多重继承

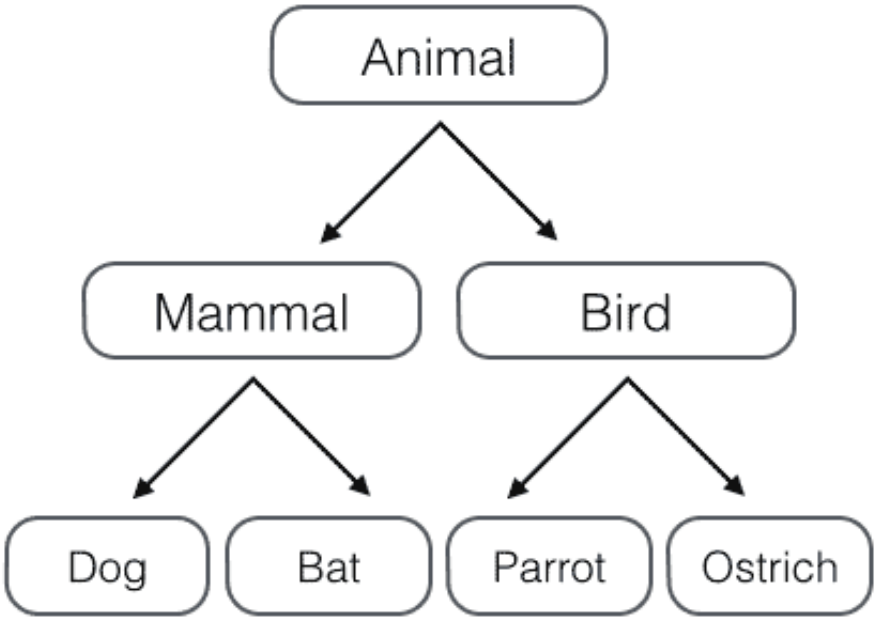
1006次阅读

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。

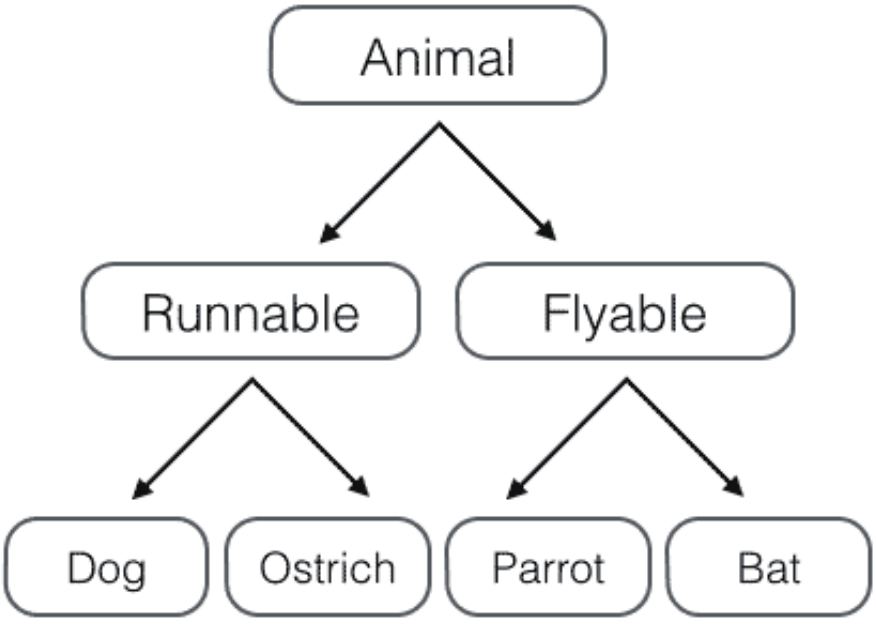
回忆一下Animal类层次的设计，假设我们要实现以下4种动物：

- 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：



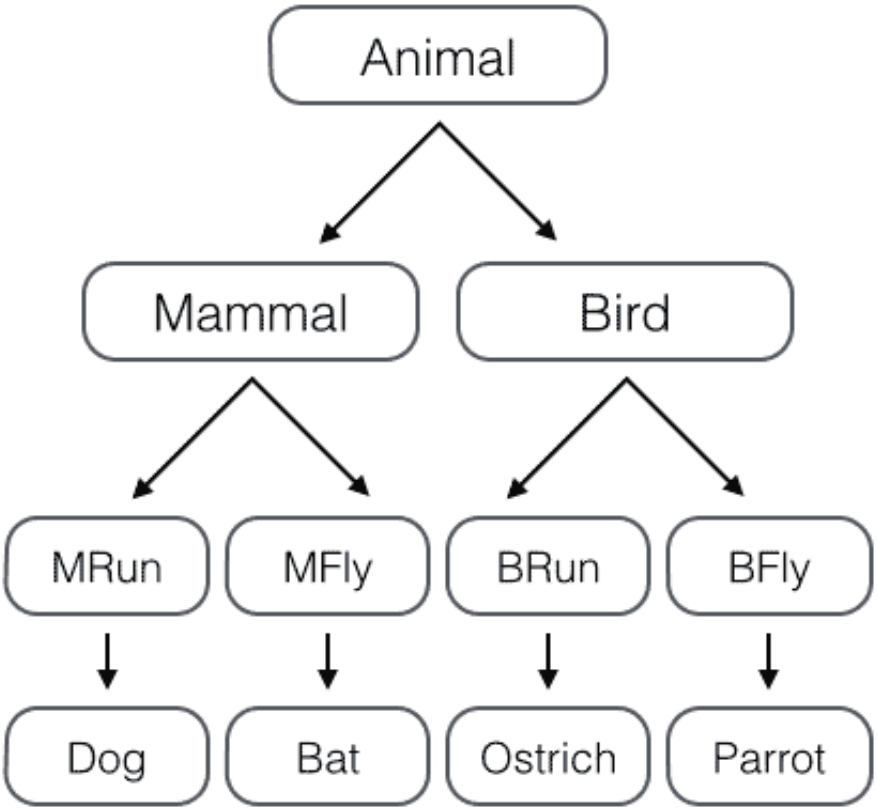
但是如果按照“能跑”和“能飞”来归类，我们就应该设计出这样的类的层次：



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

1268次阅读

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：

```
>>> 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):
            a, b = 1, 1
            for x in range(n):
                a, b = b, a + b
            return a
        if isinstance(n, slice):
            start = n.start
            stop = n.stop
            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
```

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？类似instance()？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()
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('string')
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。

使用元类

1308次阅读

---

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))
<type '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))
<type '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是创建类，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)

class MyList(list):
    __metaclass__ = ListMetaclass # 指示使用ListMetaclass来定制类
```

当我们写下__metaclass__ = ListMetaclass语句时，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：

```
>>> l = list()
>>> l.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)
        mappings = dict()
        for k, v in attrs.iteritems():
            if isinstance(v, Field):
                print('Found mapping: %s==>%s' % (k, v))
                mappings[k] = v
        for k in mappings.iterkeys():
            attrs.pop(k)
        attrs['__table__'] = name # 假设表名和类名一致
        attrs['__mappings__'] = mappings # 保存属性和列的映射关系
        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__.iteritems():
            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；
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,uid) 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，完整的代码从这里下载：

https://github.com/michaelliao/learn-python/blob/master/metaclass/simple_orm.py

最后解释一下类属性和实例属性。直接在class中定义的是类属性：

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

实例属性必须通过实例来绑定，比如self.name = 'xxx'。来测试一下：

```
>>> # 创建实例s：
>>> s = Student()
>>> # 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(s.name)
Student
>>> # 这和调用Student.name是一样的：
>>> print(Student.name)
Student
>>> # 给实例绑定name属性：
>>> s.name = 'Michael'
>>> # 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：
>>> print(s.name)
Michael
>>> # 但是类属性并未消失，用Student.name仍然可以访问：
>>> print(Student.name)
Student
>>> # 如果删除实例的name属性：
>>> del 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：
>>> print(s.name)
Student
```

因此，在编写程序的时候，千万不要把实例属性和类属性使用相同的名字。

在我们编写的ORM中，ModelMetaclass会删除掉User类的所有类属性，目的就是避免造成混淆。

错误、调试和测试

1088次阅读

在程序运行过程中，总会遇到各种各样的错误。

有的错误是程序编写有问题造成的，比如本来应该输出整数结果输出了字符串，这种错误我们通常称之为bug，bug是必须修复的。

有的错误是用户输入造成的，比如让用户输入email地址，结果得到一个空字符串，这种错误可以通过检查用户输入来做相应的处理。

还有一类错误是完全无法在程序运行过程中预测的，比如写入文件的时候，磁盘满了，写不进去了，或者从网络抓取数据，网络突然断掉了。这类错误也称为异常，在程序中通常是必须处理的，否则，程序会因为各种问题终止并退出。

Python内置了一套异常处理机制，来帮助我们进行错误处理。

此外，我们也需要跟踪程序的执行，查看变量的值是否正确，这个过程称为调试。Python的pdb可以让我们以单步方式执行代码。

最后，编写测试也很重要。有了良好的测试，就可以在程序修改后反复运行，确保程序输出符合我们编写的测试。

错误处理

1164次阅读

在程序运行的过程中，如果发生了错误，可以事先约定返回一个错误代码，这样，就可以知道是否有错，以及出错的原因。在操作系统提供的调用中，返回错误码非常常见。比如打开文件的函数open()，成功时返回文件描述符（就是一个整数），出错时返回-1。

用错误码来表示是否出错十分不便，因为函数本身应该返回的正常结果和错误码混在一起，造成调用者必须用大量的代码来判断是否出错：

```
def foo():
    r = some_function()
    if r==(-1):
        return (-1)
    # do something
    return r

def bar():
    r = foo()
    if r==(-1):
        print 'Error'
    else:
        pass
```

一旦出错，还要一级一级上报，直到某个函数可以处理该错误（比如，给用户输出一个错误信息）。

所以高级语言通常都内置了一套try...except...finally...的错误处理机制，Python也不例外。

try

让我们用一个例子来看看try的机制：

```
try:
    print 'try...'
    r = 10 / 0
    print 'result:', r
except ZeroDivisionError, e:
    print 'except:', e
finally:
    print 'finally...'
print 'END'
```

当我们认为某些代码可能会出错时，就可以用try来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即except语句块，执行完except后，如果有finally语句块，则执行finally语句块，至此，执行完毕。

上面的代码在计算10 / 0时会产生一个除法运算错误：

```
try...
except: integer division or modulo by zero
finally...
END
```

从输出可以看到，当错误发生时，后续语句print 'result:', r不会被执行，except由于捕获到ZeroDivisionError，因此被执行。最后，finally语句被执行。然后，程序继续按照流程往下走。

如果把除数0改成2，则执行结果如下：

```
try...
result: 5
finally...
END
```

由于没有错误发生，所以except语句块不会被执行，但是finally如果有，则一定会被执行（可以没有finally语句）。

你还可以猜测，错误应该有很多种类，如果发生了不同类型的错误，应该由不同的except语句块处理。没错，可以有多个except来捕获不同类型的错误：

```
try:
    print 'try...'
    r = 10 / int('a')
    print 'result:', r
except ValueError, e:
    print 'ValueError:', e
except ZeroDivisionError, e:
    print 'ZeroDivisionError:', e
finally:
    print 'finally...'
print 'END'
```

int()函数可能会抛出ValueError，所以我们用一个except捕获ValueError，用另一个except捕获ZeroDivisionError。

此外，如果没有错误发生，可以在except语句块后面加一个else，当没有错误发生时，会自动执行else语句：

```
try:
    print 'try...'
    r = 10 / int('a')
    print 'result:', r
except ValueError, e:
    print 'ValueError:', e
except ZeroDivisionError, e:
    print 'ZeroDivisionError:', e
else:
    print 'no error!'
finally:
    print 'finally...'
print 'END'
```

Python的错误其实也是class，所有的错误类型都继承自BaseException，所以在使用except时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。比如：

```
try:
    foo()
except StandardError, e:
    print 'StandardError'
except ValueError, e:
    print 'ValueError'
```

第二个except永远也捕获不到ValueError，因为ValueError是StandardError的子类，如果有，也被第一个except给捕获了。

Python所有的错误都是从BaseException类派生的，常见的错误类型和继承关系看这里：

https://docs.python.org/2/library/exceptions.html#exception-hierarchy

使用try...except捕获错误还有一个巨大的好处，就是可以跨越多层调用，比如函数main()调用foo()，foo()调用bar()，结果bar()出错了，这时，只要main()捕获到了，就可以处理：

```
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except StandardError, e:
        print 'Error!'
    finally:
        print 'finally...'
```

也就是说，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写try...except...finally的麻烦。

调用堆栈

如果错误没有被捕获，它就会一直往上抛，最后被Python解释器捕获，打印一个错误信息，然后程序退出。来看看err.py：

```
# err.py:
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    bar('0')

main()
```

执行，结果如下：

```
$ python err.py
Traceback (most recent call last):
  File "err.py", line 11, in <module>
    main()
  File "err.py", line 9, in main
    bar('0')
  File "err.py", line 6, in bar
    return foo(s) * 2
  File "err.py", line 3, in foo
    return 10 / int(s)
ZeroDivisionError: integer division or modulo by zero
```

出错并不可怕，可怕的是不知道哪里出错了。解读错误信息是定位错误的关键。我们从上往下可以看到整个错误的调用函数链：

错误信息第1行：

```
Traceback (most recent call last):
```

告诉我们这是错误的跟踪信息。

第2行：

```
  File "err.py", line 11, in <module>
    main()
```

调用main()出错了，在代码文件err.py的第11行代码，但原因是第9行：

```
  File "err.py", line 9, in main
    bar('0')
```

调用bar('0')出错了，在代码文件err.py的第9行代码，但原因是第6行：

```
  File "err.py", line 6, in bar
    return foo(s) * 2
```

原因是return foo(s) * 2这个语句出错了，但这还不是最终原因，继续往下看：

```
  File "err.py", line 3, in foo
    return 10 / int(s)
```

原因是return 10 / int(s)这个语句出错了，这是错误产生的源头，因为下面打印了：

```
ZeroDivisionError: integer division or modulo by zero
```

根据错误类型ZeroDivisionError，我们判断，int(s)本身并没有出错，但是int(s)返回0，在计算10 / 0时出错，至此，找到错误源头。

记录错误

如果不捕获错误，自然可以让Python解释器来打印出错误堆栈，但程序也被结束了。既然我们能捕获错误，就可以把错误堆栈打印出来，然后分析错误原因，同时，让程序继续执行下去。

Python内置的logging模块可以非常容易地记录错误信息：

```
# err.py
import logging

def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except StandardError, e:
        logging.exception(e)

main()
print 'END'
```

同样是出错，但程序打印完错误信息后会继续执行，并正常退出：

```
$ python err.py
ERROR:root:integer division or modulo by zero
Traceback (most recent call last):
  File "err.py", line 12, in main
    bar('0')
  File "err.py", line 8, in bar
    return foo(s) * 2
  File "err.py", line 5, in foo
    return 10 / int(s)
```

```
ZeroDivisionError: integer division or modulo by zero
END
```

通过配置，logging还可以把错误记录到日志文件里，方便事后排查。

抛出错误

因为错误是class，捕获一个错误就是捕获到该class的一个实例。因此，错误并不是凭空产生的，而是有意创建并抛出的。Python的内置函数会抛出很多类型的错误，我们自己编写的函数也可以抛出错误。

如果要抛出错误，首先根据需要，可以定义一个错误的class，选择好继承关系，然后，用raise语句抛出一个错误的实例：

```
# err.py
class FooError(StandardError):
    pass

def foo(s):
    n = int(s)
    if n==0:
        raise FooError('invalid value: %s' % s)
    return 10 / n
```

执行，可以最后跟踪到我们自己定义的错误：

```
$ python err.py
Traceback (most recent call last):
  ...
__main__.FooError: invalid value: 0
```

只有在必要的时候才定义我们自己的错误类型。如果可以选择Python已有的内置的错误类型（比如ValueError，TypeError），尽量使用Python内置的错误类型。

最后，我们来看另一种错误处理的方式：

```
# err.py
def foo(s):
    n = int(s)
    return 10 / n

def bar(s):
    try:
        return foo(s) * 2
    except StandardError, e:
        print 'Error!'
        raise

def main():
    bar('0')

main()
```

在bar()函数中，我们明明已经捕获了错误，但是，打印一个Error!后，又把错误通过raise语句抛出去了，这不有病么？

其实这种错误处理方式不但没病，而且相当常见。捕获错误目的只是记录一下，便于后续追踪。但是，由于当前函数不知道应该怎么处理该错误，所以，最恰当的方式是继续往上抛，让顶层调用者去处理。

raise语句如果不带参数，就会把当前错误原样抛出。此外，在except中raise一个Error，还可以把一种类型的错误转化成另一种类型：

```
try:
    10 / 0
except ZeroDivisionError:
    raise ValueError('input error!')
```

只要是合理的转换逻辑就可以，但是，决不应该把一个IOError转换成毫不相干的ValueError。

小结

Python内置的try...except...finally用来处理错误十分方便。出错时，会分析错误信息并定位错误发生的代码位置才是最关键的。

程序也可以主动抛出错误，让调用者来处理相应的错误。但是，应该在文档中写清楚可能会抛出哪些错误，以及错误产生的原因。

调试

860次阅读

程序能一次写完并正常运行的概率很小，基本不超过1%。总会有各种各样的bug需要修正。有的bug很简单，看看错误信息就知道，有的bug很复杂，我们需要知道出错时，哪些变量的值是正确的，哪些变量的值是错误的，因此，需要一整套调试程序的手段来修复bug。

第一种方法简单直接粗暴有效，就是用print把可能有问题的变量打印出来看看：

```
# err.py
def foo(s):
    n = int(s)
    print '>>> n = %d' % n
    return 10 / n

def main():
    foo('0')

main()
```

执行后在输出中查找打印的变量值：

```
$ python err.py
>>> n = 0
Traceback (most recent call last):
  ...
ZeroDivisionError: integer division or modulo by zero
```

用print最大的坏处是将来还得删掉它，想想程序里到处都是print，运行结果也会包含很多垃圾信息。所以，我们又有第二种方法。

断言

凡是用print来辅助查看的地方，都可以用断言（assert）来替代：

```
# err.py
def foo(s):
    n = int(s)
    assert n != 0, 'n is zero!'
    return 10 / n

def main():
    foo('0')
```

assert的意思是，表达式n != 0应该是True，否则，后面的代码就会出错。

如果断言失败，assert语句本身就会抛出AssertionError：

```
$ python err.py
Traceback (most recent call last):
  ...
AssertionError: n is zero!
```

程序中如果到处充斥着assert，和print相比也好不到哪去。不过，启动Python解释器时可以用-O参数来关闭assert：

```
$ python -O err.py
Traceback (most recent call last):
```

```
  ...
ZeroDivisionError: integer division or modulo by zero
```

关闭后，你可以把所有的assert语句当成pass来看。

logging

把print替换为logging是第3种方式，和assert比，logging不会抛出错误，而且可以输出到文件：

```
# err.py
import logging

s = '0'
n = int(s)
logging.info('n = %d' % n)
print 10 / n
```

logging.info()就可以输出一段文本。运行，发现除了ZeroDivisionError，没有任何信息。怎么回事？

别急，在import logging之后添加一行配置再试试：

```
import logging
logging.basicConfig(level=logging.INFO)
```

看到输出了：

```
$ python err.py
INFO:root:n = 0
Traceback (most recent call last):
  File "err.py", line 8, in <module>
    print 10 / n
ZeroDivisionError: integer division or modulo by zero
```

这就是logging的好处，它允许你指定记录信息的级别，有debug，info，warning，error等几个级别，当我们指定level=INFO时，logging.debug就不起作用了。同理，指定level=WARNING后，debug和info就不起作用了。这样一来，你可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。

logging的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如console和文件。

pdb

第4种方式是启动Python的调试器pdb，让程序以单步方式运行，可以随时查看运行状态。我们先准备好程序：

```
# err.py
s = '0'
n = int(s)
print 10 / n
```

然后启动：

```
$ python -m pdb err.py
> /Users/michael/Github/sicp/err.py(2)<module>()
-> s = '0'
```

以参数-m pdb启动后，pdb定位到下一步要执行的代码-> s = '0'。输入命令l来查看代码：

```
(Pdb) l
  1     # err.py
  2  -> s = '0'
  3     n = int(s)
  4     print 10 / n
[EOF]
```

输入命令n可以单步执行代码：

```
(Pdb) n
> /Users/michael/Github/sicp/err.py(3)<module>()
-> n = int(s)
(Pdb) n
> /Users/michael/Github/sicp/err.py(4)<module>()
-> print 10 / n
```

任何时候都可以输入命令p 变量名来查看变量：

```
(Pdb) p s
'0'
(Pdb) p n
0
```

输入命令q结束调试，退出程序：

```
(Pdb) n
ZeroDivisionError: 'integer division or modulo by zero'
> /Users/michael/Github/sicp/err.py(4)<module>()
-> print 10 / n
(Pdb) q
```

这种通过pdb在命令行调试的方法理论上是万能的，但实在是太麻烦了，如果有一千行代码，要运行到第999行得敲多少命令啊。还好，我们还有另一种调试方法。

pdb.set_trace()

这个方法也是用pdb，但是不需要单步执行，我们只需要import pdb，然后，在可能出错的地方放一个pdb.set_trace()，就可以设置一个断点：

```
# err.py
import pdb

s = '0'
n = int(s)
pdb.set_trace() # 运行到这里会自动暂停
print 10 / n
```

运行代码，程序会自动在pdb.set_trace()暂停并进入pdb调试环境，可以用命令p查看变量，或者用命令c继续运行：

```
$ python err.py
> /Users/michael/Github/sicp/err.py(7)<module>()
-> print 10 / n
(Pdb) p n
0
(Pdb) c
Traceback (most recent call last):
  File "err.py", line 7, in <module>
    print 10 / n
ZeroDivisionError: integer division or modulo by zero
```

这个方式比直接启动pdb单步调试效率要高很多，但也高不到哪去。

IDE

如果要比较爽地设置断点、单步执行，就需要一个支持调试功能的IDE。目前比较好的Python IDE有PyCharm：

http://www.jetbrains.com/pycharm/

另外，Eclipse加上pydev插件也可以调试Python程序。

小结

写程序最痛苦的事情莫过于调试，程序往往会以你意想不到的流程来运行，你期待执行的语句其实根本没有执行，这时候，就需要调试了。

虽然用IDE调试起来比较方便，但是最后你会发现，logging才是终极武器。

单元测试

564次阅读

如果你听说过“测试驱动开发”（TDD：Test-Driven Development），单元测试就不陌生。

单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

比如对函数abs()，我们可以编写出以下几个测试用例：

1. 输入正数，比如1、1.2、0.99，期待返回值与输入相同；
2. 输入负数，比如-1、-1.2、-0.99，期待返回值与输入相反；
3. 输入0，期待返回0；
4. 输入非数值类型，比如None、[]、{}，期待抛出TypeError。

把上面的测试用例放到一个测试模块里，就是一个完整的单元测试。

如果单元测试通过，说明我们测试的这个函数能够正常工作。如果单元测试不通过，要么函数有bug，要么测试条件输入不正确，总之，需要修复使单元测试能够通过。

单元测试通过后有什么意义呢？如果我们对abs()函数代码做了修改，只需要再跑一遍单元测试，如果通过，说明我们的修改不会对abs()函数原有的行为造成影响，如果测试不通过，说明我们的修改与原有行为不一致，要么修改代码，要么修改测试。

这种以测试为驱动的开发模式最大的好处就是确保一个程序模块的行为符合我们设计的测试用例。在将来修改的时候，可以极大程度地保证该模块行为仍然是正确的。

我们来编写一个Dict类，这个类的行为和dict一致，但是可以通过属性来访问，用起来就像下面这样：

```
>>> d = Dict(a=1, b=2)
>>> d['a']
1
>>> d.a
1
```

mydict.py代码如下：

```
class Dict(dict):

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

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

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

为了编写单元测试，我们需要引入Python自带的unittest模块，编写mydict_test.py如下：

```
import unittest
```

```
from mydict import Dict

class TestDict(unittest.TestCase):

    def test_init(self):
        d = Dict(a=1, b='test')
        self.assertEquals(d.a, 1)
        self.assertEquals(d.b, 'test')
        self.assertTrue(isinstance(d, dict))

    def test_key(self):
        d = Dict()
        d['key'] = 'value'
        self.assertEquals(d.key, 'value')

    def test_attr(self):
        d = Dict()
        d.key = 'value'
        self.assertTrue('key' in d)
        self.assertEquals(d['key'], 'value')

    def test_keyerror(self):
        d = Dict()
        with self.assertRaises(KeyError):
            value = d['empty']

    def test_attrerror(self):
        d = Dict()
        with self.assertRaises(AttributeError):
            value = d.empty
```

编写单元测试时，我们需要编写一个测试类，从unittest.TestCase继承。

以test开头的方法就是测试方法，不以test开头的方法不被认为是测试方法，测试的时候不会被执行。

对每一类测试都需要编写一个test_xxx()方法。由于unittest.TestCase提供了很多内置的条件判断，我们只需要调用这些方法就可以断言输出是否是我们所期望的。最常用的断言就是assertEquals()：

```
self.assertEquals(abs(-1), 1) # 断言函数返回的结果与1相等
```

另一种重要的断言就是期待抛出指定类型的Error，比如通过d['empty']访问不存在的key时，断言会抛出KeyError：

```
with self.assertRaises(KeyError):
    value = d['empty']
```

而通过d.empty访问不存在的key时，我们期待抛出AttributeError：

```
with self.assertRaises(AttributeError):
    value = d.empty
```

运行单元测试

一旦编写好单元测试，我们就可以运行单元测试。最简单的运行方式是在mydict_test.py的最后加上两行代码：

```
if __name__ == '__main__':
    unittest.main()
```

这样就可以把mydict_test.py当做正常的python脚本运行：

```
$ python mydict_test.py
```

另一种更常见的方法是在命令行通过参数-m unittest直接运行单元测试：

```
$ python -m unittest mydict_test
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

这是推荐的做法，因为这样可以一次批量运行很多单元测试，并且，有很多工具可以自动来运行这些单元测试。

setUp与tearDown

可以在单元测试中编写两个特殊的setUp()和tearDown()方法。这两个方法会分别在每调用一个测试方法的前后分别被执行。

setUp()和tearDown()方法有什么用呢？设想你的测试需要启动一个数据库，这时，就可以在setUp()方法中连接数据库，在tearDown()方法中关闭数据库，这样，不必在每个测试方法中重复相同的代码：

```
class TestDict(unittest.TestCase):

    def setUp(self):
        print 'setUp...'

    def tearDown(self):
        print 'tearDown...'
```

可以再次运行测试看看每个测试方法调用前后是否会打印出setUp...和tearDown...。

小结

单元测试可以有效地测试某个程序模块的行为，是未来重构代码的信心保证。

单元测试的测试用例要覆盖常用的输入组合、边界条件和异常。

单元测试代码要非常简单，如果测试代码太复杂，那么测试代码本身就可能有bug。

单元测试通过了并不意味着程序就没有bug了，但是不通过程序肯定有bug。

文档测试

403次阅读

如果你经常阅读Python的官方文档，可以看到很多文档都有示例代码。比如re模块就带了很多示例代码：

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

可以把这些示例代码在Python的交互式环境下输入并执行，结果与文档中的示例代码显示的一致。

这些代码与其他说明可以写在注释中，然后，由一些工具来自动生成文档。既然这些代码本身就可以粘贴出来直接运行，那么，可不可以自动执行写在注释中的这些代码呢？

答案是肯定的。

当我们编写注释时，如果写上这样的注释：

```
def abs(n):
    '''
    Function to get absolute value of number.

    Example:

    >>> abs(1)
    1
    >>> abs(-1)
    1
    >>> abs(0)
    0
    '''
    return n if n >= 0 else (-n)
```

无疑更明确地告诉函数的调用者该函数的期望输入和输出。

并且，Python内置的“文档测试”（doctest）模块可以直接提取注释中的代码并执行测试。

doctest严格按照Python交互式命令行的输入和输出来判断测试结果是否正确。只有测试异常的时候，可以用...表示中间一大段烦人的输出。

让我们用doctest来测试上次编写的Dict类：

```
class Dict(dict):
    '''
    Simple dict but also support access as x.y style.

    >>> d1 = Dict()
    >>> d1['x'] = 100
    >>> d1.x
    100
    >>> d1.y = 200
    >>> d1['y']
    200
    >>> d2 = Dict(a=1, b=2, c='3')
    >>> d2.c
```

```
    '3'
    >>> d2['empty']
    Traceback (most recent call last):
        ...
    KeyError: 'empty'
    >>> d2.empty
    Traceback (most recent call last):
        ...
    AttributeError: 'Dict' object has no attribute 'empty'
    '''
    def __init__(self, **kw):
        super(Dict, self).__init__(**kw)

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

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

if __name__=='__main__':
    import doctest
    doctest.testmod()
```

运行python mydict.py：

```
$ python mydict.py
```

什么输出也没有。这说明我们编写的doctest运行都是正确的。如果程序有问题，比如把__getattr__()方法注释掉，再运行就会报错：

```
$ python mydict.py
**********************************************************************
File "mydict.py", line 7, in __main__.Dict
Failed example:
    d1.x
Exception raised:
    Traceback (most recent call last):
      ...
    AttributeError: 'Dict' object has no attribute 'x'
**********************************************************************
File "mydict.py", line 13, in __main__.Dict
Failed example:
    d2.c
Exception raised:
    Traceback (most recent call last):
      ...
    AttributeError: 'Dict' object has no attribute 'c'
**********************************************************************
```

注意到最后两行代码。当模块正常导入时，doctest不会被执行。只有在命令行运行时，才执行doctest。所以，不必担心doctest会在非测试环境下执行。

小结

doctest非常有用，不但可以用来测试，还可以直接作为示例代码。通过某些文档生成工具，就可以自动把包含doctest的注释提取出来。用户看文档的时候，同时也看到了doctest。

IO编程

1077次阅读

IO在计算机中指Input/Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由CPU这个超快的计算核心来执行，涉及到数据交换的地方，通常是磁盘、网络等，就需要IO接口。

比如你打开浏览器，访问新浪首页，浏览器这个程序就需要通过网络IO获取新浪的网页。浏览器首先会发送数据给新浪服务器，告诉它我想要首页的HTML，这个动作是往外发数据，叫Output，随后新浪服务器把网页发过来，这个动作是从外面接收数据，叫Input。所以，通常，程序完成IO操作会有Input和Output两个数据流。当然也有只用一个的情况，比如，从磁盘读取文件到内存，就只有Input操作，反过来，把数据写到磁盘文件里，就只是一个Output操作。

IO编程中，Stream（流）是一个很重要的概念，可以把流想象成一个水管，数据就是水管里的水，但是只能单向流动。Input Stream就是数据从外面（磁盘、网络）流进内存，Output Stream就是数据从内存流到外面去。对于浏览网页来说，浏览器和新浪服务器之间至少需要建立两根水管，才可以既能发数据，又能收数据。

由于CPU和内存的速度远远高于外设的速度，所以，在IO编程中，就存在速度严重不匹配的问题。举个例子来说，比如要把100M的数据写入磁盘，CPU输出100M的数据只需要0.01秒，可是磁盘要接收这100M数据可能需要10秒，怎么办呢？有两种办法：

第一种是CPU等着，也就是程序暂停执行后续代码，等100M的数据在10秒后写入磁盘，再接着往下执行，这种模式称为同步IO；

另一种方法是CPU不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为异步IO。

同步和异步的区别就在于是否等待IO执行的结果。好比你去麦当劳点餐，你说“来个汉堡”，服务员告诉你，对不起，汉堡要现做，需要等5分钟，于是你站在收银台前面等了5分钟，拿到汉堡再去逛商场，这是同步IO。

你说“来个汉堡”，服务员告诉你，汉堡需要等5分钟，你可以先去逛商场，等做好了，我们再通知你，这样你可以立刻去干别的事情（逛商场），这是异步IO。

很明显，使用异步IO来编写程序性能会远远高于同步IO，但是异步IO的缺点是编程模型复杂。想想看，你得知道什么时候通知你“汉堡做好了”，而通知你的方法也各不相同。如果是服务员跑过来找到你，这是回调模式，如果服务员发短信通知你，你就得不停地检查手机，这是轮询模式。总之，异步IO的复杂度远远高于同步IO。

操作IO的能力都是由操作系统提供的，每一种编程语言都会把操作系统提供的低级C接口封装起来方便使用，Python也不例外。我们后面会详细讨论Python的IO编程接口。

注意，本章的IO编程都是同步模式，异步IO由于复杂度太高，后续涉及到服务器端程序开发时我们再讨论。

文件读写

1714次阅读

读写文件是最常见的IO操作。Python内置了读写文件的函数，用法和C是兼容的。

读写文件前，我们先必须了解一下，在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）。

读文件

要以读文件的模式打开一个文件对象，使用Python内置的open()函数，传入文件名和标示符：

```
>>> f = open('/Users/michael/test.txt', 'r')
```

标示符'r'表示读，这样，我们就成功地打开了一个文件。

如果文件不存在，open()函数就会抛出一个IOError的错误，并且给出错误码和详细的信息告诉你文件不存在：

```
>>> f=open('/Users/michael/notfound.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: '/Users/michael/notfound.txt'
```

如果文件打开成功，接下来，调用read()方法可以一次读取文件的全部内容，Python把内容读到内存，用一个str对象表示：

```
>>> f.read()
'Hello, world!'
```

最后一步是调用close()方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

```
>>> f.close()
```

由于文件读写时都有可能产生IOError，一旦出错，后面的f.close()就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用try ... finally来实现：

```
try:
    f = open('/path/to/file', 'r')
    print f.read()
finally:
    if f:
        f.close()
```

但是每次都这么写实在太繁琐，所以，Python引入了with语句来自动帮我们调用close()方法：

```
with open('/path/to/file', 'r') as f:
    print f.read()
```

这和前面的try ... finally是一样的，但是代码更佳简洁，并且不必调用f.close()方法。

调用read()会一次性读取文件的全部内容，如果文件有10G，内存就爆了，所以，要保险起见，可以反复调用read(size)方法，每次最多读取size个字节的内容。另外，调用readline()可以每

次读取一行内容，调用readlines()一次读取所有内容并按行返回list。因此，要根据需要决定怎么调用。

如果文件很小，read()一次性读取最方便；如果不能确定文件大小，反复调用read(size)比较保险；如果是配置文件，调用readlines()最方便：

```
for line in f.readlines():
    print(line.strip()) # 把末尾的'\n'删掉
```

file-like Object

像open()函数返回的这种有个read()方法的对象，在Python中统称为file-like Object。除了file外，还可以是内存的字节流，网络流，自定义流等等。file-like Object不要求从特定类继承，只要写个read()方法就行。

StringIO就是在内存中创建的file-like Object，常用作临时缓冲。

二进制文件

前面讲的默认都是读取文本文件，并且是ASCII编码的文本文件。要读取二进制文件，比如图片、视频等等，用'rb'模式打开文件即可：

```
>>> f = open('/Users/michael/test.jpg', 'rb')
>>> f.read()
'\xff\xd8\xff\xe1\x00\x18Exif\x00\x00...' # 十六进制表示的字节
```

字符编码

要读取非ASCII编码的文本文件，就必须以二进制模式打开，再解码。比如GBK编码的文件：

```
>>> f = open('/Users/michael/gbk.txt', 'rb')
>>> u = f.read().decode('gbk')
>>> u
u'\u6d4b\u8bd5'
>>> print u
测试
```

如果每次都这么手动转换编码嫌麻烦（写程序怕麻烦是好事，不怕麻烦就会写出又长又难懂又没法维护的代码），Python还提供了一个codecs模块帮我们在读文件时自动转换编码，直接读出unicode：

```
import codecs
with codecs.open('/Users/michael/gbk.txt', 'r', 'gbk') as f:
    f.read() # u'\u6d4b\u8bd5'
```

写文件

写文件和读文件是一样的，唯一区别是调用open()函数时，传入标识符'w'或者'wb'表示写文本文件或写二进制文件：

```
>>> f = open('/Users/michael/test.txt', 'w')
>>> f.write('Hello, world!')
>>> f.close()
```

你可以反复调用write()来写入文件，但是务必要调用f.close()来关闭文件。当我们写文件时，操作系统往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用close()方法时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用close()的后

果是数据可能只写了一部分到磁盘，剩下的丢失了。所以，还是用with语句来得保险：

```
with open('/Users/michael/test.txt', 'w') as f:
    f.write('Hello, world!')
```

要写入特定编码的文本文件，请效仿codecs的示例，写入unicode，由codecs自动转换成指定编码。

小结

在Python中，文件读写是通过open()函数打开的文件对象完成的。使用with语句操作文件IO是个好习惯。
