元类

类是一种对象，使用关键字class创建一个类的时候，会在内存中创建一个对象，名字即为类名，这个对象叫做类对象，拥有创建对象的能力，但是本质仍为一个对象

动态地创建类

可以在函数中创建类，使用class

内建函数type，可以知道一个对象的类型是什么

>>> print type(1) #数值的类型

>>> print type("1") #字符串的类型

>>> print type(ObjectCreator()) #实例对象的类型

<class '\_\_main\_\_.ObjectCreator'>

>>> print type(ObjectCreator) #类的类型

<type 'type'>

使用type动态的创建类

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）)

MyDogClass = type('MyDog', (), {})

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

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

等于

class Foo(object):

bar = True

使用type创建带有方法的类

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。

def echo\_bar(self): #定义了一个普通的函数

print(self.bar)

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

hasattr(Foo, 'echo\_bar') #判断Foo类中，是否有echo\_bar这个属性

False

hasattr(FooChild, 'echo\_bar') #判断FooChild类中，是否有echo\_bar这个属性

True

元类就是用来创建类的“东西”。用来创建这些类（对象）的，元类就是类的类

MyClass = MetaClass() #使用元类创建出一个对象，这个对象称为“类”

MyObject = MyClass() #使用“类”来创建出实例对象

Python中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。

\_\_metaclass\_\_属性

在定义一个类的时候为其添加\_\_metaclass\_\_属性，Python就会用元类来创建类

class Foo(Bar):

pass

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。

自定义元类

动态语言

动态语言是指运行时可以改变结构的语言

动态语言：可以在运行的过程中，修改代码

静态语言：编译时已经确定好代码，运行过程中不能修改

运行过程中给对象添加属性

就是动态的给实例绑定属性,定义的类里面是没有的,但是对象里有新加的属性,并且只有这个对象有,其他对象不会同步

运行的过程中给类添加属性

如果需要给所有实例加上新的属性,就直接给类添加属性,直接赋值

运行过程中给类添加方法

给类添加方法，是使用类名.方法名 = xxxx，那么给对象添加一个方法也是类似的,对象.方法名 = xxxx

给实例对象添加实例方法: 实例对象.方法名=types.MethodType(方法名,实例对象)

给类添加类方法: 类名.类方法名=类方法名

给类添加静态方法: 类名.静态方法名=静态方法名

运行过程中删除属性、方法

删除的方法:

del 对象.属性名

delattr(对象, "属性名")

\_\_slots\_\_变量，限制实例的属性

class Person(object):

\_\_slots\_\_ = ("name", "age")

\_\_slots\_\_定义的属性仅对当前类实例起作用，对继承的子类是不起作用的

生成器

一边循环一边计算的机制，称为生成器：generator

元素可以按照某种算法推算出来，这样就不必创建完整的list，从而节省大量的空间

在一个 generator function 中，如果没有 return，则默认执行至函数完毕，如果在执行过程中 return，则直接抛出 StopIteration 终止迭代。

创建生成器

方法1：把一个列表生成式的 [ ] 改成 ( )，打印生成器使用for…in…

方法2：列表无法实现的时候可以使用函数实现，将函数变为生成器，即将print变为yield。循环输出过程中会不断调用yield，不断中断

迭代器Iterator

可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator。

迭代是访问集合元素的方式，迭代器可以记住遍历位置的对象，只会向前不会后退

可迭代对象

以直接作用于 for 循环的数据类型有以下几种：

一类是集合数据类型，如 list 、 tuple 、 dict 、 set 、 str 等；

一类是 generator ，包括生成器和带 yield 的generator function。

这些可以直接作用于 for 循环的对象统称为可迭代对象：Iterable 。

判断是否可以迭代：使用isinstance()判断一个对象是否是Iterable

from collections import Iterable  
print(isinstance((x for x in range(10)), Iterable))  
print(isinstance([],Iterable))

iter()函数

生成器都是Iterator对象，但是list 、dict 、 str虽然是可迭代对象，却不是迭代器，可以使用iter()函数变成迭代器

from collections import Iterator  
print(isinstance([],Iterator))  
print(isinstance(iter([]),Iterator))

 凡是可作用于 for 循环的对象都是 Iterable 类型；

 凡是可作用于 next() 函数的对象都是 Iterator 类型

闭包

在函数内部再定义一个函数，并且这个函数用到了外边函数的变量，那么将这个函数以及用到的一些变量称之为闭包

内部函数对外部函数作用域里变量的引用（非全局变量），则称内部函数为闭包函数。闭包函数所引用的外部定义的变量称为自由变量。

def line\_conf(a, b):  
 def line(x):  
 return a\*x + b  
 return line  
line1 = line\_conf(1, 1)  
line2 = line\_conf(4, 5)  
print(line1(5))  
print(line2(5))

装饰器

装饰器本质上是一个函数，它可以让其他函数在不需要做任何代码变动的前提下，增加额外的功能，装饰器的返回值也是一个函数对象（函数的指针）。

装饰器函数的外部函数传入需要装饰的函数名字，返回修饰后的函数名字，内层函数（闭包）修饰函数

函数装饰器

1. 函数的函数装饰器

语法 @外部函数名 相当于 需要修饰的函数=外部函数（需要修饰的函数）

即为需要修饰的函数修饰并返回

1. 类方法的函数装饰器

类装饰器

\_\_call\_\_()可以将一个类实例变成一个可调用对象

类装饰器

装饰器链

高阶函数

map()

map()函数接收两个参数，一个是函数，一个是Iterable，map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回。

list(map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9]))

['1', '2', '3', '4', '5', '6', '7', '8', '9']

reduce()

reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算

reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)

#把str转换为int的函数：

from functools import reduce  
DIGITS = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}  
def char2num(s):  
 return DIGITS[s]  
def str2int(s):  
return reduce(lambda x, y: x \* 10 + y, map(char2num, s))  
  
filter()

Python内建的filter()函数用于过滤序列。filter()也接收一个函数和一个序列，和map()不同的是，filter()把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。

def odd\_p(n):  
 return n%2==1  
print(list(filter(odd\_p,[1,2,3,4,5,6])))