Python教程

10. 面向对象编程

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。

1. 类的定义：

```
class 类名(继承类)
继承类默认为object
```

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

2. 创建类的实例：

```
类名+()
```

```
stu = Student()
```

```
可以自由给实例绑定属性
```

```
stu.name = 'ljx'
print(stu.name)
```

```
ljx
```

```
    stu.name = 'ljx'
    print(stu.name)
```

```
ljx
```

1. 由于类起到模板的作用，所以在创建实例时就要绑定类必要的属性。
2. 绑定的方式是类自带的__init__方法。

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

```
    stu.name = 'ljx'
    print(stu.name)
```

```
ljx
```

1. __init__ 方法的第一个参数self：表示创建实例本身。因此将属性绑定在self上就是绑定在实例本身
2. 有了__init__方法，就不能传空的参数，必须传入与__init__方法匹配的参数，但self不需要传，python解析器自己会自动把变量实例传进去

```
stu2 = Student("cara",100)
print(stu2.name)
print(stu2.score)
```

```
cara
100
```

与普通函数不同，类内部定义的函数第一个参数需要是self,但调用函数是不需要传入self

数据封装

面向对象编程一个重要特点就是数据封装。
在类的内部定义访问类的属性数据的函数就是类的方法。在类的内部定义访问数据的方法就把数据封装起来了。

```
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))
```

```
stu3 = Student('a',200)
stu3.print_score()
```

```
a:200
```

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'
```

```
  File "<ipython-input-10-c3e2e6e2b5a1>", line 7
    Traceback (most recent call last):
                          ^
SyntaxError: invalid syntax
```

11. 面向对象高级编程

11.1 使用slot

前一节说过，python作为动态语言。可以自由的给实例绑定实例和方法。

```
class Animal(object):
    pass

# 动态添加属性
dog = Animal()
dog.name = "money"
print(dog.name)


def set_food(self,amount):
    print('amount:',amount)
```

```
# 给单个实例动态绑定方法
from types import MethodType
dog.set_food = MethodType(set_food,dog)
dog.set_food(50)
```

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'
```

可以通过给类绑定方法，这样，每个实例也绑定了方法

```
Animal.set_food = set_food

cat = Animal()
cat.set_food(20)
```

使用slots

限制class能添加的属性

```
class People(object):
    __slots__ = ('name','age') #使用tuple定义允许添加的属
性

p = People()
p.job = 'teacher'
print(p.job)
```

```
p.name = 'cici'
print(p.name)
```

注意:
使用的slots限制仅对类本身有作用，对于继承的子类不起作用

```
class Cat(Animal):
    pass

c = Cat()
c.age = 1
print(c.age)
```

如果子类也定义slot,那么子类的属性就会变得也受限制，能够添加的属性是子类限制的属性加上父类限制的属性

使用@property

将属性的读写操作直接用一个方法实行来完成。@property将这个方法的操作性质变成属性来使用。使得属性不用暴露于外部，并能够给属性添加逻辑检查。

```
# 一般逻辑，使用set和get方法读写属性的值
class Student(object):
    pass

    def set_age(self,age):
        self.age = age
        # 逻辑检查
        if self.age < 0:
            print(f'error age:{self.age}')

    def get_age(self):
        return self.age

s = Student()
s.set_age(10)
age = s.get_age()
print(age)
```

```
s1 = Student()
s1.set_age(-1)
```

```
# 使用@property

class Student(object):
    # 使用@property，将getter方法变成属性
    @property
    def age(self):
        return self._age

    # 同时@property自己会创建方法属性的setter装饰器，负责把setter方法变成属性
    @age.setter
    def age(self,age):
        self._age = age

s1 = Student()
s1.age = 10
age = s1.age
print(age)
```

只定义@property，那么属性只可读不可写，同时定义@age.setter，那么属性可读可写

```
class Student():
    pass

    # usual_score 只可写？
    @usual_score.setter
    def usual_score(self,value):
        self._usual_score = value

    # final_score 可读可写
    @property
```

```
    def final_score(self):
        return self._final_score

    @final_score.setter
    def final_score(self,value):
        self._final_score = value

    # total_score 只可读
    @property
    def total_score(self):
        return self._usual_score*0.5 + _final_score*0.5

s3 = Student()
s3.usual_score = 90
u_score = s3.usual_score
print(u_score)

s3.final_score = 100
f_score = s3.final_score
print(f_score)

t_score = s3.total_score
print(t_score)
```

以上，需要定义@property后才能定义@usual_score.setter

```
class Student():
    pass

    # usual_score 可度写
    @property
    def usual_score(self):
        return self._usual_score

    @usual_score.setter
    def usual_score(self,value):
```

```
        self._usual_score = value

    # final_score 可读可写
    @property
    def final_score(self):
        return self._final_score

    @final_score.setter
    def final_score(self,value):
        self._final_score = value

    # total_score 只可读
    @property
    def total_score(self):
        return self._usual_score*0.5 +
self._final_score*0.5

s3 = Student()
s3.usual_score = 90
u_score = s3.usual_score
print(u_score)

s3.final_score = 100
f_score = s3.final_score
print(f_score)

t_score = s3.total_score
print(t_score)

#自定义total_score
s3.total_score = 1000
test = s3.total_score
print(test)
# >>AttributeError: can't set attribute
```

```
class Student(object):
    #方法名和属性名重名
    @property
    def age(self):
        return self.age
```

```
class Student(object):
    # 方法名称和实例变量均为birth:
    @property
    def birth(self):
        return self.birth

s = Student()
s.age = 10
print(s.age)
```

**注意

属性方法名和实例变量重名，会造成递归调用，导致栈溢出报错.
但是以上代码为什么没有这个错误。

```
class Screen(object):
    @property
    def width(self):
        return self._width

    @width.setter
    def width(self,value):
        self._width = value

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self,value):
```

```
        self._height = value

    @property
    def resolution(self):
        return self._height * self._width

s = Screen()
s.height = 10
s.width = 10
print(s.resolution)
```

```
# 测试:
s1 = Screen()
s1.width = 1024
s1.height = 768
print('resolution =', s1.resolution)
if s1.resolution == 786432:
    print('测试通过!')
else:
    print('测试失败!')
```

多重继承

class Dog(Animal,Runable)

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。

定制类

重写python类自带方法?

使用枚举类

对于需要大量定义常量的地方，封装为枚举类。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类型的枚举类
Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr',
'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))
```

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.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。

```
@unique
class Weekday(Enum):
    Sun = 0
    Mon = 1
    Tue = 1 #设置重复的值
    Wed = 3
    Thu = 4
    Fri = 5
    Sat = 6
```

访问枚举类型：

```
#访问常量
day1 = Weekday.Mon
print(day1)

day2 = Weekday.Tue
print(day2)

print(Weekday['Mon'])

print(day1 == Weekday.Mon)

print(day1 == Weekday.Tue)

#获取常量的值
print(Weekday.Tue.value)

#通过值获取常量
print(Weekday(1))

#通过不存在的值获取常量
print(Weekday(7))
```

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。

```
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：

class的名称；
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；
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：

当前准备创建的类的对象；
类的名字；
类继承的父类集合；
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 "", line 1, in
> > 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()全部由父类Model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：

排除掉对Model类的修改；
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）；
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？

12. 错误，调试，测试

错误

错误处理

**使用logging打印错误信息，同时让程序继续往下运行

```
import logging

print('start test....')
try:
    print(2/0)
except Exception as e:
    logging.exception(e)

    logging.info(e)

print('end...')
```

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：

```
class FooError(ValueError):
    pass

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

foo('0')
```

调试

断言

将print()来查看的地方，用断言(assert)来替代

使用assert

```
# assert 预期表达式，非预期错误信息

a = 6

assert a == 4, f'a != 4,a:{a}'
```

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：

```
import logging

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

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

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

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。

测试

为了编写单元测试，我们需要引入Python自带的unittest模块。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assertEqual()：

```
self.assertEqual(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']
```

**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。

文档测试

执行注释文档代码

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。

```
# mydict2.py
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()
```

```
$ python mydict2.py
```

13. IO编程

13.1 文件读写

使用with open('file path',mode,encoding,errors)语句。with语句会自动执行close操作

mode：
r：读

rb：读取二进制

w：写

wb：写二进制

a：追加

encoding：指定编码
errors：遇到编码错误处理，如ignore