -*- coding: utf-8 -*-
class Screen(object):
    pass
```

```
# 测试:
s = Screen()
s.width = 1024
s.height = 768
print('resolution =', s.resolution)
if s.resolution == 786432:
    print('测试通过!')
else:
    print('测试失败!')
```

多重继承

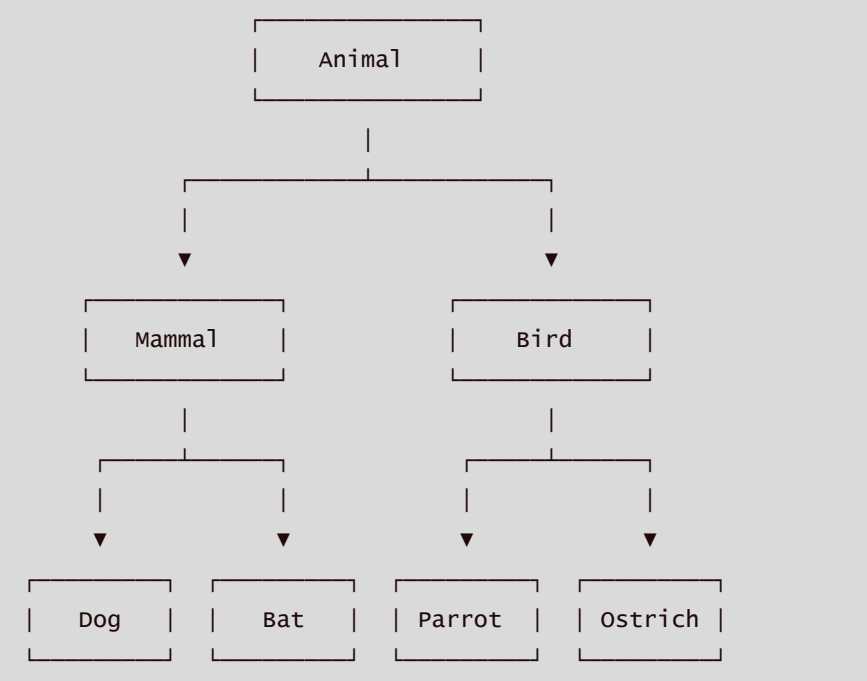
继承是面向对象编程的一个重要的方式，因为通过继承，子类就可以扩展父类的功能。

回忆一下`Animal`类层次的设计，假设我们要实现以下4种动物：

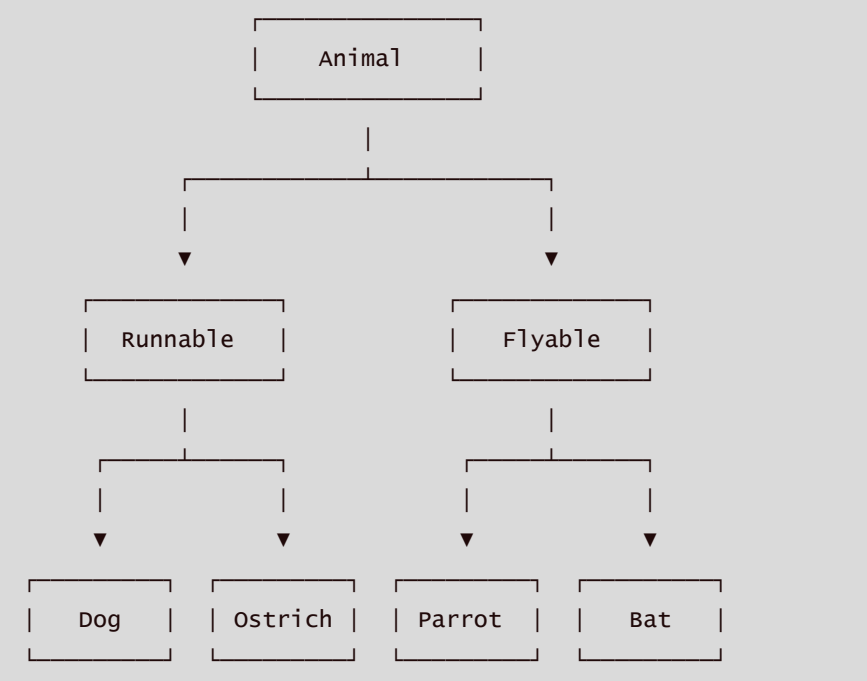
- Dog - 狗狗；

- Bat - 蝙蝠；
- Parrot - 鹦鹉；
- Ostrich - 鸵鸟。

如果按照哺乳动物和鸟类归类，我们可以设计出这样的类的层次：



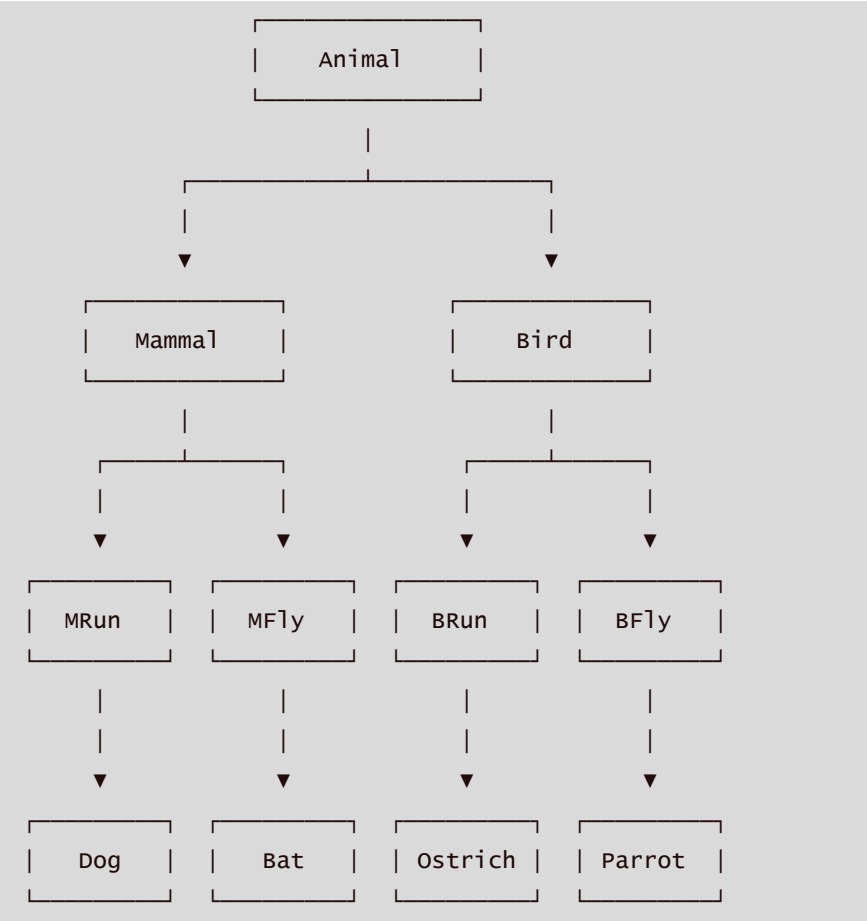
但是如果按照“能跑”和“能飞”来归类，我们就应该设计出这样的类的层次：



如果要把上面的两种分类都包含进来，我们就得设计更多的层次：

- 哺乳类：能跑的哺乳类，能飞的哺乳类；
- 鸟类：能跑的鸟类，能飞的鸟类。

这么一来，类的层次就复杂了：



如果要再增加“宠物类”和“非宠物类”，这么搞下去，类的数量会呈指数增长，很明显这样设计是不行的。

正确的做法是采用多重继承。首先，主要的类层次仍按照哺乳类和鸟类设计：

```
class Animal(object):
    pass

# 大类:
class Mammal(Animal):
    pass

class Bird(Animal):
    pass

# 各种动物:
class Dog(Mammal):
    pass

class Bat(Mammal):
    pass

class Parrot(Bird):
    pass

class Ostrich(Bird):
    pass
```

现在，我们要给动物再加上`Runnable`和`Flyable`的功能，只需要先定义好`Runnable`和`Flyable`的类：

```
class Runnable(object):
    def run(self):
        print('Running...')

class Flyable(object):
    def fly(self):
        print('Flying...')
```

对于需要`Runnable`功能的动物，就多继承一个`Runnable`，例如`Dog`：

```
class Dog(Mammal, Runnable):
    pass
```

对于需要`Flyable`功能的动物，就多继承一个`Flyable`，例如`Bat`：

```
class Bat(Mammal, Flyable):
    pass
```

通过多重继承，一个子类就可以同时获得多个父类的所有功能。

MixIn

在设计类的继承关系时，通常，主线都是单一继承下来的，例如，`Ostrich`继承自`Bird`。但是，如果需要“混入”额外的功能，通过多重继承就可以实现，比如，让`Ostrich`除了继承自`Bird`外，再同时继承`Runnable`。这种设计通常称之为MixIn。

为了更好地看出继承关系，我们把`Runnable`和`Flyable`改为`RunnableMixIn`和`FlyableMixIn`。类似的，你还可以定义出肉食动物`CarnivorousMixIn`和植食动物`HerbivoresMixIn`，让某个动物同时拥有好几个MixIn：

```
class Dog(Mammal, RunnableMixIn, CarnivorousMixIn):
    pass
```

MixIn的目的就是给一个类增加多个功能，这样，在设计类的时候，我们优先考虑通过多重继承来组合多个MixIn的功能，而不是设计多层次的复杂的继承关系。

Python自带的很多库也使用了MixIn。举个例子，Python自带了`TCPServer`和`UDPServer`这两类网络服务，而要同时服务多个用户就必须使用多进程或多线程模型，这两种模型由`ForkingMixIn`和`ThreadingMixIn`提供。通过组合，我们就可以创造出合适的服务来。

比如，编写一个多进程模式的TCP服务，定义如下：

```
class MyTCPServer(TCPServer, ForkingMixIn):
    pass
```

编写一个多线程模式的UDP服务，定义如下：

```
class MyUDPServer(UDPServer, ThreadingMixIn):
    pass
```

如果你打算搞一个更先进的协程模型，可以编写一个`CoroutineMixIn`：

```
class MyTCPServer(TCPServer, CoroutineMixIn):
    pass
```

这样一来，我们不需要复杂而庞大的继承链，只要选择组合不同的类的功能，就可以快速构造出所需的子类。

小结

- 由于Python允许使用多重继承，因此，MixIn就是一种常见的设计。
- 只允许单一继承的语言（如Java）不能使用MixIn的设计。

定制类

看到类似`__slots__`这种形如`__xxx__`的变量或者函数名就要注意，这些在Python中是有特殊用途的。

`__slots__`我们已经知道怎么用了，`__len__()`方法我们也知道是为了能让class作用于`len()`函数。

除此之外，Python的class中还有许多这样有特殊用途的函数，可以帮助我们定制类。

str

我们先定义一个`Student`类，打印一个实例：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...
>>> print(Student('Michael'))
<__main__.Student object at 0x109afb190>
```

打印出一堆`<__main__.Student object at 0x109afb190>`，不好看。

怎么才能打印得好看呢？只需要定义好`__str__()`方法，返回一个好看的字符串就可以了：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...     def __str__(self):
...         return 'Student object (name: %s)' % self.name
...
>>> print(Student('Michael'))
Student object (name: Michael)
```

这样打印出来的实例，不但好看，而且容易看出实例内部重要的数据。

但是细心的朋友会发现直接敲变量不用`print`，打印出来的实例还是不好看：

```
>>> s = Student('Michael')
>>> s
<__main__.Student object at 0x109afb310>
```

这是因为直接显示变量调用的不是`__str__()`，而是`__repr__()`，两者的区别是`__str__()`返回用户看到的字符串，而`__repr__()`返回程序开发者看到的字符串，也就是说，`__repr__()`是为调试服务的。

解决办法是再定义一个`__repr__()`。但是通常`__str__()`和`__repr__()`代码都是一样的，所以，有个偷懒的写法：

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

iter

如果一个类想被用于`for ... in`循环，类似list或tuple那样，就必须实现一个`__iter__()`方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的`__next__()`方法拿到循环的下一个值，直到遇到`StopIteration`错误时退出循环。

我们以斐波那契数列为例，写一个Fib类，可以作用于for循环：

```
class Fib(object):
    def __init__(self):
        self.a, self.b = 0, 1 # 初始化两个计数器a，b

    def __iter__(self):
        return self # 实例本身就是迭代对象，故返回自己

    def __next__(self):
        self.a, self.b = self.b, self.a + self.b # 计算下一个值
        if self.a > 100000: # 退出循环的条件
            raise StopIteration()
        return self.a # 返回下一个值
```

现在，试试把Fib实例作用于for循环：

```
>>> for n in Fib():
...     print(n)
...
1
1
2
3
5
...
46368
75025
```

getitem

Fib实例虽然能作用于for循环，看起来和list有点像，但是，把它当成list来使用还是不行，比如，取第5个元素：

```
>>> Fib()[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Fib' object does not support indexing
```

要表现得像list那样按照下标取出元素，需要实现`__getitem__()`方法：

```
class Fib(object):
    def __getitem__(self, n):
        a, b = 1, 1
        for x in range(n):
            a, b = b, a + b
        return a
```

现在，就可以按下标访问数列的任意一项了：

```
>>> f = Fib()
>>> f[0]
1
>>> f[1]
1
>>> f[2]
2
>>> f[3]
3
>>> f[10]
89
>>> f[100]
573147844013817084101
```

但是list有个神奇的切片方法：

```
>>> list(range(100))[5:10]
[5, 6, 7, 8, 9]
```

对于Fib却报错。原因是`__getitem__()`传入的参数可能是一个int，也可能是一个切片对象`slice`，所以要做判断：

```
class Fib(object):
    def __getitem__(self, n):
        if isinstance(n, int): # n是索引
            a, b = 1, 1
            for x in range(n):
                a, b = b, a + b
            return a
        if isinstance(n, slice): # n是切片
            start = n.start
            stop = n.stop
            if start is None:
                start = 0
            a, b = 1, 1
            L = []
            for x in range(stop):
                if x >= start:
                    L.append(a)
                a, b = b, a + b
            return L
```

现在试试Fib的切片：

```
>>> f = Fib()
>>> f[0:5]
[1, 1, 2, 3, 5]
>>> f[:10]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

但是没有对step参数作处理：

```
>>> f[:10:2]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

也没有对负数作处理，所以，要正确实现一个`__getitem__()`还是有很多工作要做的。

此外，如果把对象看成`dict`，`__getitem__()`的参数也可能是一个可以作key的object，例如`str`。

与之对应的是`__setitem__()`方法，把对象视作list或dict来对集合赋值。最后，还有一个`__delitem__()`方法，用于删除某个元素。

总之，通过上面的方法，我们自己定义的类表现得和Python自带的list、tuple、dict没什么区别，这完全归功于动态语言的“鸭子类型”，不需要强制继承某个接口。

getattr

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。比如定义`Student`类：

```
class Student(object):

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

调用`name`属性，没问题，但是，调用不存在的`score`属性，就有问题了：

```
>>> s = Student()
>>> print(s.name)
Michael
>>> print(s.score)
Traceback (most recent call last):
  ...
AttributeError: 'Student' object has no attribute 'score'
```

错误信息很清楚地告诉我们，没有找到`score`这个attribute。

要避免这个错误，除了可以加上一个`score`属性外，Python还有另一个机制，那就是写一个`__getattr__()`方法，动态返回一个属性。修改如下：

```
class Student(object):

    def __init__(self):
        self.name = 'Michael'

    def __getattr__(self, attr):
        if attr=='score':
            return 99
```

当调用不存在的属性时，比如`score`，Python解释器会试图调用`__getattr__(self, 'score')`来尝试获得属性，这样，我们就有机会返回`score`的值：

```
>>> s = Student()
>>> s.name
'Michael'
>>> s.score
99
```

返回函数也是完全可以的：

```
class Student(object):

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

只是调用方式要变为：

```
>>> s.age()
25
```

注意，只有在没有找到属性的情况下，才调用`__getattr__`，已有的属性，比如`name`，不会在`__getattr__`中查找。

此外，注意到任意调用如`s.abc`都会返回`None`，这是因为我们定义的`__getattr__`默认返回就是`None`。要让class只响应特定的几个属性，我们就要按照约定，抛出`AttributeError`的错误：

```
class Student(object):

    def __getattr__(self, attr):
        if attr=='age':
            return lambda: 25
        raise AttributeError('\'Student\' object has no
attribute \'%s\'' % attr)
```

这实际上可以把一个类的所有属性和方法调用全部动态化处理了，不需要任何特殊手段。

这种完全动态调用的特性有什么实际作用呢？作用就是，可以针对完全动态的情况作调用。

举个例子：

现在很多网站都搞REST API，比如新浪微博、豆瓣啥的，调用API的URL类似：

- http://api.server/user/friends
- http://api.server/user/timeline/list

如果要写SDK，给每个URL对应的API都写一个方法，那得累死，而且，API一旦改动，SDK也要改。

利用完全动态的`__getattr__`，我们可以写出一个链式调用：

```
class Chain(object):

    def __init__(self, path=''):
        self._path = path

    def __getattr__(self, path):
        return Chain('%s/%s' % (self._path, path))

    def __str__(self):
        return self._path

    __repr__ = __str__
```

试试：

```
>>> Chain().status.user.timeline.list
'/status/user/timeline/list'
```

这样，无论API怎么变，SDK都可以根据URL实现完全动态的调用，而且，不随API的增加而改变！

还有些REST API会把参数放到URL中，比如GitHub的API：

```
GET /users/:user/repos
```

调用时，需要把`:user`替换为实际用户名。如果我们能写出这样的链式调用：

```
Chain().users('michael').repos
```

就可以非常方便地调用API了。有兴趣的童鞋可以试试写出来。

call

一个对象实例可以有自己的属性和方法，当我们调用实例方法时，我们用`instance.method()`来调用。能不能直接在实例本身上调用呢？在Python中，答案是肯定的。

任何类，只需要定义一个`__call__()`方法，就可以直接对实例进行调用。请看示例：

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

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

调用方式如下：

```
>>> s = Student('Michael')
>>> s() # self参数不要传入
My name is Michael.
```

`__call__()`还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

如果你把对象看成函数，那么函数本身其实也可以在运行期动态创建出来，因为类的实例都是运行期创建出来的，这么一来，我们就模糊了对象和函数的界限。

那么，怎么判断一个变量是对象还是函数呢？其实，更多的时候，我们需要判断一个对象是否能被调用，能被调用的对象就是一个`Callable`对象，比如函数和我们上面定义的带有`__call__()`的类实例：

```
>>> callable(Student())
True
>>> callable(max)
True
>>> callable([1, 2, 3])
False
>>> callable(None)
False
>>> callable('str')
False
```

通过`callable()`函数，我们就可以判断一个对象是否是“可调用”对象。

小结

- Python的class允许定义许多定制方法，可以让我们非常方便地生成特定的类。

- 本节介绍的是最常用的几个定制方法，还有很多可定制的方法，请参考Python的官方文档。

使用枚举类

当我们需要定义常量时，一个办法是用大写变量通过整数来定义，例如月份：

```
JAN = 1
FEB = 2
MAR = 3
...
NOV = 11
DEC = 12
```

好处是简单，缺点是类型是`int`，并且仍然是变量。

更好的方法是为这样的枚举类型定义一个class类型，然后，每个常量都是class的一个唯一实例。Python提供了`Enum`类来实现这个功能：

```
from enum import Enum

Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr', 'May',
'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))
```

这样我们就获得了`Month`类型的枚举类，可以直接使用`Month.Jan`来引用一个常量，或者枚举它的所有成员：

```
for name, member in Month.__members__.items():
    print(name, '=>', member, ',', member.value)
```

`value`属性则是自动赋给成员的`int`常量，默认从`1`开始计数。

如果需要更精确地控制枚举类型，可以从`Enum`派生出自定义类：

```
from enum import Enum, unique

@unique
class Weekday(Enum):
    Sun = 0 # Sun的value被设定为0
    Mon = 1
    Tue = 2
    Wed = 3
    Thu = 4
    Fri = 5
    Sat = 6
```

`@unique`装饰器可以帮助我们检查保证没有重复值。

访问这些枚举类型可以有若干种方法：

```
>>> day1 = Weekday.Mon
>>> print(day1)
Weekday.Mon
>>> print(Weekday.Tue)
Weekday.Tue
>>> print(Weekday['Tue'])
Weekday.Tue
>>> print(Weekday.Tue.value)
2
>>> print(day1 == Weekday.Mon)
True
>>> print(day1 == Weekday.Tue)
False
>>> print(Weekday(1))
Weekday.Mon
>>> print(day1 == Weekday(1))
True
>>> Weekday(7)
Traceback (most recent call last):
  ...
ValueError: 7 is not a valid Weekday
>>> for name, member in Weekday.__members__.items():
...     print(name, '=>', member)
...
Sun => Weekday.Sun
Mon => Weekday.Mon
Tue => Weekday.Tue
Wed => Weekday.Wed
Thu => Weekday.Thu
Fri => Weekday.Fri
Sat => Weekday.Sat
```

可见，既可以用成员名称引用枚举常量，又可以直接根据value的值获得枚举常量。

练习

把`Student`的`gender`属性改造为枚举类型，可以避免使用字符串：

```
# -*- coding: utf-8 -*-
from enum import Enum, unique
class Gender(Enum):
    Male = 0
    Female = 1

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

```
# 测试:
bart = Student('Bart', Gender.Male)
if bart.gender == Gender.Male:
    print('测试通过!')
else:
    print('测试失败!')
```

小结

- `Enum`可以把一组相关常量定义在一个class中，且class不可变，而且成员可以直接比较。

使用元类

type()

动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。

比方说我们要定义一个`Hello`的class，就写一个`hello.py`模块：

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

当Python解释器载入`hello`模块时，就会依次执行该模块的所有语句，执行结果就是动态创建出一个`Hello`的class对象，测试如下：

```
>>> from hello import Hello
>>> h = Hello()
>>> h.hello()
Hello, world.
>>> print(type(Hello))
<class 'type'>
>>> print(type(h))
<class 'hello.Hello'>
```

`type()`函数可以查看一个类型或变量的类型，`Hello`是一个class，它的类型就是`type`，而`h`是一个实例，它的类型就是class `Hello`。

我们说class的定义是运行时动态创建的，而创建class的方法就是使用`type()`函数。

`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，比如，我们可以通过`type()`函数创建出`Hello`类，而无需通过`class Hello(object)...`的定义：

```
>>> def fn(self, name='world'): # 先定义函数
...     print('Hello, %s.' % name)
...
>>> Hello = type('Hello', (object,), dict(hello=fn)) # 创建Hello class
>>> h = Hello()
>>> h.hello()
Hello, world.
>>> print(type(Hello))
<class 'type'>
>>> print(type(h))
<class '__main__.Hello'>
```

要创建一个class对象，`type()`函数依次传入3个参数：

1. class的名称；
2. 继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元素写法；
3. class的方法名称与函数绑定，这里我们把函数`fn`绑定到方法名`hello`上。

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，仅仅是扫描一下class定义的语法，然后调用`type()`函数创建出class。

正常情况下，我们都用`class Xxx...`来定义类，但是，`type()`函数也允许我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用编译器，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂。

`metaclass`

除了使用`type()`动态创建类以外，要控制类的创建行为，还可以使用`metaclass`。

`metaclass`，直译为元类，简单的解释就是：

当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例。

但是如果我们想创建出类呢？那就必须根据`metaclass`创建出类，所以：先定义`metaclass`，然后创建类。

连接起来就是：先定义`metaclass`，就可以创建类，最后创建实例。

所以，`metaclass`允许你创建类或者修改类。换句话说，你可以把类看成是`metaclass`创建出来的“实例”。

`metaclass`是Python面向对象里最难理解，也是最难使用的魔术代码。正常情况下，你不会碰到需要使用`metaclass`的情况，所以，以下内容看不懂也没关系，因为基本上你不会用到。

我们先看一个简单的例子，这个`metaclass`可以给我们自定义的`MyList`增加一个`add`方法：

定义`ListMetaclass`，按照默认习惯，`metaclass`的类名总是以`Metaclass`结尾，以便清楚地表示这是一个`metaclass`：

```
# metaclass是类的模板，所以必须从`type`类型派生：
class ListMetaclass(type):
    def __new__(cls, name, bases, attrs):
        attrs['add'] = lambda self, value:
self.append(value)
        return type.__new__(cls, name, bases, attrs)
```

有了`ListMetaclass`，我们在定义类的时候还要指示使用`ListMetaclass`来定制类，传入关键字参数`metaclass`：

```
class MyList(list, metaclass=ListMetaclass):
    pass
```

当我们传入关键字参数`metaclass`时，魔术就生效了，它指示Python解释器在创建`MyList`时，要通过`ListMetaclass.__new__()`来创建，在此，我们可以修改类的定义，比如，加上新的方法，然后，返回修改后的定义。

`__new__()`方法接收到的参数依次是：

1. 当前准备创建的类的对象；
2. 类的名字；
3. 类继承的父类集合；
4. 类的方法集合。

测试一下`MyList`是否可以调用`add()`方法：

```
>>> L = MyList()
>>> L.add(1)
>> L
[1]
```

而普通的`list`没有`add()`方法：

```
>>> L2 = list()
>>> L2.add(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'add'
```

动态修改有什么意义？直接在`MyList`定义中写上`add()`方法不是更简单吗？正常情况下，确实应该直接写，通过`metaclass`修改纯属变态。

但是，总会遇到需要通过`metaclass`修改类定义的。ORM就是一个典型的例子。

ORM全称“Object Relational Mapping”，即对象-关系映射，就是把关系数据库的一行映射为一个对象，也就是一个类对应一个表，这样，写代码更简单，不用直接操作SQL语句。

要编写一个ORM框架，所有的类都只能动态定义，因为只有使用者才能根据表的结构定义出对应的类来。

让我们来尝试编写一个ORM框架。

编写底层模块的第一步，就是先把调用接口写出来。比如，使用者如果使用这个ORM框架，想定义一个`User`类来操作对应的数据库表`User`，我们期待他写出这样的代码：

```
class User(Model):
    # 定义类的属性到列的映射：
    id = IntegerField('id')
    name = StringField('username')
    email = StringField('email')
    password = StringField('password')

# 创建一个实例：
u = User(id=12345, name='Michael', email='test@orm.org',
password='my-pwd')
# 保存到数据库：
u.save()
```

其中，父类`Model`和属性类型`StringField`、`IntegerField`是由ORM框架提供的，剩下的魔术方法比如`save()`全部由`metaclass`自动完成。虽然`metaclass`的编写会比较复杂，但ORM的使用者用起来却异常简单。

现在，我们就按上面的接口来实现该ORM。

首先来定义`Field`类，它负责保存数据库表的字段名和字段类型：

```
class Field(object):

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

    def __str__(self):
        return '<%s:%s>' % (self.__class__.__name__,
self.name)
```

在`Field`的基础上，进一步定义各种类型的`Field`，比如`StringField`，`IntegerField`等等：

```
class StringField(Field):

    def __init__(self, name):
        super(StringField, self).__init__(name,
'varchar(100)')

class IntegerField(Field):

    def __init__(self, name):
        super(IntegerField, self).__init__(name, 'bigint')
```

下一步，就是编写最复杂的`ModelMetaclass`了：

```
class ModelMetaclass(type):

    def __new__(cls, name, bases, attrs):
        if name=='Model':
            return type.__new__(cls, name, bases, attrs)
        print('Found model: %s' % name)
        mappings = dict()
        for k, v in attrs.items():
            if isinstance(v, Field):
                print('Found mapping: %s ==> %s' % (k, v))
                mappings[k] = v
        for k in mappings.keys():
            attrs.pop(k)
        attrs['__mappings__'] = mappings # 保存属性和列的映射
关系
        attrs['__table__'] = name # 假设表名和类名一致
        return type.__new__(cls, name, bases, attrs)
```

以及基类`Model`：

```
class Model(dict, metaclass=ModelMetaclass):

    def __init__(self, **kw):
        super(Model, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Model' object has no
attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value
```

```
    def save(self):
        fields = []
        params = []
        args = []
        for k, v in self.__mappings__.items():
            fields.append(v.name)
            params.append('?')
            args.append(getattr(self, k, None))
        sql = 'insert into %s (%s) values (%s)' %
(self.__table__, ','.join(fields), ','.join(params))
        print('SQL: %s' % sql)
        print('ARGS: %s' % str(args))
```

当用户定义一个`class User(Model)`时，Python解释器首先在当前类`User`的定义中查找`metaclass`，如果没有找到，就继续在父类`Model`中查找`metaclass`，找到了，就使用`Model`中定义的`metaclass`的`ModelMetaclass`来创建`User`类，也就是说，`metaclass`可以隐式地继承到子类，但子类自己却感觉不到。

在`ModelMetaclass`中，一共做了几件事情：

1. 排除掉对`Model`类的修改；
2. 在当前类（比如`User`）中查找定义的类的所有属性，如果找到一个Field属性，就把它保存到一个`__mappings__`的dict中，同时从类属性中删除该Field属性，否则，容易造成运行时错误（实例的属性会遮盖类的同名属性）；
3. 把表名保存到`__table__`中，这里简化为表名默认为类名。

在`Model`类中，就可以定义各种操作数据库的方法，比如`save()`，`delete()`，`find()`，`update`等等。

我们实现了`save()`方法，把一个实例保存到数据库中。因为有表名，属性到字段的映射和属性值的集合，就可以构造出`INSERT`语句。

编写代码试试：

```
u = User(id=12345, name='Michael', email='test@orm.org',
password='my-pwd')
u.save()
```

输出如下：

```
Found model: User
Found mapping: email ==> <StringField:email>
Found mapping: password ==> <StringField:password>
Found mapping: id ==> <IntegerField:uid>
Found mapping: name ==> <StringField:username>
SQL: insert into User (password,email,username,id) values
(?,?,?,?)
ARGS: ['my-pwd', 'test@orm.org', 'Michael', 12345]
```

可以看到，`save()`方法已经打印出了可执行的SQL语句，以及参数列表，只需要真正连接到数据库，执行该SQL语句，就可以完成真正的功能。

不到100行代码，我们就通过`metaclass`实现了一个精简的ORM框架，是不是非常简单？

小结

- `metaclass`是Python中非常具有魔术性的对象，它可以改变类创建时的行为。这种强大的功能使用起来务必小心。