

22.1 元类(了解)

1. 类也是对象

在大多数编程语言中，类就是一组用来描述如何生成一个对象的代码段。在 Python 中这一点仍然成立：

```
>>> class ObjectCreator(object):
...     pass
...
>>> my_object = ObjectCreator()
>>> print(my_object)
<__main__.ObjectCreator object at 0x8974f2c>
```

但是，Python 中的类还远不止如此。类同样也是一种对象。是的，没错，就是对象。只要你使用关键字 `class`，Python 解释器在执行的时候就会创建一个对象。

下面的代码段：

```
>>> class ObjectCreator(object):
...     pass
...
```

将在内存中创建一个对象，名字就是 `ObjectCreator`。这个对象（类对象 `ObjectCreator`）拥有创建对象（实例对象）的能力。但是，它的本质仍然是一个对象，于是乎你可以对它做如下的操作：

1. 你可以将它赋值给一个变量
2. 你可以拷贝它
3. 你可以为它增加属性
4. 你可以将它作为函数参数进行传递

下面是示例：

```
>>> print(ObjectCreator) # 你可以打印一个类，因为它其实也是一个对象
<class '__main__.ObjectCreator'>
>>> def echo(o):
...     print(o)
...
>>> echo(ObjectCreator) # 你可以将类做为参数传给函数
<class '__main__.ObjectCreator'>
>>> print(hasattr(ObjectCreator, 'new_attribute'))
False
>>> ObjectCreator.new_attribute = 'foo' # 你可以为类增加属性
>>> print(hasattr(ObjectCreator, 'new_attribute'))
```

```
True
>>> print(ObjectCreator.new_attribute)
foo
>>> ObjectCreatorMirror = ObjectCreator # 你可以将类赋值给一个变量
>>> print(ObjectCreatorMirror())
<__main__.ObjectCreator object at 0x8997b4c>
```

2. 动态地创建类

因为类也是对象，你可以在运行时动态地创建它们，就像其他任何对象一样。首先，你可以在函数中创建类，使用 `class` 关键字即可。

```
>>> def choose_class(name):
...     if name == 'foo':
...         class Foo(object):
...             pass
...         return Foo      # 返回的是类，不是类的实例
...     else:
...         class Bar(object):
...             pass
...         return Bar
...
>>> MyClass = choose_class('foo')
>>> print(MyClass) # 函数返回的是类，不是类的实例
<class '__main__.Foo'>
>>> print(MyClass()) # 你可以通过这个类创建类实例，也就是对象
<__main__.Foo object at 0x89c6d4c>
```

但这还不够动态，因为你仍然需要自己编写整个类的代码。由于类也是对象，所以它们必须是通过什么东西来生成的才对。

当你使用 `class` 关键字时，Python 解释器自动创建这个对象。但就和 Python 中的大多数事情一样，Python 仍然提供给你手动处理的方法。

还记得内建函数 `type` 吗？这个古老但强大的函数能够让你知道一个对象的类型是什么，就像这样：

```
>>> print(type(1)) # 数值的类型
<type 'int'>
>>> print(type("1")) # 字符串的类型
<type 'str'>
>>> print(type(ObjectCreator())) # 实例对象的类型
<class '__main__.ObjectCreator'>
>>> print(type(ObjectCreator)) # 类的类型
<class 'type'>
```

仔细观察上面的运行结果，发现使用 `type` 对 `ObjectCreator` 查看类型是，答案为 `type`，是不是有些惊讶。。。看下面

3. 使用 type 创建类

type 还有一种完全不同的功能，动态的创建类。

type 可以接受一个类的描述作为参数，然后返回一个类。（要知道，根据传入参数的不同，同一个函数拥有两种完全不同的用法是一件很傻的事情，但这在 Python 中是为了保持向后兼容性）

type 可以像这样工作：

type(类名, 由父类名称组成的元组（针对继承的情况，可以为空），包含属性的字典（名称和值）)

比如下面的代码：

```
In [2]: class Test: #定义了一个 Test 类
...:     pass
...:
In [3]: Test() # 创建了一个 Test 类的实例对象
Out[3]: <__main__.Test at 0x10d3f8438>
```

可以手动像这样创建：

```
Test2 = type("Test2", (), {}) # 定了一个 Test2 类
In [5]: Test2() # 创建了一个 Test2 类的实例对象
Out[5]: <__main__.Test2 at 0x10d406b38>
```

我们使用"Test2"作为类名，并且也可以把它当做一个变量来作为类的引用。类和变量是不同的，这里没有任何理由把事情弄的复杂。即 type 函数中第 1 个实参，也可以叫做其他的名字，这个名字表示类的名字

```
In [23]: MyDogClass = type('MyDog', (), {})
```

```
In [24]: print(MyDogClass)
<class '__main__.MyDog'>
```

使用 help 来测试这 2 个类

```
In [10]: help(Test2) # 用 help 查看 Test 类
```

Help on class Test in module __main__:

```
class Test(builtins.object)
|   Data descriptors defined here:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
```

```
|  __weakref__  
|  list of weak references to the object (if defined)
```

In [8]: help(Test2) #用 help 查看 Test2 类

Help on class Test2 in module __main__:

```
class Test2(builtins.object)  
|   Data descriptors defined here:  
|  
|   __dict__  
|       dictionary for instance variables (if defined)  
|  
|   __weakref__  
|       list of weak references to the object (if defined)
```

4. 使用 type 创建带有属性的类

type 接受一个字典来为类定义属性，因此

```
>>> Foo = type('Foo', (), {'bar': True})
```

可以翻译为：

```
>>> class Foo(object):  
...     bar = True
```

请问 bar 是什么属性？

并且可以将 Foo 当成一个普通的类一样使用：

```
>>> print(Foo)  
<class '__main__.Foo'>  
>>> print(Foo.bar)  
True  
>>> f = Foo()  
>>> print(f)  
<__main__.Foo object at 0x8a9b84c>  
>>> print(f.bar)  
True
```

当然，你可以继承这个类，代码如下：

```
>>> class FooChild(Foo):  
...     pass
```

就可以写成：

```
>>> FooChild = type('FooChild', (Foo,), {})  
>>> print(FooChild)
```

```
<class '__main__.FooChild'>
>>> print(FooChild.bar) # bar 属性是由 Foo 继承而来
True
```

注意:

- type 的第 2 个参数，元组中是父类的名字，而不是字符串
- 添加的属性是类属性，并不是实例属性

5. 使用 type 创建带有方法的类

最终你会希望为你的类增加方法。只需要定义一个有着恰当签名的函数并将其作为属性赋值就可以了。

添加实例方法

```
In [46]: def echo_bar(self): # 定义了一个普通的函数
...:     print(self.bar)
...:
```

```
In [47]: FooChild = type('FooChild', (Foo,), {'echo_bar': echo_bar}) #
让 FooChild 类中的 echo_bar 属性，指向了上面定义的函数
```

```
In [48]: hasattr(Foo, 'echo_bar') # 判断 Foo 类中 是否有 echo_bar 这个属性
Out[48]: False
```

```
In [49]:
```

```
In [49]: hasattr(FooChild, 'echo_bar') # 判断 FooChild 类中 是否有 echo_b
ar 这个属性
Out[49]: True
```

```
In [50]: my_foo = FooChild()
```

```
In [51]: my_foo.echo_bar()
True
```

添加静态方法

```
In [36]: @staticmethod
...: def test_static():
...:     print("static method ....")
...:
```

```
In [37]: Foochild = type('Foochild', (Foo,), {"echo_bar": echo_bar, "te
st_static": test_static})
```

```
In [38]: fooclid = Foochild()
```

```
In [39]: fooclid.test_static
Out[39]: <function __main__.test_static>
```

```
In [40]: fooclid.test_static()
static method ....
```

```
In [41]: fooclid.echo_bar()
True
```

添加类方法

```
In [42]: @classmethod
...: def test_class(cls):
...:     print(cls.bar)
...:
```

```
In [43]:
```

```
In [43]: Foochild = type('Foochild', (Foo,), {"echo_bar":echo_bar, "test_static": test_static, "test_class": test_class})
```

```
In [44]:
```

```
In [44]: fooclid = Foochild()
```

```
In [45]: fooclid.test_class()
True
```

你可以看到，在 Python 中，类也是对象，你可以动态的创建类。这就是当你使用关键字 **class** 时 Python 在幕后做的事情，而这就是通过元类来实现的。

较为完整的使用 **type** 创建类的方式：

```
class A(object):
    num = 100

    def print_b(self):
        print(self.num)

    @staticmethod
    def print_static():
        print("----haha-----")

    @classmethod
    def print_class(cls):
        print(cls.num)

B = type("B", (A,), {"print_b": print_b, "print_static": print_static, "print_class": print_class})
```

```
b = B()
b.print_b()
b.print_static()
b.print_class()
# 结果
# 100
# ----haha-----
# 100
```

6. 到底什么是元类（终于到主题了）

元类就是用来创建类的“东西”。你创建类就是为了创建类的实例对象，不是吗？但是我们已经学习到了 Python 中的类也是对象。

元类就是用来创建这些类（对象）的，元类就是类的类，你可以这样理解为：

```
MyClass = MetaClass() # 使用元类创建一个对象，这个对象称为“类”
my_object = MyClass() # 使用“类”来创建出实例对象
```

你已经看到了 `type` 可以让你像这样做：

```
MyClass = type('MyClass', (), {})
```

这是因为函数 **type** 实际上是一个元类。`type` 就是 Python 在背后用来创建所有类的元类。现在你想知道那为什么 `type` 会全部采用小写形式而不是 `Type` 呢？好吧，我猜这是为了和 `str` 保持一致性，`str` 是用来创建字符串对象的类，而 `int` 是用来创建整数对象的类。`type` 就是创建类对象的类。你可以通过检查 `__class__` 属性来看到这一点。Python 中所有的东西，注意，我是指所有的东西——都是对象。这包括整数、字符串、函数以及类。它们全部都是对象，而且它们都是从一个类创建而来，这个类就是 `type`。

```
>>> age = 35
>>> age.__class__
<type 'int'>
>>>
>>> name = 'bob'
>>> name.__class__
<type 'str'>
>>>
>>> def foo(): pass
>>> foo.__class__
<type 'function'>
>>>
>>> class Bar(object): pass
>>> b = Bar()
>>> b.__class__
<class '__main__.Bar'>
>>>
```

现在，对于任何一个`__class__`的`__class__`属性又是什么呢？

```
>>> age.__class__.__class__
<type 'type'>
>>> foo.__class__.__class__
<type 'type'>
>>> b.__class__.__class__
<type 'type'>
```

因此，元类就是创建类这种对象的东西。`type` 就是 Python 的内建元类，当然了，你也可以创建自己的元类。

7. `__metaclass__` 属性

你可以在定义一个类的时候为其添加`__metaclass__`属性。

```
class Foo(object):
    metaclass = something...
    ...省略...
```

如果你这么做了，Python 就会用元类来创建类 `Foo`。小心点，这里面有些技巧。你首先写下 `class Foo(object)`，但是类 `Foo` 还没有在内存中创建。Python 会在类的定义中寻找`__metaclass__`属性，如果找到了，Python 就会用它来创建类 `Foo`，如果没有找到，就会用内建的 `type` 来创建这个类。把下面这段话反复读几次。当你写如下代码时：

```
class Foo(Bar):
    pass
```

Python 做了如下的操作：

1. `Foo` 中有`__metaclass__`这个属性吗？如果是，Python 会通过`__metaclass__`创建一个名字为 `Foo` 的类(对象)
2. 如果 Python 没有找到`__metaclass__`，它会继续在 `Bar`（父类）中寻找`__metaclass__`属性，并尝试做和前面同样的操作。
3. 如果 Python 在任何父类中都找不到`__metaclass__`，它就会在模块层次中去寻找`__metaclass__`，并尝试做同样的操作。
4. 如果还是找不到`__metaclass__`，Python 就会用内置的 `type` 来创建这个类对象。

现在的问题就是，你可以在`__metaclass__`中放置些什么代码呢？答案就是：**可以创建一个类的东西**。那为什么可以用来创建一个类呢？`type`，或者任何使用到 `type` 或者子类化 `type` 的东东都可以。

8. 自定义元类

元类的主要目的就是为了**当创建类时能够自动地改变类**。

假想一个很傻的例子，你决定在你的模块里所有的类的属性都应该是大写形式。有好几种方法可以办到，但其中一种就是通过在模块级别设定`__metaclass__`。采用这种方法，这个模块中的所有类都会通过这个元类来创建，我们只需要告诉元类把所有的属性都改成大写形式就万事大吉了。

幸运的是，`__metaclass__`实际上可以被任意调用，它并不需要是一个正式的类。所以，我们这里就先以一个简单的函数作为例子开始。

```
#-*- coding:utf-8 -*-
def upper_attr(class_name, class_parents, class_attr):

    #遍历属性字典，把不是__开头的属性名字变为大写
    new_attr = {}
    for name,value in class_attr.items():
        if not name.startswith("__"):
            new_attr[name.upper()] = value

    #调用 type 来创建一个类
    return type(class_name, class_parents, new_attr)

class Foo(object, metaclass=upper_attr):
    bar = 'bip'

print(hasattr(Foo, 'bar'))
print(hasattr(Foo, 'BAR'))

f = Foo()
print(f.BAR)
```

现在让我们再做一次，这一次用一个真正的 `class` 来当做元类。

```
#coding=utf-8

class UpperAttrMetaClass(type):
    # __new__ 是在__init__之前被调用的特殊方法
    # __new__是用来创建对象并返回之的方法
    # 而__init__只是用来将传入的参数初始化给对象
    # 你很少用到__new__，除非你希望能够控制对象的创建
    # 这里，创建的对象是类，我们希望能够自定义它，所以我们这里改写__new__
    # 如果你希望的话，你也可以在__init__中做些事情
    # 还有一些高级的用法会涉及到改写__call__特殊方法，但是我们这里不用
    def __new__(cls, class_name, class_parents, class_attr):
        # 遍历属性字典，把不是__开头的属性名字变为大写
```

```
new_attr = {}
for name, value in class_attr.items():
    if not name.startswith("__"):
        new_attr[name.upper()] = value

# 方法 1: 通过'type'来做类对象的创建
return type(class_name, class_parents, new_attr)

# 方法 2: 复用 type.__new__ 方法
# 这就是基本的 OOP 编程, 没什么魔法
# return type.__new__(cls, class_name, class_parents, new_attr)

# python3 的用法
class Foo(object, metaclass=UpperAttrMetaClass):
    bar = 'bip'

print(hasattr(Foo, 'bar'))
# 输出: False
print(hasattr(Foo, 'BAR'))
# 输出: True

f = Foo()
print(f.BAR)
# 输出: 'bip'
```

就是这样, 除此之外, 关于元类真的没有别的可说的了。但就元类本身而言, 它们其实是很简单的:

1. 拦截类的创建
2. 修改类
3. 返回修改之后的类

究竟为什么要使用元类?

现在回到我们的大主题上来, 究竟是因为什么你会去使用这样一种容易出错且晦涩的特性? 好吧, 一般来说, 你根本就用不上它:

“元类就是深度的魔法, 99%的用户应该根本不必为此操心。如果你想搞清楚究竟是否需要用到元类, 那么你就不需要它。那些实际用到元类的人都非常清楚地知道他们需要做什么, 而且根本不需要解释为什么要用元类。”—— Python 界的领袖 **Tim Peters**

22.2 元类实现 ORM

1. ORM 是什么

ORM 是 python 编程语言后端 web 框架 Django 的核心思想，“Object Relational Mapping”，即对象-关系映射，简称 ORM。（JAVA 也是 ORM）

一个句话理解就是：创建一个实例对象，用创建它的类名当做数据表名，用创建它的类属性对应数据表的字段，当对这个实例对象操作时，能够对应 MySQL 语句



demo:

```
class User(父类省略):  
    uid = ('uid', "int unsigned")  
    name = ('username', "varchar(30)")  
    email = ('email', "varchar(30)")  
    password = ('password', "varchar(30)")  
    ...省略...
```

```
u = User(uid=12345, name='Michael', email='test@orm.org', password='my-pwd')  
u.save()  
# 对应如下 sql 语句  
# insert into User (username,email,password,uid)  
# values ('Michael','test@orm.org','my-pwd',12345)
```

说明

1. 所谓的 ORM 就是让开发者在操作数据库的时候，能够像操作对象时通过 xxxx. 属性=yyyy 一样简单，这是开发 ORM 的初衷
2. 只不过 ORM 的实现较为复杂，Django 中已经实现了很复杂的操作，本节知识主要通过完成一个 insert 相类似的 ORM，理解其中的道理就可以了

2. 通过元类简单实现 ORM 中的 insert 功能

```
class ModelMetaClass(type):
    def __new__(cls, name, bases, attrs):
        mappings = dict()
        # 判断是否需要保存
        for k, v in attrs.items():
            # 判断是否是指定的 StringField 或者 IntegerField 的实例对象
            if isinstance(v, tuple):
                print('Found mapping: %s ==> %s' % (k, v))
                mappings[k] = v

        # 删除这些已经在字典中存储的属性
        for k in mappings.keys():
            attrs.pop(k)

        # 将之前的 uid/name/email/password 以及对应的对象引用、类名字
        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)

class User(metaClass=ModelMetaClass):
    uid = ('uid', "int unsigned")
    name = ('username', "varchar(30)")
    email = ('email', "varchar(30)")
    password = ('password', "varchar(30)")
    # 当指定元类之后，以上的类属性将不在类中，而是在__mappings__属性指定的字典中存储
    # 以上 User 类中有
    # __mappings__ = {
    #     "uid": ('uid', "int unsigned")
    #     "name": ('username', "varchar(30)")
    #     "email": ('email', "varchar(30)")
    #     "password": ('password', "varchar(30)")
    # }
    # __table__ = "User"
    def __init__(self, **kwargs):
        for name, value in kwargs.items():
            setattr(self, name, value)
```

```
def save(self):
    fields = []
    args = []
    for k, v in self.__mappings__.items():
        fields.append(v[0])
        args.append(getattr(self, k, None))

    sql = 'insert into %s (%s) values (%s)' % (self.__table__, ','.join(fields), ','.join([str(i) for i in args]))
    print('SQL: %s' % sql)
```

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

执行的效果:

```
Found mapping: password ==> ('password', 'varchar(30)')
Found mapping: email ==> ('email', 'varchar(30)')
Found mapping: uid ==> ('uid', 'int unsigned')
Found mapping: name ==> ('username', 'varchar(30)')
SQL: insert into User (uid,password,username,email) values (12345,my-pwd,Michael,test@orm.org)
```

3. 完善对数据类型的检测

```
class ModelMetaclass(type):
    def __new__(cls, name, bases, attrs):
        mappings = dict()
        # 判断是否需要保存
        for k, v in attrs.items():
            # 判断是否是指定的 StringField 或者 IntegerField 的实例对象
            if isinstance(v, tuple):
                print('Found mapping: %s ==> %s' % (k, v))
                mappings[k] = v

        # 删除这些已经在字典中存储的属性
        for k in mappings.keys():
            attrs.pop(k)

        # 将之前的 uid/name/email/password 以及对应的对象引用、类名字
        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)
```

```
class User(metaclass=ModelMetaclass):
```

```
uid = ('uid', "int unsigned")
name = ('username', "varchar(30)")
email = ('email', "varchar(30)")
password = ('password', "varchar(30)")
# 当指定元类之后，以上的类属性将不在类中，而是在__mappings__属性指定的字典中存储
# 以上 User 类中有
# __mappings__ = {
#     "uid": ('uid', "int unsigned")
#     "name": ('username', "varchar(30)")
#     "email": ('email', "varchar(30)")
#     "password": ('password', "varchar(30)")
# }
# __table__ = "User"
def __init__(self, **kwargs):
    for name, value in kwargs.items():
        setattr(self, name, value)

def save(self):
    fields = []
    args = []
    for k, v in self.__mappings__.items():
        fields.append(v[0])
        args.append(getattr(self, k, None))

    args_temp = list()
    for temp in args:
        # 判断入如果是数字类型
        if isinstance(temp, int):
            args_temp.append(str(temp))
        elif isinstance(temp, str):
            args_temp.append("'%s'" % temp)
    sql = 'insert into %s (%s) values (%s)' % (self.__table__, ','.join(fields), ','.join(args_temp))
    print('SQL: %s' % sql)

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

运行效果如下:

```
Found mapping: uid ==> ('uid', 'int unsigned')
Found mapping: password ==> ('password', 'varchar(30)')
Found mapping: name ==> ('username', 'varchar(30)')
Found mapping: email ==> ('email', 'varchar(30)')
```

```
SQL: insert into User (email,uid,password,username) values ('test@orm.org',12345,'my-pwd','Michael')
```

4. 抽取到基类中

```
class ModelMetaclass(type):
    def __new__(cls, name, bases, attrs):
        mappings = dict()
        # 判断是否需要保存
        for k, v in attrs.items():
            # 判断是否是指定的 StringField 或者 IntegerField 的实例对象
            if isinstance(v, tuple):
                print('Found mapping: %s ==> %s' % (k, v))
                mappings[k] = v

        # 删除这些已经在字典中存储的属性
        for k in mappings.keys():
            attrs.pop(k)

        # 将之前的 uid/name/email/password 以及对应的对象引用、类名字
        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)

class Model(object, metaclass=ModelMetaclass):
    def __init__(self, **kwargs):
        for name, value in kwargs.items():
            setattr(self, name, value)

    def save(self):
        fields = []
        args = []
        for k, v in self.__mappings__.items():
            fields.append(v[0])
            args.append(getattr(self, k, None))

        args_temp = list()
        for temp in args:
            # 判断如果是数字类型
            if isinstance(temp, int):
                args_temp.append(str(temp))
            elif isinstance(temp, str):
                args_temp.append("'%s'" % temp)
        sql = 'insert into %s (%s) values (%s)' % (self.__table__, ','.join(fields), ','.join(args_temp))
        print('SQL: %s' % sql)
```

```
class User(Model):
    uid = ('uid', "int unsigned")
    name = ('username', "varchar(30)")
    email = ('email', "varchar(30)")
    password = ('password', "varchar(30)")

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

22.3 接口类与抽象类

继承有两种用途：

```
"""
一：继承基类的方法，并且做出自己的改变或者扩展（代码重用）
二：声明某个子类兼容于某基类，定义一个接口类 Interface，接口类中定义了一些接口名（就是函数名）
且并未实现接口的功能，子类继承接口类，并且实现接口中的功能
三、接口隔离原则：使用多个专门的接口，而不使用单一的总接口。即客户端不应该依赖那些不需要的接口
"""
"""
接口类：基于同一个接口实现的类 刚好满足接口隔离原则 面向对象开发的思想规范
接口类，python 原生不支持 在 python 中，并没有接口类这种东西，即便不通过专门的模块定义接口，我们也应该有一些基本的概念
"""
```

一、接口类单继承

我们来看一段代码去了解为什么需要接口类

```
class Alipay:
```



```
def pay(self,money):
    print('支付宝支付了')
class Apppay:
    def pay(self,money):
        print('苹果支付了')
class Weicht:
    def pay(self,money):
        print('微信支付了')
def pay(payment,money):          # 支付函数，总体负责支付，对应支付的对
象和要支付的金额
    payment.pay(money)
p=Alipay()
pay(p,200)          #支付宝支付了
```

这段代码，实现了一个有趣的功能，就是通过一个总体的支付函数，实现了不同种类的支付方式，不同是支付方式作为对象，传入函数中
但是开发中容易出现一些问题，那就是类中的函数名不一致，就会导致调用的时候找不到类中对应方法，例题如下：

```
class Alipay:
    def paying(self,money):      #这里类的方法可能由于程序员的疏忽，写
的 not 是一致的 pay, 导致后面调用的时候找不到 pay
        print('支付宝支付了')
class Apppay:
    def pay(self,money):
        print('苹果支付了')
class Weicht:
    def pay(self,money):
        print('微信支付了')
def pay(payment,money):          # 支付函数，总体负责支付，对应支付的对
象和要支付的金额
    payment.pay(money)
p=Alipay()    #不报错
pay(p,200)    #调用执行就会报错, 'Alipay' object has no attribute
'pay'
```

这时候怎么办呢？可以手动抛异常：**NotImplementedError** 来解决开发中遇到的问题

```
class payment:
    def pay(self, money):
        e=Exception('缺少编写 pay 方法')
        raise e    #手动抛异常
class Alipay(payment):
    def paying(self, money): # 这里类的方法不是一致的 pay, 导致后面调用
                              的时候找不到 pay
        print('支付宝支付了')
def pay(payment, money): # 支付函数, 总体负责支付, 对应支付的对象和要
                          支付的金额
    payment.pay(money)

p = Alipay() # 不报错
pay(p, 200)
```

也可以借用 abc 模块来处理这种错误

```
from abc import abstractmethod, ABCMeta    #接口类中定义了一些接口
名: Pay, 且并未实现接口的功能, 子类继承接口类, 并且实现接口中的功能
class Payment(metaclass=ABCMeta):    #抽象出的共同功能 Pay
    @abstractmethod
    def pay(self, money):pass    #这里面的 pay 来源于下面类中的方法
pay, 意思把这个方法规范为统一的标准, 另外建一个规范类 Payment
class Alipay(Payment):
    def paying(self, money):    #这里出现 paying 和我们规范的 pay 不
    一样, 那么在实例化 Alipay 的时候就会报错
        print('支付宝支付了')
class Weicht(Payment):
    def pay(self, money):
        print('微信支付了')
def pay(pay_obj, money):
    pay_obj.pay(money)
```

`p=Alipay()` #实例化的时候就会报错 `Can't instantiate abstract class Alipay with abstract methods pay` 之前两个例子都是在执行的时候报错，这里不一样的是实例化就会知道是哪里发生错误了

"""

总结：用 `abc` 模块装饰后，在实例化的时候就会报错，那么当我们代码很长的時候，就可以早一点预知错误，所以以后在接口类类似問題中用这个模块
接口继承实质上是要求“做出一个良好的抽象，这个抽象规定了一个兼容接口，使得外部调用者无需关心具体细节，可一视同仁的处理实现了特定接口的所有对象”——这在程序设计上，叫做归一化。

"""

二、接口类多继承

```
from abc import abstractmethod, ABCMeta
class Walk_animal(metaclass=ABCMeta):
    @abstractmethod
    def walk(self):
        print('walk')
class Swim_animal(metaclass=ABCMeta):
    @abstractmethod
    def swim(self):pass
class Fly_animal(metaclass=ABCMeta):
    @abstractmethod
    def fly(self):pass
#如果正常一个老虎有跑和跑的方法的话，我们会这么做
class Tiger:
    def walk(self):pass
    def swim(self):pass
#但是我们使用接口类多继承的话就简单多了，并且规范了相同功能
class Tiger(Walk_animal,Swim_animal):pass
#如果此时再有一个天鹅 swan,会飞，走，游泳 那么我们这么做
class Swan(Walk_animal,Swim_animal, Fly_animal):pass
# 这就是接口多继承
```

为什么需要接口类

三、抽象类

#抽象类

抽象类的本质还是类，

指的是一组类的相似性，包括数据属性（如 `all_type`）和函数属性（如 `read`、`write`），而接口只强调函数属性的相似性

"""

1.抽象类是一个介于类和接口之间的一个概念，同时具备类和接口的部分特性，可以用来**实现归一化设计**

2.在继承抽象类的过程中，我们应该尽量避免多继承；

3.而在继承接口的时候，我们反而鼓励你来多继承接口

一般情况下 单继承 能实现的功能都是一样的，所以在父类中可以有一些简单的基础实现

多继承的情况 由于功能比较复杂，所以不容易抽象出相同的功能的具体实现写在父类中

"""

为什么要有抽象类

从设计角度去看，如果类是从现实对象抽象而来的，那么抽象类就是基于类抽象而来的。

从实现角度来看，抽象类与普通类的不同之处在于：**抽象类中有抽象方法，该类不能被实例化，只能被继承，且子类必须实现抽象方法。**这一点与接口有点类似，但其实是不同的

#一切皆文件

import abc #利用 abc 模块实现抽象类

```
class All_file(metaclass=abc.ABCMeta):
    all_type='file'
    @abc.abstractmethod #定义抽象方法，无需实现功能
    def read(self):
```

```
        '子类必须定义读功能'
        pass

    @abc.abstractmethod #定义抽象方法, 无需实现功能
    def write(self):
        '子类必须定义写功能'
        pass

# class Txt(All_file):
#     pass
#
# t1=Txt() #报错,子类没有定义抽象方法

class Txt(All_file): #子类继承抽象类, 但是必须定义 read 和 write 方法
    def read(self):
        print('文本数据的读取方法')

    def write(self):
        print('文本数据的读取方法')

class Sata(All_file): #子类继承抽象类, 但是必须定义 read 和 write 方法
    def read(self):
        print('硬盘数据的读取方法')

    def write(self):
        print('硬盘数据的读取方法')

class Process(All_file): #子类继承抽象类, 但是必须定义 read 和 write
方法
    def read(self):
        print('进程数据的读取方法')

    def write(self):
        print('进程数据的读取方法')

wenbenwenjian=Txt()

yingpanwenjian=Sata()

jinchengwenjian=Process()

#这样大家都是被归一化了,也就是一切皆文件的思想
wenbenwenjian.read()
```

```
yingpanwenjian.write()
jinchengwenjian.read()

print(wenbenwenjian.all_type)
print(yingpanwenjian.all_type)
print(jinchengwenjian.all_type)
```

四、扩展：

不管是抽象类还是接口类： 面向对象的开发规范 所有的接口类和抽象类都不能实例化

java：

java 里的所有类的继承都是**单继承**，所以抽象类完美的解决了单继承需求中的规范问题

但对于多继承的需求，由于 java 本身语法的不支持，所以创建了接口 **Interface** 这个概念来解决多继承的规范问题


python：

python 中没有接口类：

python 中自带多继承 所以我们直接用 **class** 来实现了接口类

python 中支持抽象类： 一般情况下 单继承 不能实例化

且可以实现 python 代码

 <https://blog.csdn.net/zhangquan2015/article/details/82808399>

五、注意

"""

1. 多继承问题

在继承抽象类的过程中，我们应该尽量避免多继承；

而在继承接口的时候，我们反而鼓励你来多继承接口

2. 方法的实现

在抽象类中，我们可以对一些抽象方法做出基础实现；

而在接口类中，任何方法都只是一种规范，具体的功能需要子类实现

"""

