### 传值和传引用

函数参数的传递机制本质上是调用函数（过程）或者被调用函数（过程，）在调用发生时进行通信的方法问题。基本的参数传递有两种：值传递和引用传递。

值传递（pass by value）的过程中，被调函数的形式参数作为被调函数的局部变量处理，在堆栈中开辟了内存空间来存放由主调函数放进来的实参的值，形成了实参的一个副本。其**特点**是被调函数对形式参数的任何操作都是作为局部变量进行，不会影响主调函数的变量的值

引用传递（pass by reference）过程中，被调函数的形式参数虽然也作为局部变量堆栈中开辟了内存空间，但是这是存放的是有主调函数放进来的实参变量的地址。被调函数对形参的任何操作都被处理成间接寻址，即通过堆栈中存放的地址访问主调函数中的实参变量。其**特点**对形参做的任何操作都会影响到主调函数中的实参变量。

def test(c):

print "test before "

print id(c) # 39601564

c+=2

print "test after +"

print id(c) # 39601564

return c

if \_\_name\_\_=="\_\_main\_\_":

a=2

print "main before invoke test"

print id(a) # 39601564

n=test(a)

print "main afterf invoke test"

print a

print id(a) # 39601564

**总结**：python不允许程序员选择采用传值还是传递引用。Python参数传递的方法是“传递对象引用”的方式。这种方式相当于传值和传递引用的一种综合。如果遇到可变对象（mutable）（字典dict或者列表list）的引用，那么就通过“传引用”来传递对象。如果是一个不可变对象（immutable）（数字，字符或者元素）的引用，那么就通过“传值”来传递对象。

### 整数对象的实现

对于数值较小的整数对象在内存中会很频繁的使用，如果每次都向内存申请空间，请求释放，会影响python’性能，好在整数对象属于不可变对象，可悲共享而不会被修改导致问题。所以为小整数对象划定一个范围，即**小整数对象池**。在python运行时初始化并创建范围内的所有整数，这个范围内的整数对象是被共享的，一次创建，多次使用。

### 3. Metaclass

（1）什么是类？

类就是一组用来描述如何生成一个对象的代码段。但是在python中远不止如此。类同样也是一种对象。只要使用了class，那么python解释器在执行的时候就会创建一个对象。

class ObjectCreator(object)

上面的代码将在内存中创建一个对象，名字就是ObjectCreator。这个对象（类）自身拥有创建（类实例）的能力，而这就是为什么他是一个类的原因。

动态创建类

因为类也是对象，可以在运行时动态的创建他们，就像其他任何对象一样。只要使用class关键字即可。然而，这还是不够动态，我们其实可以使用伟大的type函数。像下面这样：

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

举个例子：

>>> class MyShinyClass(object):

…       pass

>>>MyShinyClass = type('MyShinyClass', (), {})  # 返回一个类对象

两者效果是一样的。

当然也可以继承这个类：

>>> class FooChild(Foo):

…       pass

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

两者的效果又是一样的。

我们可以看到，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的时候，python在幕后做的事情，而这些就是通过metaclass来实现的。

（2）那么到底什么是元类？

元类就是类的类。我们从上面可以看出type实际上是一个元类。因为它就是python在背后用来创建所有类的元类。

举一些例子：

>>> 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'>

那么对上面的类来说，\_\_class\_\_的\_\_class\_\_的属性是什么呢？

>>> a.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> age.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> foo.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> b.\_\_class\_\_.\_\_class\_\_

<type 'type'>

（3）\_\_metaclass\_\_属性

你可以在写一个类的时候为其添加\_\_metaclass\_\_属性：

class Foo(object):

\_\_metaclass\_\_ = something…

class MyList(list, metaclass=ListMetaclass): # 也可以通过这种方法实现

pass

如果你这么做了，那么python就会使用元类来创建Foo。小心哦，在写下class Foo(object)

但是类对象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，那么python就会用它来创建类Foo，如果没有找到，就会用内建的type类来创建类Foo。

class Foo(Bar):

    pass

Foo在中有\_\_metaclass\_\_这个属性吗？

有->python在内存中通过\_\_metaclass\_\_创建一个名字为Foo的**类对象**。

没有->它会继续在父类中寻找->层次模块中寻找->用内置的type中寻找

总结：其实，就元类而言，1.拦截类的创建，2.修改类，3.返回修改之后的类别

主要作用：元类的主要用途是创建API。一个典型的例子是Django ORM。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。

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，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和属性类别StringField、IntengerField是由ORM框架提供的，剩下的魔术方法如save()全部使用metaclass自动完成。

现在我们来实现一下

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的基础上定义各种类型的Field，比如StringField，IntegerField

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')

下一步编写最复杂udeModelMetaClass

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))

编写如下的代码：

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

u.save()

会有如下的输出：

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]

### @staticmethod和@classmethod

Python中的三种方法：

1. 静态方法（staticmethod）
2. 类方法（classmethod）
3. 实例方法

def foo(x):

print "executing foo(%s)"%(x)

class A(object):

def foo(self,x): # self表示的是实例本身

print "executing foo(%s,%s)"%(self,x)

@classmethod

def class\_foo(cls,x): # cls是这个类本身

print "executing class\_foo(%s,%s)"%(cls,x)

@staticmethod

def static\_foo(x): # 如果使用了staticmethod就可以忽略这个self

print "executing static\_foo(%s)"%x

a=A()

|  |  |  |  |
| --- | --- | --- | --- |
| 、 | 实例方法 | 类方法 | 静态方法 |
| a=A() | a.foo(x) | a.class\_foo(x) | a.static\_foo(x) |
| A | 不可用 | A.class\_foo(x) | A.static\_foo(x) |

这里先理解下函数参数里面的self和cls.这个self和cls是对类或者实例的绑定,对于一般的函数来说我们可以这么调用foo(x),这个函数就是最常用的,它的工作跟任何东西(类,实例)无关.对于实例方法

在类里每次定义方法的时候都需要绑定这个实例,就是foo(self, x),为什么要这么做呢?因为实例方法的调用离不开实例,我们需要把实例自己传给函数,调用的时候是这样的a.foo(x)(其实是foo(a, x)).类方法一样,只不过它传递的是类而不是实例,A.class\_foo(x).注意这里的self和cls可以替换别的参数,但是python的约定是这俩,还是不要改的好.

对于静态方法其实和普通的方法一样,不需要对谁进行绑定,唯一的区别是调用的时候需要使用a.static\_foo(x)或者A.static\_foo(x)来调用.

**区别**：一个是要用绑定实例，一个是传递类

### 自省

自省就是面向对象的语言所写的程序在运行时,所能知道对象的类型.简单一句就是运行时能够获得对象的类型.比如type(),dir(),getattr(),hasattr(),isinstance().

### 单下划线

（1）解释器中：\_符号是指交互解释器中最后一次执行语句的返回结果。这种用法最初出现在CPython解释器中，其他解释器后来也都跟进了。

（2）作为名称使用：这个跟上面有点类似。\_用作**被丢弃**的名称：**for** \_ **in** range(n)

（3）i18n：\_还可以被用作函数名。这种情况，单下划线经常被用作国际化和本地化字符串翻译查询的函数名。

（4）以单下划线的前缀名称指定了这个名称是“**私有的**”。在某些导入“import \*”的场景中，下一个使用你代码的人（或你的本人），会明白这个名称仅仅内部使用。以单下划线的名称都不会被导入，除非模块/包的\_\_all\_\_列表名称确实包含了这些名称。

### 6. 双**下划线**

双下划线作为前缀对解释器有特定含义。Python会改写这些名称，以免和子类的中定义的名称产生冲突。任何\_\_spam这种形式（至少两个下划线做开头，绝大部分还有一个下划线做结尾）的标识符，都会被替换为\_classname\_\_spam，其中classname是当前类名并带上一个下划线做前缀。

举个栗子

>>> class A(object):

... def \_internal\_use(self):

... pass

... def \_\_method\_name(self):

... pass

...

>>> dir(A())

['\_A\_\_method\_name', ..., '\_internal\_use']

可以看到，\_internal\_use没有变化，但\_\_method\_name被改写成\_ClassName\_method\_name现在创建一个A的子类B，这就不会轻易覆盖掉A中的\_\_method\_name了：

>>> class B(A):

... def \_\_method\_name(self):

... pass

...

>>> dir(B())

['\_A\_\_method\_name', '\_B\_\_method\_name', ..., '\_internal\_use']

前后都带有双下划线的名称

这是Python的特殊命名方法，这是一种惯例。一种确保Python系统中的名称不会跟用户自定义的名称发生冲突的方式。

### 字符串格式化%和format

.format在血多方面看起来更便利。对于%最麻烦的地方是她无法同时传递一个变量和一个元素，你可能认为下面的代码不会有什么问题：

"hi there %s" % name

但是如果name加号是(1,2,3)，它将会抛出一个TypeError异常。为了保证她总是正确，你必须这么做:

"hi there %s" % (name,) # 提供一个单元素的数组而不是一个参数

### \*args and \*\*kwargs

使用\*args和\*\*kwargs只是为了方便并没有强制使用它们

当你不确定你的函数里将要传递多少参数时你可以用\*args。例如，他可以传递任意数量的参数

>>> def print\_everything(\*args):

for count, thing in enumerate(args):

... print '{0}. {1}'.format(count, thing)

...

>>> print\_everything('apple', 'banana', 'cabbage')

0. apple

1. banana

2. cabbage

相似的，\*\*kwargs允许你使用没有实现定义的参数名称：

>>> def table\_things(\*\*kwargs):

... for name, value in kwargs.items():

... print '{0} = {1}'.format(name, value)

...

>>> table\_things(apple = 'fruit', cabbage = 'vegetable')

cabbage = vegetable

apple = fruit

你也可以混合着用，命名参数首先获得参数值然后所有的其他参数都传递给\*args和\*\*kwargs。命名参数在列表的最前端。例如

def table\_things(titlestring, \*\*kwargs)

\*args和\*\*kwargs可以同时在函数的定义中，但是\*args必须在\*\*kwargs前面

当调用函数的时候可以用\*和\*\*语法：

>>> def print\_three\_things(a, b, c):

... print 'a = {0}, b = {1}, c = {2}'.format(a,b,c)

...

>>> mylist = ['aardvark', 'baboon', 'cat']

>>> print\_three\_things(\*mylist)

a = aardvark, b = baboon, c = cat

def foo(\*args, \*\*kwargs):

print 'args = ', args

print 'kwargs = ', kwargs

print '---------------------------------------'

if \_\_name\_\_ == '\_\_main\_\_':

foo(1,2,3,4)

foo(a=1,b=2,c=3)

foo(1,2,3,4, a=1,b=2,c=3)

foo('a', 1, None, a=1, b='2', c=3)

输出结果如下：

args =  (1, 2, 3, 4)   
kwargs =  {}   
---------------------------------------   
args =  ()   
kwargs =  {'a': 1, 'c': 3, 'b': 2}   
---------------------------------------   
args =  (1, 2, 3, 4)   
kwargs =  {'a': 1, 'c': 3, 'b': 2}   
---------------------------------------   
args =  ('a', 1, None)   
kwargs =  {'a': 1, 'c': 3, 'b': '2'}   
---------------------------------------

可以看到，这两个是python中的可变参数。\*args表示任何多个无名参数，它是一个tuple；\*\*kwargs表示关键字参数，它是一个dict。并且同时使用\*args和\*\*kwargs时，必须\*args参数列要在\*\*kwargs前，像foo(a=1, b='2', c=3, a', 1, None, )这样调用的话，会提示语法错误“SyntaxError: non-keyword arg after keyword arg”。

### 迭代器和生成器

1. 迭代协议
   1. 定义：对象需要提供next方法，她要么返回迭代中的下一项，要么就引起一个StopIteration异常，终止迭代
   2. 可迭代对象：实现了迭代协议的对象
   3. 协议是一种约定，可迭代对象实现迭代器协议，Python的内置工具（如for循环，sum，min，max等函数）使用迭代器协议的访问对象。

举个例子：

在所有语言中我们都可以使用for循环来遍历数组，python的list底层实现是一个数组，所以，我们可以使用for循环来便利list。文件也是先了迭代协议，那么也是可以遍历的。

1. 生成器：对延迟操作提供了支持。延迟操作是指需要的时候才产生结果，而不是立即产生结果。
   1. **特点**：语法上使用yeild，自动实现迭代协议，状态挂起
   2. **注意事项：**生成器只能遍历一次（比如前面使用sum求和了，后面的就无法再使用生成器了）
   3. Python提供了两种不同的方式提供生成器
      1. 生成器函数，常规函数的定义，但是，使用yield语句而不是return语句返回结果。Yeild语句一次返回一个结果，每个结果中间处于挂起函数的状态，下次将会从挂起的地方重新执行。
      2. 生成器表达式：类似与列表推导，但是，生成器返回按需生产结果的对象，而不是一次构建一个列表（这也说明了，其实生成器更简洁）
   4. 看个例子（生成器返回自然数的平方）：

def gensquares(N):

for I in range(N):

yield I \*\* 2

for item in gensquares(5):

print item,

使用普通函数：

def gensquares(N):

res = []

for I in range(N):

res.append(i\*i)

return res

for item in gensquares(5):

print item,

* 1. 生成器表达式

Squares = [x\*\*2 for x in range(5)]

Squares = (x\*\*2 for x in range(5)) # 使用next返回值，利用list先转换成列表

迭代器概述：

迭代器是访问集合内元素的一种方式。迭代对象从集合的第一个元素开始访问，直到所有的元素都被访问一遍后就结束。

但是，迭代器不能回退啊，只能前进迭代。

迭代器不是线程安全的，在多线程环境下，对可变集合使用迭代器是一种危险的操作。但是如果坚持使用不可变的集合，这也不是大问题。

对于原生支持随机访问的数据结构（tuple，list），迭代器和经典for循环的访问相比并无优势，反而丢失了索引值（可以使用内建函数enumerate()这个找回索引值）。但是对于无法随机访问的数据结构(比如set)，迭代器是唯一的访问元素的方式

迭代器的另外一个优点是她不需要求你实现准备好整个迭代过程中所有的元素。迭代仅仅在迭代至某个元素时才计算该元素，而在这之前或之后，元素可以不存在或者被销毁。这个特点使得它可以遍历一些巨大的集合，比如几个g的文件后者斐波那契额数列等等。这些特点被超过年为延迟计算或惰性求值

因为在各种场合下迭代器使用的很广泛，所以在后来的语法中，python加入了for作为迭代器的语法糖

<http://www.cnblogs.com/huxi/archive/2011/07/01/2095931.html>

http://www.cnblogs.com/huxi/archive/2011/07/14/2106863.html

### 装饰器

装饰器是一个很著名的设计模式，经常被用于有切面需求的场景，较为经典的有插入日志、性能测试、事务处理等。装饰器是解决这类问题的绝佳设计，有了装饰器，我们就可以抽离出大量函数中与函数功能本身无关的雷同代码并继续重用。概括的讲，**装饰器的作用就是为已经存在的对象添加额外的功能。**

#### # 装饰器就是把其他函数作为参数的函数

def my\_shiny\_new\_decorator(a\_function\_to\_decorate):

# 在函数里面,装饰器在运行中定义函数: 包装.

# 这个函数将被包装在原始函数的外面,所以可在原始函数之前和之后执行其他代码

def the\_wrapper\_around\_the\_original\_function():

# 把要在原始函数被调用前的代码放在这里

print "Before the function runs"

# 调用原始函数(用括号)

a\_function\_to\_decorate()

# 把要在原始函数调用后的代码放在这里

print "After the function runs"

# 在这里"a\_function\_to\_decorate" 函数永远不会被执行

# 在这里返回刚才包装过的函数

# 在包装函数里包含要在原始函数前后执行的代码.

return the\_wrapper\_around\_the\_original\_function

# 加入你建了个函数,不想修改了

def a\_stand\_alone\_function():

print "I am a stand alone function, don't you dare modify me"

a\_stand\_alone\_function()

#输出: I am a stand alone function, don't you dare modify me

# 现在,你可以装饰它来增加它的功能

# 把它传递给装饰器,它就会返回一个被包装过的函数.

a\_stand\_alone\_function\_decorated = my\_shiny\_new\_decorator(a\_stand\_alone\_function)

a\_stand\_alone\_function\_decorated()

#输出s:

#Before the function runs

#I am a stand alone function, don't you dare modify me

#After the function runs

现在,你或许每次都想用a\_stand\_alone\_function\_decorated代替a\_stand\_alone\_function,很简单,只需要用my\_shiny\_new\_decorator返回的函数重写a\_stand\_alone\_function:

a\_stand\_alone\_function = my\_shiny\_new\_decorator(a\_stand\_alone\_function)

a\_stand\_alone\_function()

#输出:

#Before the function runs

#I am a stand alone function, don't you dare modify me

#After the function runs

# 想到了吗,这就是装饰器干的事!

关于执行顺序：

def bread(func):

def wrapper():

print "</''''''\>"

func()

print "<\\_\_\_\_\_\_/>"

return wrapper

def ingredients(func):

def wrapper():

print "#tomatoes#"

func()

print "~salad~"

return wrapper

def sandwich(food="--ham--"):

print food

sandwich()

#outputs: --ham--

sandwich = bread(ingredients(sandwich))

sandwich()

#outputs:

#</''''''\>

# #tomatoes#

# --ham--

# ~salad~

#<\\_\_\_\_\_\_/>

#### 看看使用python的语法糖：

@bread

@ingredients

def sandwich(food="--ham--"):

print food

sandwich()

#outputs:

#</''''''\>

# #tomatoes#

# --ham--

# ~salad~

#<\\_\_\_\_\_\_/>

改变一下顺序

@ingredients

@bread

def strange\_sandwich(food="--ham--"):

print food

strange\_sandwich()

#outputs:

##tomatoes#

#</''''''\>

# --ham--

#<\\_\_\_\_\_\_/>

# ~salad~

#### 向装饰器传入参数：

# 向装饰器里面传入参数  
def decorator(function):  
 def wrapper(arg1, arg2):  
 print "I got args! Look:", arg1, arg2  
 function(arg1, arg2)  
 return wrapper  
  
@decorator  
def func(first\_name, last\_name):  
 print "My name is", first\_name, last\_name  
  
func("Peter", "Venkman")  
# 输出:  
#I got args! Look: Peter Venkman  
#My name is Peter Venkman

#### 装饰方法：

#在Python里方法和函数几乎一样.唯一的区别就是方法的第一个参数是一个当前对象的(self)  
#也就是说你可以用同样的方式来装饰方法!只要记得把self加进去:  
def decorator(function):  
 def wrapper(self, lie):  
 lie = lie - 3 # 女性福音 :-)  
 return function(self, lie)  
 return wrapper  
  
class Lucy(object):  
  
 def \_\_init\_\_(self):  
 self.age = 32  
  
 @decorator  
 def sayYourAge(self, lie):  
 print "I am %s, what did you think?" % (self.age + lie)  
  
l = Lucy()  
l.sayYourAge(-3)  
#输出: I am 26, what did you think?

通用的方法

def decorator(function\_to\_decorate):

# 包装器接受所有参数

def wrapper(\*args, \*\*kwargs):

print "Do I have args?:"

print args

print kwargs

# 现在把\*args,\*\*kwargs解包

# 如果你不明白什么是解包的话,请查阅:

# http://www.saltycrane.com/blog/2008/01/how-to-use-args-and-kwargs-in-python/

function\_to\_decorate(\*args, \*\*kwargs)

return wrapper

@decorator

def function\_with\_no\_argument():

print "Python is cool, no argument here."

function\_with\_no\_argument()

#输出

#Do I have args?:

#()

#{}

#Python is cool, no argument here.

@decorator

def function\_with\_arguments(a, b, c):

print a, b, c

function\_with\_arguments(1,2,3)

#输出

#Do I have args?:

#(1, 2, 3)

#{}

#1 2 3

@decorator

def function\_with\_named\_arguments(a, b, c, platypus="Why not ?"):

print "Do %s, %s and %s like platypus? %s" %\

(a, b, c, platypus)

function\_with\_named\_arguments("Bill", "Linus", "Steve", platypus="Indeed!")

#输出

#Do I have args ? :

#('Bill', 'Linus', 'Steve')

#{'platypus': 'Indeed!'}

#Do Bill, Linus and Steve like platypus? Indeed!

class Mary(object):

def \_\_init\_\_(self):

self.age = 31

@decorator

def sayYourAge(self, lie=-3): # 可以加入一个默认值

print "I am %s, what did you think ?" % (self.age + lie)

m = Mary()

m.sayYourAge()

#输出

# Do I have args?:

#(<\_\_main\_\_.Mary object at 0xb7d303ac>,)

#{}

#I am 28, what did you think?

#### 把参数传递给装饰器

def decorator\_maker():

print "I make decorators! I am executed only once: "+\

"when you make me create a decorator."

def my\_decorator(func):

print "I am a decorator! I am executed only when you decorate a function."

def wrapped():

print ("I am the wrapper around the decorated function. "

"I am called when you call the decorated function. "

"As the wrapper, I return the RESULT of the decorated function.")

return func()

print "As the decorator, I return the wrapped function."

return wrapped

print "As a decorator maker, I return a decorator"

return my\_decorator

# 让我们建一个装饰器.它只是一个新函数.

new\_decorator = decorator\_maker()

#输出:

#I make decorators! I am executed only once: when you make me create a decorator.

#As a decorator maker, I return a decorator

# 下面来装饰一个函数

def decorated\_function():

print "I am the decorated function."

decorated\_function = new\_decorator(decorated\_function)

#输出:

#I am a decorator! I am executed only when you decorate a function.

#As the decorator, I return the wrapped function

# Let’s call the function:

decorated\_function()

#输出:

#I am the wrapper around the decorated function. I am called when you call the decorated function.

#As the wrapper, I return the RESULT of the decorated function.

#I am the decorated function.

我们来个更简洁的：

def decorated\_function():

print "I am the decorated function."

decorated\_function = decorator\_maker()(decorated\_function)

#输出:

#I make decorators! I am executed only once: when you make me create a decorator.

#As a decorator maker, I return a decorator

#I am a decorator! I am executed only when you decorate a function.

#As the decorator, I return the wrapped function.

# 最后:

decorated\_function()

#输出:

#I am the wrapper around the decorated function. I am called when you call the decorated function.

#As the wrapper, I return the RESULT of the decorated function.

#I am the decorated function.

让我们再次简化一下、

@decorator\_maker()

def decorated\_function():

print "I am the decorated function."

#输出:

#I make decorators! I am executed only once: when you make me create a decorator.

#As a decorator maker, I return a decorator

#I am a decorator! I am executed only when you decorate a function.

#As the decorator, I return the wrapped function.

#最终:

decorated\_function()

#输出:

#I am the wrapper around the decorated function. I am called when you call the decorated function.

#As the wrapper, I return the RESULT of the decorated function.

#I am the decorated function.

那么，到底怎么传输给装饰器呢？

def decorator\_maker\_with\_arguments(decorator\_arg1, decorator\_arg2):

print "I make decorators! And I accept arguments:", decorator\_arg1, decorator\_arg2

def my\_decorator(func):

# 这里传递参数的能力是借鉴了 closures.

# 如果对closures感到困惑可以看看下面这个:

# http://stackoverflow.com/questions/13857/can-you-explain-closures-as-they-relate-to-python

print "I am the decorator. Somehow you passed me arguments:", decorator\_arg1, decorator\_arg2

# 不要忘了装饰器参数和函数参数!

def wrapped(function\_arg1, function\_arg2) :

print ("I am the wrapper around the decorated function.\n"

"I can access all the variables\n"

"\t- from the decorator: {0} {1}\n"

"\t- from the function call: {2} {3}\n"

"Then I can pass them to the decorated function"

.format(decorator\_arg1, decorator\_arg2,

function\_arg1, function\_arg2))

return func(function\_arg1, function\_arg2)

return wrapped

return my\_decorator

@decorator\_maker\_with\_arguments("Leonard", "Sheldon")

def decorated\_function\_with\_arguments(function\_arg1, function\_arg2):

print ("I am the decorated function and only knows about my arguments: {0}"

" {1}".format(function\_arg1, function\_arg2))

decorated\_function\_with\_arguments("Rajesh", "Howard")

#输出:

#I make decorators! And I accept arguments: Leonard Sheldon

#I am the decorator. Somehow you passed me arguments: Leonard Sheldon

#I am the wrapper around the decorated function.

#I can access all the variables

# - from the decorator: Leonard Sheldon

# - from the function call: Rajesh Howard

#Then I can pass them to the decorated function

#I am the decorated function and only knows about my arguments: Rajesh Howard

我们换成变量来试试

c1 = "Penny"

c2 = "Leslie"

@decorator\_maker\_with\_arguments("Leonard", c1)

def decorated\_function\_with\_arguments(function\_arg1, function\_arg2):

print ("I am the decorated function and only knows about my arguments:"

" {0} {1}".format(function\_arg1, function\_arg2))

decorated\_function\_with\_arguments(c2, "Howard")

#输出:

#I make decorators! And I accept arguments: Leonard Penny

#I am the decorator. Somehow you passed me arguments: Leonard Penny

#I am the wrapper around the decorated function.

#I can access all the variables

# - from the decorator: Leonard Penny

# - from the function call: Leslie Howard

#Then I can pass them to the decorated function

#I am the decorated function and only knows about my arguments: Leslie Howard

### 鸭子类型

在程序设计中，鸭子类型（英语：duck typing）是动态类型的一种风格。在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定。比如在python中，有很多file-like的东西，比如StringIO,GzipFile,socket。它们有很多相同的方法，我们把它们当作文件使用。

又比如list.extend()方法中,我们并不关心它的参数是不是list,只要它是可迭代的,所以它的参数可以是list/tuple/dict/字符串/生成器等.

鸭子类型在动态语言中经常使用，非常灵活，使得python不想java那样专门去弄一大堆的设计模式。

### 重载

先看重载是为了干什么？主要是为了解决两个问题：

1. 可变参数类型
2. 可变参数个数

另外，一个基本的设计原则是，仅仅当两个函数除了参数类型和参数个数不同以外，其功能是完全相同的，此时才使用函数重载，如果两个函数的功能其实不同，那么不应当使用重载，而应当使用一个名字不同的函数。

那么在Python中：

那么对于情况 1 ，函数功能相同，但是参数类型不同，python 如何处理？答案是根本不需要处理，因为 python 可以接受任何类型的参数，如果函数的功能相同，那么不同的参数类型在 python 中很可能是相同的代码，没有必要做成两个不同函数。

那么对于情况 2 ，函数功能相同，但参数个数不同，python 如何处理？大家知道，答案就是缺省参数。对那些缺少的参数设定为缺省参数即可解决问题。因为你假设函数功能相同，那么那些缺少的参数终归是需要用的。

另外补充一点：在动态语言中，有鸭子类型，如果走起路来像鸭子，叫起来也像鸭子，那么他就是鸭子。一个对象的特征不是由它的类型决定，而是通过对象中的方法决定，所以函数重载在动态语言中就显得没有意义了，因为函数可以通过鸭子类型来处理不同类型的对象了。鸭子类型也是多态性的一种表现。

### 新式类和旧式类

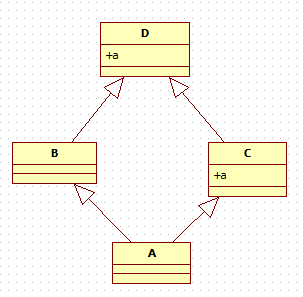
python在2.2版本中引入了descriptor功能，也正是基于这个功能实现了新式类(new-styel class)的对象模型，同时解决了之前版本中经典类(classic class)系统中出现的多重继承中的MRO(Method Resolution Order)的问题，同时引入了一些新的概念，比如classmethod, staticmethod, super,Property等，这些新功能都是基于descriptor而实现的。总而言之，通过学习descriptor可以更多地了解python的运行机制。我在这也大概写一个汇总，写一下对这些东西的理解。欢迎大家讨论。

创建方法：

#新式类  
**class** C(object):  
    **pass**  
#经典类  
**class** B:  
    **pass**

**MRO(Method Resolution Order， 方法解析顺序)**

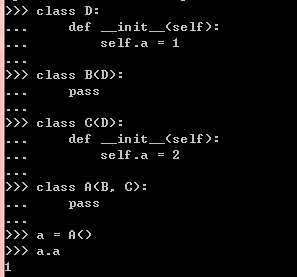
对于下图的多继承关系：



b = A()，当调用b.a的时候会发生什么事呢？

在经典对象模型中，方法和属性的查找链是按照从左到右，深度优先的方式进行查找。所以当A的实例b要使用属性a时，它的查找顺序为:A->B->D->C->A，这样做就会忽略类C的定义a，而先找到的基类D的属性a，这是一个bug，这个问题在新式类中得到修复，新的对象模型采用的是从左到右，广度优先的方式进行查找，所以查找顺序为A->B->C->D，可以正确的返回类C的属性a。

经典类：



新式类：

