# 浅拷贝和深拷贝

对于没有嵌套子对象的不可改变对象，例如：整数对象，字符串和元组对象等，不会进行拷贝，不会创建另一个对象  
浅拷贝：使用[:],list,copy,dict,set这几种方法  
深拷贝：deepcopy  
  
浅拷贝：  
L = [1,2,3,4,[56,1]]  
L1 = L[:]  
print('原始列表：',id(L))  
print('拷贝的列表：',id(L1))  
print('拷贝前的列表中的元素：',id(L[0]))  
print('拷贝后的列表中的元素：',id(L[0]))  
  
深拷贝：  
import copy  
L = [1,2,3,4,[56,1]]  
L1 = copy.deepcopy(L)  
print('原始列表：',id(L))  
print('拷贝的列表：',id(L1))  
print('拷贝前的列表中的元素：',id(L[4][1]))  
print('拷贝后的列表中的元素：',id(L1[4][0]))  
运行结果：  
原始列表： 51246184  
拷贝的列表： 51360264  
拷贝前的列表中的元素： 1690355632  
拷贝后的列表中的元素： 1690356512

# Python三大特性之封装

1：封装数据：主要原因是：保护私隐，明确区分内外。将数据隐藏起来这不是目的。隐藏起来然后对外提供操作该数据的接口，然后我  
们可以在接口附加上对该数据操作的限制，以此完成对数据属性操作的严格控制。  
2：封装方法：目的是隔离复杂度  
封装方法举例：  
1. 你的身体没有一处不体现着封装的概念：你的身体把膀胱尿道等等这些尿的功能隐藏了起来，然后为你提供一个尿的接口就可以了（  
接口就是你的。。。，），你总不能把膀胱挂在身体外面，上厕所的时候就跟别人炫耀：hi，man，你瞅我的膀胱，看看我是怎么尿的。  
2. 电视机本身是一个黑盒子，隐藏了所有细节，但是一定会对外提供了一堆按钮，这些按钮也正是接口的概念，所以说，封装并不是单  
纯意义的隐藏！！！  
3. 快门就是傻瓜相机为傻瓜们提供的方法，该方法将内部复杂的照相功能都隐藏起来了  
提示：在编程语言里，对外提供的接口（接口可理解为了一个入口），可以是函数，称为接口函数，这与接口的概念还不一样，接口代表  
一组接口函数的集合体。  
  
class Student(object):  
 def \_\_init\_\_(self, age):  
 self.\_\_age = age  
 def get\_age(self):  
 return self.\_\_age  
 def set\_age(self, age):  
 if 0 <= age <= 100:  
 self.\_\_age = age  
 else:   
 raise ValueError("年龄输入不规范！！！")  
s = Student(18)  
print(s.get\_age())  
s.set\_age(221)  
print(s.get\_age())  
运行结果：  
D:\Pythonproject\venv\Scripts\python.exe D:/Pythonproject/venv/Include/封装.py  
Traceback (most recent call last):  
18  
 File "D:/Pythonproject/venv/Include/封装.py", line 13, in <module>  
 s.set\_age(221)  
 File "D:/Pythonproject/venv/Include/封装.py", line 10, in set\_age  
 raise ValueError("年龄输入不规范！！！")  
ValueError: 年龄输入不规范！！！  
  
class Student(object):  
 def \_\_init\_\_(self, age):  
 self.\_\_age = age  
 def get\_age(self):  
 return self.\_\_age  
 def set\_age(self, age):  
 if 0<= age <= 100:  
 self.\_\_age = age  
 else:  
 raise ValueError("年龄输入不规范！！！")  
s = Student(18)  
print(s.get\_age())  
s.set\_age(22)  
print(s.get\_age())  
运行结果：  
D:\Pythonproject\venv\Scripts\python.exe D:/Pythonproject/venv/Include/封装.py  
18  
22  
上面的代码就实现了一个简单的封装  
  
class Student(object):  
 def \_\_init\_\_(self, age):  
 self.\_\_age = age  
 def get\_age(self):  
 return self.\_\_age  
 def set\_age(self, age):  
 if 0<= age <= 100:  
 self.\_\_age = age  
 else:  
 raise ValueError("年龄输入不规范！！！")  
s = Student(18)  
print(s.get\_age())  
s.set\_age(22)  
print(s.get\_age())

# Python三大特性之继承

　　面向对象编程 (OOP) 语言的一个主要功能就是“继承”。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原  
来的类的情况下对这些功能进行扩展。  
　　通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”，继承的过程，就是从一般到特殊的过程。在  
某些 OOP 语言中，一个子类可以继承多个基类。但是一般情况下，一个子类只能有一个基类，要实现多重继承，可以通过多级继承来实现。  
　　继承概念的实现方式主要有2类：实现继承、接口继承。  
实现继承是指使用基类的属性和方法而无需额外编码的能力。  
接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力(子类重构爹类方法)。  
　　在考虑使用继承时，有一点需要注意，那就是两个类之间的关系应该是“属于”关系。例如，Employee 是一个人，Manager 也是一  
个人，因此这两个类都可以继承 Person 类。但是 Leg 类却不能继承 Person 类，因为腿并不是一个人。  
　　OO开发范式大致为：划分对象→抽象类→将类组织成为层次化结构(继承和合成) →用类与实例进行设计和实现几个阶段。  
基类，派生类就是(抽象类)  
  
class Animal(object): 父类超类 继承方法就是在类名后的括号中写入父类的类名  
 def eat(self):  
 print('吃饭')  
 def drink(self):  
 print('喝水')  
class Dog(Animal): 派生类  
 def swim(self):  
 print("游泳")  
class Bird(Animal):  
 def fly(self):  
 print('飞翔')  
  
d = Dog()  
d.eat()  
d.swim()  
d.drink()  
Bird().fly()  
  
重写  
注意:父类的方法在子类是可以调用的，可以通过super().xxx()来调用  
  
class Animal(object): 父类超类 继承方法就是在类名后的括号中写入父类的类名  
 def eat(self):  
 self.\_\_age = 18  
 print('吃饭')  
 def drink(self):  
 print('喝水')  
class Dog(Animal): 派生类  
 def eat(self):  
 super().eat() 表示调用父类中的方法！!!  
 print('狗狗吃饭！！！')  
 def swim(self):  
 print("游泳")  
class Bird(Animal):  
 def fly(self):  
 print('飞翔')  
d = Dog()  
d.eat()  
print(d.\_Animal\_\_age)

# Python三大特性之多态

1、只关心对象的实例方法是否同名，不关心对象所属的类型；  
2、对象所属的类之间，继承关系可有可无；  
3、多态的好处可以增加代码的外部调用灵活度，让代码更加通用，兼容性比较强；  
4、多态是调用方法的技巧，不会影响到类的内部设计。  
  
Python中是动态语言  
再调用函数的时候不会检查参数的类型,所以和静态语言(java)的多态是有区别的  
实现多态必须要满足三个条件：  
1，重写  
2，继承  
3，父类类型的变量引用或子类类型的实例对象  
鸭子类型：当看到一只鸟，走起开像鸭子，叫起来像鸭子，游泳起来像鸭子，不关心到底是不是鸭子。只是关心他的方法  
  
class Person(object):  
 def \_\_init\_\_(self, name, gender):  
 self.name = name  
 self.gender = gender  
 def whoAmI(self):  
 return 'I am a Person, my name is %s' % self.name  
  
class Student(Person):  
 def \_\_init\_\_(self, name, gender, score):  
 super(Student, self).\_\_init\_\_(name, gender)  
 self.score = score  
  
 def whoAmI(self):  
 return 'I am a Student, my name is %s' % self.name  
  
class Teacher(Person):  
 def \_\_init\_\_(self, name, gender, course):  
 super(Teacher, self).\_\_init\_\_(name, gender)  
 self.course = course  
  
 def whoAmI(self):  
 return 'I am a Teacher, my name is %s' % self.name  
t = Teacher('张三老师','男','99')  
print('course:',t.course)  
print('gender:',t.gender)  
print('name:',t.name)  
print(t.whoAmI())

# MRO

MRO方法解析顺序Method Resoution Order它指的是对于一颗继承树，当调用最底层类对象所对应的方法时，Python解析器在类继承  
树上搜索方法的顺序  
  
class A(object):  
 def f(self):  
 print("A.f")  
  
class B(A):  
 def f(self):  
 print("B.f")  
  
class C(A):  
 def f(self):  
 print("C.f")  
  
class D(B,C):  
 def f(self):  
 print("D.f")  
print(print(D.mro()))  
运行结果：[<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]  
None  
  
  
class A(object):  
 def f(self):  
 print("A.f")  
  
class B(A):  
 def f(self):  
 print("B.f")  
  
class C(A):  
 def f(self):  
 print("C.f")  
  
class D(B,C):  
 def f(self):  
 super(B, self).f() 了解RMO就是为这个做基础，它们的顺序以此为 D B C A object super传入的第一个参数就是传入参数后面的后面的哪一个  
 print("D.f")  
D().f()  
运行结果：  
C.f  
D.f  
  
如果想调用指定父类重写的方法可以给super传入两个形参  
第一个实参a\_type是一个类对象。第二个obj是一个实例对象  
  
class A(object):  
 def f(self):  
 print("A.f")  
  
class B(A):  
 def f(self):  
 print("B.f")  
  
class C(A):  
 def f(self):  
 print("C.f")  
  
class D(B,C):  
 def f(self):  
 super(B, self).f() 了解RMO就是为这个做基础，它们的顺序以此为 D B C A object super传入的第一个参数就是传入参数后面的后面的哪一个  
 print("D.f")  
D().f()  
m = 1

# Python之获取对象信息

issubclass()

用于判断类对象和类对象之间的关系 第一个实参是类对象，第二个实参是类对象或者是元组，如果第一个实参是第二个实  
参子类则返回True 如果第二个实参是第二个实参元组中任意一个子类则返回True  
  
isinstance()

用于判断实例对象和实例对象之间的关系 第一个参数是实例对象就是D（），第二个参数是类名或者是元组，如果第一个实  
参是第二个实参的实例对象或者是子类的实参则返回True  
  
  
class A(object):  
 pass  
  
class B(A):  
 pass  
  
class C(A):  
 pass  
  
class D(B,C):  
 pass  
class E():  
 pass  
s = (A,B,C,E)  
print(issubclass(D,A)) 用来判断D是否是A的子类  
print(issubclass(D,B)) 用来判断D是否是B的子类  
print(issubclass(D,s)) 用来判断D是否是元组s的子类 注意：这里判断的必须是元组而不是列表  
  
class A(object):  
 pass  
  
class B(A):  
 pass  
  
class C(A):  
 pass  
  
class D(B,C):  
 pass  
class E():  
 pass  
print(isinstance(D(),D)) True  
print(isinstance(D(),E)) False  
print(isinstance(D(),C)) True  
print(isinstance(D(),object)) True

type（）  
内置函数type（）  
print(type(int)) <class 'type'> 类对象  
print(type(str)) <class 'type'> 类对象  
print(type([])) <class 'list'> 实例对象  
print(type(())) <class 'tuple'>  
print(type(dict())) <class 'dict'> 实例对象  
print(type(156)) <class 'int'>  
print(type('')) <class 'str'>  
def fun():  
 pass  
print(type(fun)) <class 'function'>  
print(type(print)) <class 'builtin\_function\_or\_method'>  
使用通过运算符'=='来判断是否是同一种类型   
注意：如果不是基本类型就不能使用==进行判断，但是可以使用types模块中定义的变量  
import types  
print(type(print) == types.BuiltinFunctionType)  
运行结果：  
True  
  
  
dir（）

可以访问到属性和方法的列表，包括从父类中继承的属性和方法

Pthon之特殊属性和特殊方法

以双下划线开头的属性和方法被称为特殊方法  
在定义类的时候会重写一个或者多个属性和方法  
特殊属性\_\_dict\_\_ 返回的是字典 key为属性或方法  
 class Myclass(object):  
 ca = 'ca'  
 def \_\_init\_\_(self):  
 pass  
 def im(self):  
 pass  
 @classmethod  
 def cm(self):  
 pass  
 @staticmethod  
 def sm():  
 pass  
 print(Myclass.\_\_dict\_\_)  
 {'\_\_module\_\_': '\_\_main\_\_', 'ca': 'ca', '\_\_init\_\_': <function Myclass.\_\_init\_\_ at 0x012A57C0>,  
 'im': <function Myclass.im at 0x012A5778>, 'cm': <classmethod object at 0x00ADE6B8>,  
 'sm': <staticmethod object at 0x02E5B310>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Myclass' objects>,  
 '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Myclass' objects>, '\_\_doc\_\_': None}  
  
  
反射：

以字符串的方式来操作（包括：增删改查）对象的属性和方法  
用于反射的内置函数有‘四个’  
 hasattr(object,name) 用于指定object中是否有name这个属性,或者是方法  
 class Myclass(object):  
 ca = 'ca'  
 def \_\_init\_\_(self):  
 pass  
 def im(self):  
 pass  
 @classmethod  
 def cm(self):  
 pass  
 @staticmethod  
 def sm():  
 pass  
 print(hasattr(Myclass,'ca')) True  
 print(hasattr(Myclass,'sm()')) True 注意：第二个传入的是不管是字符串属性还是方法都是名字而不是sm()  
  
  
getattr(object, name,[default]) 如果不存在指定的属性和方法就会抛出异常，如果指定了这个默认值，就会返回这个默认值  
getattr：用于获取指定name的方法或者属性，如果没有找到指定的方法或者属性那就返回default中的默认设置，如果没有默认设置那就抛出错误  
 class Myclass(object):  
 ca = 'ca'  
 def \_\_init\_\_(self):  
 pass  
 def im(self):  
 pass  
 @classmethod  
 def cm(self):  
 pass  
 @staticmethod  
 def sm():  
 print('执行了静态方法sm()')  
 getattr(Myclass,'sm','没有你指定的方法！！！')() 注意这只是获取到了这个方法但不是执行了这个方法,所以在后面加一个小括号就等于访问了这个方法  
 print(getattr(Myclass,'sm','没有你指定的方法！！！'))  
 运行结果：  
 执行了静态方法sm()  
 <function Myclass.sm at 0x037956E8>  
  
  
  
setatter（object,name,value） ：用于添加或者修改指定的属性和方法  
setatter（object,name,value）等价于object.name = value  
第一个参数表示要使用的类  
第二个参数用来填写要指定的要修改的属性和方法的名字  
第三个参数用来填写要修改后的值  
 class Myclass(object):  
 ca = 'ca'  
 def \_\_init\_\_(self):  
 pass  
 def im(self):  
 pass  
 @classmethod  
 def cm(self):  
 pass  
 @staticmethod  
 def sm():  
 print('执行了静态方法sm()')  
 setattr(Myclass,'name' ,1) 因为Mycalss中不存在name的属性，那这个操作就相当于在给Myclass类新添加一个属性name并且赋值为1  
 Myclass.name = 2 对通过setatter方法绑定的属性进行修改值为2  
 print(Myclass.name) 打印name的值：2  
 setattr(Myclass,'name',3) 再次通过setattr对name的值进行修改  
 print(Myclass.name) 打印name的值：3  
  
  
delettr（object,name）:用于删除指定的属性和方法  
delettr（object,name）等价于del object.name  
object:类名  
name：表示要删除的方法或者属性的名字  
  
  
注意：只有在不知道对象的名称的时候才会查询  
  
  
len（）

用于返回长度   
内置函数len的实参都是内置对象的实例对象list,str,dict  
如果不是内置对象就会抛出错误,所以要进行实现len方法，所以必须实现特殊方法\_\_len\_\_()这个特殊方法  
class MyClass(object):  
 def \_\_len\_\_(self):  
 return len(str(MyClass))  
print(len(MyClass())) 注意：在这里必须使用Myclass()而不是使用Mycalss  
  
  
类对象的  
\_\_iter\_\_（）和\_\_next\_\_()  
for-in语句不能使用自定义类对象的实例对象，必须要实现\_\_iter\_\_和\_\_next\_\_这两个方法  
首先会调用特殊方法\_\_iter\_\_发返回一个可迭代对象  
然后不断调用可迭代对象的特殊方法\_\_next\_\_（）直到遇到StopIteration()退出循环  
class MyClass(object):  
 name = '张三'  
 age = 17  
 genter = '男'  
 def \_\_init\_\_(self):  
 self.data = 0  
 def \_\_len\_\_(self):  
 return len(str(MyClass))  
 def \_\_iter\_\_(self):  
 return dir(MyClass)  
 def \_\_next\_\_(self): 返回下一个可迭代的值  
 if self.data > len(dir(MyClass)):  
 raise StopIteration()  
 else:  
 self.data += 1  
 return dir(MyClass)[self.data]  
for item in MyClass():  
 print(item)  
代码运行如下：  
Traceback (most recent call last):  
 File "D:/Pythonproject/venv/Include/Python高级/获取对象的信息.py", line 198, in <module>  
 for item in MyClass():  
TypeError: iter() returned non-iterator of type 'list'  
  
  
类对象的特殊方法之

\_\_add\_\_()和\_\_radd\_\_()  
注意：在标准算数运算符是不能用在类对象上的，必须实现的特殊方法，对于  
‘+’就是\_\_add\_\_()和\_\_radd\_\_() 注意运算符左边实现的运算符是\_\_add\_\_（）右边实现的是\_\_radd\_\_()  
‘-’就是\_sub\_\_()和\_\_rsub\_\_()  
'\*'对应的就是\_\_mul\_\_()和\_\_rmul\_\_()  
'/'对应的就是\_\_truediv\_\_()和\_\_rtruediv\_\_()  
'//'对应的是\_\_floordiv\_\_()和\_\_rfloordiv\_\_()  
之所以可以使用加法或者乘法来操作字符串列表，是因为他们都实现了这些个方法  
class MyClass(object):  
 def \_\_add\_\_(self, other):  
 print('MyClass特殊方法\_\_add\_\_被调用')  
 return NotImplemented  
 def \_\_radd\_\_(self, other):  
 print('MyClass特殊方法\_\_radd\_\_被调用')  
 return 'xxx'  
class MyClass1(object):  
 def \_\_add\_\_(self, other):  
 print('MyClass1特殊方法\_\_add\_\_被调用')  
 return 'yyy'  
 def \_\_radd\_\_(self, other):  
 print('MyClass1特殊方法\_\_add\_\_被调用')  
 return 'yyy'  
  
m1 = MyClass()  
m2 = MyClass1()  
print(m1+m2)  
print(m2+m1) 注意：当没m2里面有\_\_add方法时直接去里面找，如果没有去radd方法里面去找  
  
  
>>> class MyClass(object):  
... pass  
...  
>>> mc = MyClass()  
>>> mc  
<\_\_main\_\_.MyClass object at 0x02C252C8>  
>>> print(mc)  
<\_\_main\_\_.MyClass object at 0x02C252C8>  
>>> str(mc)  
'<\_\_main\_\_.MyClass object at 0x02C252C8>'  
>>> repr(mc)  
'<\_\_main\_\_.MyClass object at 0x02C252C8>'  
  
  
特殊方法\_\_new\_\_（）  
python解析器主要处理过程两大部  
第一步：使用\_\_new\_\_（）进行创建  
 首先会查找是否实现了特殊方法\_\_new\_\_（）找不到去父类查找，直到到object  
第二部：使用\_\_init\_\_（）进行初始化  
class MyClass(object):  
 def \_\_new\_\_(cls, \*args, \*\*kwargs):  
 print("父类的\_\_new\_\_()被调用，其形参cls对应的实参di:%s" % id(cls))  
 obj = super().\_\_new\_\_(cls)  
 print("创建的实例对象的id：%s" % id(obj))  
 return obj  
class Child(MyClass):  
 def \_\_init\_\_(self, name):  
 print('创建的初始化self的id:%s' % id(self))  
 self.name = name  
print("父类的id:%s" % id(MyClass))  
print("子类的id：%s" % id(Child))  
child = Child('张三')  
print("创建的子类对象的id：%s" % id(child))

对象的引用计数

通常情况下，开发人员无序关心内存的分配和释放，当创建一个对象的时候系统会自动分配一块内存，当存储该对象的对象。当对象不在被使用  
 时候，系统会进行系统进行垃圾回收  
 为了确保使用中的对象不会被销毁，python使用引用计数和计算内存每个对象被引用的次数，当对象引用计数为0的时候，它才会被销毁  
  
对象的引用计数加1的情形  
1，对象赋值给变量  
2，引用对象的变量赋值给另一个变量  
3，对象作为容器（列表，元组，字典，集合）中的元素  
4，对象作为函数引用参数时的实参  
对象的引用计数减1的情形  
1，对象离开他的引用域，例如：对象所在的函数执行完毕  
2，对象引用被现式销毁  
3，引用对象的变量被赋予新值  
4，从容其中删除对象，或对象所在的容器被销毁  
import sys  
class MyClass(object):  
 pass  
def doString(m):  
 pass  
m = MyClass() 现在之后变量m在引用所以他是1  
print('当前引用计数为：',sys.getrefcount(m))  
n = m 引用计数再次加1变为2  
print('当前引用计数为：',sys.getrefcount(m))  
l = {1,2,m} 引用计数再次加1变为3  
print('当前引用计数为：',sys.getrefcount(m))  
doString(m) 引用计数再次加1变为4 注意：在函数执行过程中，没有引用是不会加1的，当函数结束之后引用计数会-1变为3  
print('当前引用计数为：',sys.getrefcount(m))  
 用来判断引用计数的个数  
  
  
类对象的特殊方法\_\_del\_\_()  
系统会自动销毁不需要的对象以释放内存，因此。当对象被销毁时候通常不需要手动进行执行清理工作  
但是，当我们用自己创建的资源时，就需要之情一些额外的清理工作，例如自己定义了类对象打开并写入一些数据，就需要在实例对象之前关闭这些  
问文件  
当引用为0时并不会立刻被销毁，所以合适被调用也是不确定的  
class MyClass(object):  
 def \_\_del\_\_(self):  
 print('del被调用')  
mc = MyClass()  
  
  
类对象的特殊方法\_\_getattr\_\_（）  
如果访问指定的属性和方法不存在的时候就会抛出错误，如果定义\_\_getattr\_\_（）就会被调用  
class Myclass(object):  
 def \_\_getattr\_\_(self, item):  
 print("您调用的属性或者方法不存在！")  
mc = Myclass()  
print(mc.date) 'Myclass' object has no attribute 'date  
运行结果：  
您调用的属性或者方法不存在！  
代码：  
class Myclass(object):  
 def \_\_getattr\_\_(self, item): 注意：这里的item表示访问不存在的值的名称  
 if item == 'date':  
 return 18  
 else:  
 print("您调用的属性或者方法不存在！")  
  
mc = Myclass()  
print(mc.date) 'Myclass' object has no attribute 'date'  
运行结果：18  
  
  
特殊方法  
\_\_getitem\_\_（self,key）  
执行obj[key]  
\_\_setitem\_\_(self, key, value)  
obj[key] = value  
\_\_delitem\_\_(self, key)  
del obj[key]  
对于自定义实例对象，在默认情况下是不能使用中括号那样来操作对象的(像列表或者字典)  
class Myclass(object):  
 def \_\_init\_\_(self):  
 self.data = {}  
 def \_\_getitem\_\_(self, item): 注意：这里的item表示访问不存在的值的名称  
 return self.data[key]  
 def \_\_setitem\_\_(self, key, value):  
 self.data[key] = value  
 def \_\_delitem\_\_(self, key):  
 del self.data[key]  
mc = Myclass()  
mc['1'] = 1  
print(mc.data['1'])  
  
  
\_\_call\_\_()  
class Myclass(object):  
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 print(args,kwargs)  
mc = Myclass()  
mc() 像调用函数一样调用特殊方法\_\_call\_\_() 自动调用  
mc(1,2,x = 7)  
  
callable(函数名)查看函数否实是可调用的 注意：一般函数都是可调用的  
代码：  
class Myclass(object):  
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 print(args,kwargs)  
mc = Myclass()  
mc() 像调用函数一样调用特殊方法\_\_call\_\_() 自动调用  
mc(1,2,x = 7)  
print(callable(mc))  
运行结果：  
() {}  
(1, 2) {'x': 7}  
True  
  
  
\_\_doc\_\_()

用于表示类对象的文档字符串  
class Myclass(object):  
 ""这是一个Myclass类""  
 def \_\_call\_\_(self, \*args, \*\*kwargs):  
 ""\_\_call\_\_的文档字符串""  
 print(args,kwargs)  
 def \_\_init\_\_(self):  
 print(super().\_\_doc\_\_)  
print(Myclass.\_\_doc\_\_)  
Myclass()  
  
  
Python是动态语言，所以在动态地绑定属性和方法

\_\_slots\_\_

class Myclass(object):  
 \_\_slots\_\_ = ("attr1", "do\_sth1")  
mc = Myclass()  
mc.attr1 = 18  
def do\_sth1(self):  
 print("do\_sth1被调用了")  
  
from types import MethodType  
mc.do\_sth1 = MethodType(do\_sth1, mc)  
mc.do\_sth1()  
总结：如果对象动态属性进行限制，那就是用\_\_slots\_\_（）  
  
  
默认情况下，在访问属性时，默认是使用特殊属性\_\_dict\_\_['x']来进行访问的  
代码：  
class Myclass(object):  
 \_\_slots\_\_ = ("attr1", "do\_sth1")  
mc = Myclass()  
mc.attr1 = 18  
def do\_sth1(self):  
 print("do\_sth1被调用了")  
  
from types import MethodType  
mc.do\_sth1 = MethodType(do\_sth1, mc)  
mc.do\_sth1()  
print(Myclass().\_\_dict\_\_)  
运行结果：AttributeError: 'Myclass' object has no attribute '\_\_dict\_\_'  
 当定义特殊属性当定义特殊属性\_\_slots\_\_之后系统就不会访问\_\_dict\_\_可而是为每个对象创建一个描述器之后系统就不会访问\_\_dict\_\_可而是为  
每个对象创建一个描述器，访问属性时就会直接调用这个描述器,因此定义特殊属性当定义特殊属性\_\_slots\_\_之后系统就不会访问\_\_dict\_\_可而是  
为每个对象创建一个描述器，所以时\_\_slots\_\_可以减少内存消耗，和提升访问速度，只对实例对象起作用，对其子类是不起作用的，但是如果子类也  
继承了\_\_slote\_\_。那么子类也可以绑定父类定义的值  
  
class Myclass(object):  
 \_\_slots\_\_ = ("attr1", "do\_sth1")  
mc = Myclass()  
mc.attr1 = 18  
def do\_sth1(self):  
 print("do\_sth1被调用了")  
  
from types import MethodType  
mc.do\_sth1 = MethodType(do\_sth1, mc)  
mc.do\_sth1()  
print(Myclass().\_\_dict\_\_)

# 模块

模块就是python中的一个.py文件  
在模块中定义的函数，和类，都成为模块的属性  
  
为什么使用模块  
代码重用：在模块中可以引用另一个模块，从而重用另一个模块的属性  
避免命名冲突：不同模块可以去相同的名字  
  
为了更好的组织和管理模块  
为了更好的管理模块，python引用了包  
在某个目录添加模块\_\_init\_\_.py之后，这个目录就变成了一个包  
因此，包时包含特定模块的特殊目录，模块\_\_init\_\_.py时初始化其所在的包，如果不需要初始化，其内容可以为空  
目录支持嵌套，包也支持嵌套  
  
使用当前项目中的模块  
必须要通过import进行导入  
直接导入  
 import MRO  
 print(MRO.m)  
 运行结果：1  
绝对导入：导入模块名字的绝对路径  
 import a.b.mod3  
 print(a.b.mode3.m)  
 此方法等同于  
 from a.b.mod3 import m  
 print(m)  
相对导入:导入的就是相对路径  
 使用'.'表示当前路径。'..'表示当前路径的父目录  
 注意：当直接运行某个模块时，这个模块就成了主模块位于最顶层，无法构成相对关系,所以当直接运行这个模块时就不能使用相对导入  
   
导入不同模块时出现相同属性  
错误的导入方式：  
 from moda import v  
 from modb import v  
 后面导入的属性会把前面的属性给覆盖掉  
正确的导入方式  
 给导入的属性起一个别名  
 from moda import v as va  
 from modb import v as vb  
 导入整个模块  
 import moda  
 import modb  
 print（moda.v）  
 print（modb.v）  
  
  
import导入的流程  
当时用户import进行导入时，系统会使用sys模块中的modules属性值来查找模块是否已经被导入了  
如果已经被导入了，系统什么都不做  
如果没有被导入  
 解释器会按照某种路径搜索模块  
 路径：存放在模块sys的变量path中  
 >>> import sys  
 >>> sys.path  
 ['', 'C:\\Users\\28646\\AppData\\Local\\Programs\\Python\\Python38-32\\python38.zip', 当前目录  
 'C:\\Users\\28646\\AppData\\Local\\Programs\\Python\\Python38-32\\DLLs', 标准库路径  
 'C:\\Users\\28646\\AppData\\Local\\Programs\\Python\\Python38-32\\lib',   
 'C:\\Users\\28646\\AppData\\Local\\Programs\\Python\\Python38-32',   
 'C:\\Users\\28646\\AppData\\Local\\Programs\\Python\\Python38-32\\lib\\site-packages']第三方库的安装目录  
 修改解释器搜索模块路径  
 直接修改sys.path  
 根据上述结果可知，sys.path。是一个列表  
 可以在代码运行的时候直接修改sys.path,但是在代码运行后就会失效  
 import sys  
 sys.path.insert(0,'C:\\Users\\28646\\Downloads') 首先就会,直接添加进去  
 import Xiazai 注意：一定要在设置之后导入，不然会找不到  
 print(Xiazai.f())  
 运行结果:f被调用  
 None  
 注意：这种方法在代码运行之后就会失效  
 设置环境变量PYTHONPASS  
 修改之后对应的路径会自动添加到sys.path中  
 修改之后搜索路径运行仍然有效  
 （可选）将搜索的模块可能会被编译为pyc字节码文件  
 当搜索的模块第一次被导入时会编译为pyc字节码文件，存放在该模块同目录下的\_\_pycache\_\_中  
 命名格式为：模块名.cpython-版本号.pyc  
 再次加载的时候就被缓存起来了，再次加载如果该模块没有发生变化，则无需将模块编译为pyc字节码文件，从而提升加载速度  
 执行编译的字节码文件从而运行模块  
 import pack1.pack2.pack3.modc  
 print('modd被运行') 会从父包开始运行  
 导入包中的模块时，会先导入包中的\_\_init\_\_.py,因此在运行被导入的模块钱，会从最顶层的父包开始，回一次运行所有\_\_init\_\_.py  
 import sys,pprint  
 import MRO  
 print(MRO.m)  
 pprint.pprint(sys.modules) 打印已经导入的模块，使用pprint只是为了让一行只显示一个字典  
  
  
  
dir()  
当调用内置函数dir时如果以'模块作为参数'会返回执行模块所有的属性和方法  
 代码：  
 import pprint  
 pprint.pprint(dir(pprint))  
 运行结果：返回的pprint所有的属性和方法  
如果不带任何参数  
 会返回当前模块中的局部作用域的所有属性  
 import pprint  
 pprint.pprint(dir())  
  
  
重新加载被导入的模块  
用于已经被修改的模块进行重新加载  
使用标准库函数reload重新加载  
格式：  
import importlib  
importlib.reload(模块名字)  
  
  
模块特殊属性\_\_doc\_\_(docstring)  
位于模块的第一行的字符串被称为模块的文档字符串，通常使用三个引号来表示  
模块文档字符串对模块功能简要描述  
可以实用工具根据文档字符串自动的生成文档  
访问文档字符串  
 通过\_\_doc\_\_  
 print(\_\_doc\_\_)  
 调用内置函数help()得到的帮助信息中也会包含文档字符串  
  
  
模块特殊属性\_\_name\_\_  
在不同情况下的取值  
 对于被导入的模块\_\_name\_\_就是模块名  
 对于直接运行的模块，特殊属性\_\_name\_\_为mian  
 print(\_\_name\_\_) \_\_main\_\_  
可以根据特殊属性name的值判断是否执行模块中的测试代码  
对于模块中的个测试代码通常直接运行才需要执行，而当模块被导入时不需要执行  
 if \_\_name\_\_ == 'name':  
 测试代码  
 这样导入模块时是不会执行的，只有当运行模块才会执行  
  
  
访问控制  
 在属性前，添加\_之后就不能导入属性了，只是from 模块名 import \* 对于import 模块名仍然可以使用相应的属性  
 定义特殊属性\_\_all\_\_  
 使用语句from 模块名 import \*只能导入特殊属性\_\_all\_\_定义的属性，但是使用语句import 模块名仍然可以导入所有属性（只是在某种程度  
 上限制了)  
 语法\_\_all\_\_ = ['属性名','属性名']  
 如果\_\_all\_\_ = ['\_a']这种情况\_\_all\_\_具有更高的优先级

# Anaconda

Anaconda:我们经常会用到很多第三方库  
如果使用工具pip3逐个安装，不仅费时费力，而且还要考虑兼容性问题  
Anaconda是一个基于python是基于数据处理和科学计算的平台，内置了很多有用的第三方库  
安装后就相当于把大量的第三方库安装好了  
  
安装和配置Anaconda  
 官方地址：https://www.anaconda.com/download  
   
 安装程序会把Anaconda安装目录下的bin目录添加到系统变量PATH中  
 added by Anaconda3 5,0,1 installer  
 export PATH="/Users/zhangringchao/anaconda3/bin:$PATH"  
 因此安装完Adaconda后，会使用自带的Python，从而  
 在命令行输入的python3来自于<Anaconda的安装目录>/bin  
 使用pip3安装的第三方库会被安装到<Anaconda的安装目录>/lib/python3.x/site-packages  
   
Anaconda第三方库管理工具conda  
 Anaconda使用工具conda进行第三方库进行管理的，类似于pip3  
 查看帮助信息：conda  
 列出Anaconda已经安装的所有第三方库：conda list  
 模糊搜索：conda search xxx  
 安装指定第三方库：conda install xxx=y.y (只有一个等号)  
 升级第三方库：conda update xxx  
 卸载指定的第三方库：conda remove xxx  
 查看conda之后的某个帮助信息：conda <命令> --help3

python提供了  
在windows《python的安装路径》\lib  
标准库是很好的学习资料  
  
如果想使用必须使用import进行导入  
1,导入整个模块  
2，导入模块中的属性  
  
导入整个模块  
import [包名.] 模块名  
如果被导入的模块在一个包结构中，必须通过其所有的副包然后导航到这个模块  
顶层父包名.子包名....子包名  
  
导入模块以后就可以访问模块中的属性了(变量，函数，类)  
语法格式：import [包名.] 模块名.属性名  
  
import os  
print(os.environ)  
python中指定的  
print(os.getenv('PYTHON\_HOME'))  
  
再导入的时候可以给导入的模块起一个别名  
import [包名.]模块名 as 模块的别名  
代码：  
import os as operating\_system  
print(operating\_system.environ)  
  
带入模块中的属性  
 格式：from [包名.] 模块名 import 属性名  
 导入模块属性之后就可以直接访问模块中的属性了，就无需添加前缀了，但是与添加前缀相比，代码可读性比较差，因为别人看到后不知道调用的时拿  
 的属性  
 from os import environ  
 print(environ)  
一次导入多个属性  
 格式：from [包名.] 模块名 import 属性名1,属性名2,属性名N  
 from os import environ,MutableMapping,getenv  
 print(environ)  
给绑定的起别名  
 语法格式:from [包名.] 模块名 import 属性名1 as 别名,属性名2 as 别名,属性名N as 别名  
 from os import environ as en,MutableMapping,getenv  
 print(en)  
可以将模块中的属性一次性全部导入  
 from [包名.] 模块名 import \*  
 强烈不推荐，因为效率比较低，代码可读性差，容易出错，尤其是当两个模块中出现相同的属性时  
  
当导入整个模块时也可以使用整个模块使用from进行导入  
但是在调用的时候也是要添加属性名

生成器

不存在元组生成式：因为元组时不可改变对象  
但是将列表生成式[]改成（），程序并不会报错，只是得到一个generator(生成器对象)  
 m = (i \* i for i in range(10))  
 print(m)  
 print(next(m))  
 多次调用next()  
   
 使用for - in语句进行迭代  
 m = (i \* i for i in range(20))  
 for item in m:  
 print(item)  
生成器保存的并不时被保存的所有元素，而是保存所有元素的算法，用于for-in语句时，元素时在不断被推算出来的  
因此生成器时惰性推算的，也就是说用到某个匀速时才会推算。  
如果需要创建一个较大的容器就可以考虑生成器从而节省大量空间  
上面的被称为生成表达式  
当推算的算法比较复杂时可以通过生成器函数来得到生成器  
def fib(n):  
 i = 1  
 a,b = 1,1  
 while i<n:  
 yield a 通过  
 a,b = b,a+b  
 i+=1  
fib(6)  
生成器yield返回推算出元素。生成器函数，当调用函数next()或者使用for-in语句进行迭代时，执行完yield语句就会将生成器挂起，下次从挂起的  
地方继续执行  
  
def fib(n):  
 i = 1  
 a,b = 1,1  
 while i<n:  
 yield a 通过  
 a,b = b,a+b  
 i+=1  
gf = fib(6)  
for item in gf:  
 print(item)

可迭代对象

可以用于for\_in语句的对象被称为可迭代对象（Iterable）  
  
range()，列表，元组，字符串，字典，生成器  
调用内置函数isinstance()判断一个对象是否是时可迭代对象，标准库collections中的Inerable类表示可迭代对象  
from collections import Iterable  
print(isinstance([1,2,3],Iterable))  
运行结果：True  
  
迭代器对象：  
首先是一个可迭代对象，可以作为next（）的实参，从而支持惰性推算，那就被称为迭代器对象  
Iterator  
列表元组，集合字典这些都不是迭代器对象，生成器可以  
生成器时迭代器的一种  
判断是否时迭代器对象：isinsrance()  
from collections.abc import Iterator  
print(isinstance([1,2,3],Iterator))  
  
使用调用内置函数iter（）把不支持惰性推算的不可迭代对象转化为可迭代对象  
from collections.abc import Iterator  
print(isinstance(iter([1,2]),Iterator))  
运行结果：True  
  
如果一个对象同时实现了\_\_iter\_\_()和\_\_next\_\_()，那该对象也被称为迭代器对象  
如果将该对象用于for-in语句，for-in语句会先调用特殊方法\_\_iter\_\_（）返回一个可迭代对象，然后不断调用可迭代对象的\_\_next\_\_()返回下一个  
值，直到遇到StopIteration时退出循环  
例子：  
class MyIter(object):  
 def \_\_init\_\_(self):  
 self.data = 0  
  
 def \_\_iter\_\_(self):  
 return self  
 def \_\_next\_\_(self):  
 if self.data > 5:  
 raise StopIteration  
 else:  
 self.data += 1  
 return self.data  
 return self  
for item in MyIter():  
 print(item)  
例子：  
class Fib(object):  
 def \_\_init\_\_(self):  
 self.a,self.b = 0, 1  
 def \_\_iter\_\_(self):  
 return self  
 def \_\_next\_\_(self):  
 self.a,self.b = self.b,self.a+self.b  
 if self.a > 50:  
 raise StopIteration()  
 return self.a  
  
for item in Fib():  
 print(item)

异常

异常指的是，陈虚谷在没有语法错误。在运行时间产生的特定的错误  
那些特定错误会被看做异常：  
 内置的异常对象和自定义的异常对象会被看作错误  
 内置的：  
 BaseException:所有异常的基类,包括退出异常和非退出异常；  
 SystemExit: 解释器请求退出  
 KeyboardInterrupt: 用户中断执行(通常是输入^C)  
 Exception: 常规错误的基类  
 StopIteration: 迭代器没有更多的值  
 GeneratorExit: 生成器(generator)发生异常来通知退出  
 ArithmeticError: 所有数值计算错误的基类  
 FloatingPointError: 浮点计算错误  
 OverflowError: 数值运算超出最大限制  
 ZeroDivisionError: 除(或取模)零 (所有数据类型)  
 AssertionError: 断言语句失败  
 AttributeError: 对象没有这个属性  
 EOFError: 没有内建输入,到达EOF标记  
 EnvironmentError: 操作系统错误的基类  
 IOError: 输入/输出操作失败  
 OSError: 操作系统错误  
 WindowsError: 系统调用失败  
 ImportError: 导入模块/对象失败  
 LookupError: 无效数据查询的基类  
 IndexError: 序列中没有此索引(index)  
 KeyError: 映射中没有这个键  
 MemoryError: 内存溢出错误(对于Python 解释器不是致命的)  
 NameError: 未声明/初始化对象 (没有属性)  
 UnboundLocalError: 访问未初始化的本地变量  
 ReferenceError: 弱引用(Weak reference)试图访问已经垃圾回收了的对象  
 RuntimeError: 一般的运行时错误  
 NotImplementedError: 尚未实现的方法  
 SyntaxError: Python语法错误  
 IndentationError: 缩进错误  
 TabError: Tab和空格混用  
 SystemError: 一般的解释器系统错误  
 TypeError: 对类型无效的操作  
 ValueError: 传入无效的参数  
 UnicodeError: Unicode相关的错误  
 UnicodeDecodeError: Unicode解码时的错误  
 UnicodeEncodeError: Unicode编码时错误  
 UnicodeTranslateError: Unicode转换时错误  
 Warning: 警告的基类  
 DeprecationWarning: 关于被弃用的特征的警告  
 FutureWarning: 关于构造将来语义会有改变的警告  
 OverflowWarning: 旧的关于自动提升为长整型(long)的警告  
 PendingDeprecationWarning: 关于特性将会被废弃的警告  
 RuntimeWarning: 可疑的运行时行为(runtime behavior)的警告  
 SyntaxWarning: 可疑的语法的警告  
 UserWarning: 用户代码生成的警告  
 所有内置异常类对象基类都是Exception  
  
  
使用try - except语句捕获异常  
 try:  
 result = (1 / 0)  
 except ZeroDivisionError:  
 print('0不能作为除数')  
 运行结果：0不能作为除数  
 try - except从上到下依次查找错误，如果找到了错误，那就不会在查找其他的了  
 代码：  
 try:  
 result = (1 / 0)  
 except ZeroDivisionError:  
 print('0不能作为除数')  
 except ImportError:  
 print('导入错误')  
 except TypeError:  
 print('类型错误')  
 print('结束')  
   
 try:  
 result = (1 / 0)  
 except Exception:  
 print('抛出错误')  
 print('结束')  
 如果多个中处理的方法一样，那就可以通过元组放在一个里面进行处理  
 try:  
 result = (1 / 0)  
 except (Exception,TypeError,ZeroDivisionError):  
 print('抛出错误')  
 print('结束')  
   
 try:  
 result = (1 / 0)  
 except ZeroDivisionError as ex:  
 print(type(ex))  
 print(ex)  
 print('抛出错误')  
 print('结束')  
else:如果语句中没有产生异常，那就会执行else:语句  
while True:  
 try:  
 i = int(input("请输入一个整数："))  
 except ValueError:  
 print("输入格式不正确（无效输入）")  
 else:  
 print('输入的整数为：',i)  
 break  
添加finally: 通常在finally中释放资源  
try:  
 pass  
except:  
 pass  
finally:  
 print('这里的代码一定会被执行')  
  
  
使用raise抛出异常  
try:  
 raise (ValueError('抛出ValueError异常'),Exception('抛出Exception异常')) 如果没有参数则就把不会显示参数的内容  
except ValueError:  
 print('接收了ValueError异常')  
except Exception:  
 print('接收了Exception异常')  
else:  
 print('没有检测到异常')  
finally:  
 print('程序结束，释放资源')  
运行结果：  
接收了Exception异常  
程序结束，释放资源  
  
  
自定义异常  
尽管python内置的异常类对象，但是有时候我么也可以创建自定义的异常类对象  
继承Exception或者他的子类  
class MyException(Exception):  
 def \_\_init\_\_(self,msg1):  
 self.msg1 = msg1  
try:  
 raise MyException('来自自定义异常')  
except ValueError:  
 print('自定义异常对象')  
正如所有内置异常类基类时Exception  
  
  
def f1():  
 print(1 / 0)  
def f2():  
 f1()  
def f3():  
 f2()  
f3()  
运行结果：  
Traceback (most recent call last):  
 File "D:/Pythonproject/venv/Include/Python高级/异常.py", line 143, in <module>  
 f3()  
 File "D:/Pythonproject/venv/Include/Python高级/异常.py", line 142, in f3  
 f2()  
 File "D:/Pythonproject/venv/Include/Python高级/异常.py", line 140, in f2  
 f1()  
 File "D:/Pythonproject/venv/Include/Python高级/异常.py", line 138, in f1  
 print(1 / 0)  
ZeroDivisionError: division by zero  
当发生异常时，异常实例对象会抛给函数的调用者，一直向上抛，一直到python解析器捕获  
  
所以可以在合适的曾进行捕获处理  
def f1():  
 print(1 / 0)  
def f2():  
 f1()  
def f3():  
 try:  
 f2()  
 except Exception as e:  
 print(e)  
f3()  
  
  
在捕获异常对象之后，可以调用exc\_info进行异常的相关信息  
exc\_info（）返回的是一个包含三个元素的元组，这三个元素分别表示，异常的类型，异常的错误信息，包含异常堆栈的Tracenack对象  
import sys  
def f1():  
 print(1/0)  
  
def f2():  
 try:  
 f1()  
 except ZeroDivisionError:  
 ex\_type,ex\_value,ex\_tracenack = sys.exc\_info()  
  
 print('异常的信息：',ex\_type)  
 print('异常的错误信息：',ex\_value)  
 print('异常调用的堆栈的跟踪信息：',ex\_tracenack)  
f2()  
当然也可以查看traceback的详细信息,可以调用标准库中的模块traceback中的函数traceback.extract\_tb(ex\_tracenack)  
import sys  
import traceback  
def f1():  
 print(1/0)  
  
def f2():  
 try:  
 f1()  
 except ZeroDivisionError:  
 ex\_type,ex\_value,ex\_tracenack = sys.exc\_info()  
 print('异常的信息：',ex\_type)  
 print('异常的错误信息：',ex\_value)  
 print('异常调用的堆栈的跟踪信息：',ex\_tracenack)  
  
 tb = traceback.extract\_tb(ex\_tracenack)  
 for filename, lienum, funcname, source in tb: 文件名，函数，行数，源码  
 print('文件名：', filename)  
 print('函数：', lienum)  
 print('行数：', funcname)  
 print('源码：', source)  
f2()  
运行结果:  
异常的信息： <class 'ZeroDivisionError'>  
异常的错误信息： division by zero  
异常调用的堆栈的跟踪信息： <traceback object at 0x00E51888>  
文件名： D:/Pythonproject/venv/Include/Python高级/异常.py  
函数： 197  
行数： f2  
源码： f1()  
文件名： D:/Pythonproject/venv/Include/Python高级/异常.py  
函数： 193  
行数： f1  
源码： print(1/0)  
  
import sys  
import traceback  
def f1():  
 print(1/0)  
  
def f2():  
 try:  
 f1()  
 except ZeroDivisionError:  
 ex\_type,ex\_value,ex\_tracenack = sys.exc\_info()  
 print('异常的信息：',ex\_type)  
 print('异常的错误信息：',ex\_value)  
 print('异常调用的堆栈的跟踪信息：',ex\_tracenack)  
  
 tb = traceback.extract\_tb(ex\_tracenack)  
 for filename, lienum, funcname, source in tb: 文件名，函数，行数，源码  
 print('文件名：', filename)  
 print('函数：', lienum)  
 print('行数：', funcname)  
 print('源码：', source)  
f2()

## with语句

上下文管理器  
 如果一个对象实现了\_\_enter\_\_()和\_\_exit\_\_()，那么就成这个类遵守了上下文管理协议，同时这个类对象的实例对象被称为上下文管理器  
 with语句会让上下文管器创建一个运行时的上下文，当进入到上下文时，就会自动调用特殊方法\_\_enter\_\_()，当离开运行时会调用特殊方法  
\_\_exit\_\_()  
with语法格式  
with 上下文表达式 [as 变量]：  
 with语句体  
  
  
class Context (object):  
 def \_\_enter\_\_(self):  
 print('特殊方法\_\_enter\_\_被调用了')  
 return self  
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb): 当产生异常时，就会将异常的三个值赋予给这三个值  
 print('特殊方法\_\_exit\_\_被调用了')  
 def do\_sth(self):  
 print('do\_sth被调用了')  
  
with Context() as mcm: Context()现在这个类中执行特殊方法\_\_enter\_\_()然后把返回值return self赋值给mcm，然后在执行do\_sth这个方法  
 mcm.do\_sth()  
  
特殊方法\_\_enter\_\_被调用了  
do\_sth被调用了  
特殊方法\_\_exit\_\_被调用了  
  
class Context (object):  
 def \_\_enter\_\_(self):  
 print('特殊方法\_\_enter\_\_被调用了')  
 return self  
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb): 当产生异常时，就会将异常的三个值赋予给这三个值  
 print('特殊方法\_\_exit\_\_被调用了')  
 print('异常的信息：', exc\_type)  
 print('异常的错误信息：', exc\_val)  
 print('异常调用的堆栈的跟踪信息：', exc\_tb)  
 return True  
 def do\_sth(self):  
 print('do\_sth被调用了')  
 print(1/0)  
  
with Context() as mcm: Context()现在这个类中执行特殊方法\_\_enter\_\_()然后把返回值return self赋值给mcm，然后在执行do\_sth这个方法  
 mcm.do\_sth()  
运行结果：  
特殊方法\_\_enter\_\_被调用了  
do\_sth被调用了  
特殊方法\_\_exit\_\_被调用了  
异常的信息： <class 'ZeroDivisionError'>  
异常的错误信息： division by zero  
异常调用的堆栈的跟踪信息： <traceback object at 0x0121F688>  
  
  
总结：如果with产生异常，那么sys.exc\_info()返回值列表的三个值会自动传递个特殊方法exit()的形参exc\_type, exc\_val, exc\_tb，这三个  
形参分别代表的时异常的类型，异常的错误信息，异常的调用堆栈的跟踪信息  
特殊方法\_\_exit\_\_()总会被执行，所以一般用来关闭文件关闭资源与finally类似  
  
  
当with语句体产生异常并且\_\_exit\_\_()没有返回True时也是可以通过try - extecp进行捕获处理的  
class Context (object):  
 def \_\_enter\_\_(self):  
 print('特殊方法\_\_enter\_\_被调用了')  
 return self  
 def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb): 当产生异常时，就会将异常的三个值赋予给这三个值  
 print('特殊方法\_\_exit\_\_被调用了')  
 print('异常的信息：', exc\_type)  
 print('异常的错误信息：', exc\_val)  
 print('异常调用的堆栈的跟踪信息：', exc\_tb)  
 def do\_sth(self):  
 print('do\_sth被调用了')  
 print(1/0)  
  
try:  
 with Context() as mcm: Context()现在这个类中执行特殊方法\_\_enter\_\_()然后把返回值return self赋值给mcm，然后在执行do\_sth这个方法  
 mcm.do\_sth()  
except ZeroDivisionError as err:  
 print(err)  
运行结果：  
特殊方法\_\_enter\_\_被调用了  
do\_sth被调用了  
特殊方法\_\_exit\_\_被调用了  
异常的信息： <class 'ZeroDivisionError'>  
异常的错误信息： division by zero  
异常调用的堆栈的跟踪信息： <traceback object at 0x02D0F648>  
division by zero

# Python中操文件

包括打开文件，读取文件，关闭文件  
 把数据存储到文件中时存储的方式之一  
  
  
打开文件  
open() 返回值是一个文件对象，通过文件对象读文件，写文件  
file:文件路径，及可以指定绝对路径也可以指定先对路径（必须）  
mode：文件的打开方式，默认值是'r',表示只读方式打开  
 r 模式  
 1、如果文件存在，打开文件，光标置于开头，原数据可读，不可写入新数据；  
 2、如果文件不存在，报错；  
 r+ 模式  
 1、如果文件存在，打开文件，光标置于开头，原数据保留，可写入新数据；  
 2、如果文件不存在，报错；  
 3、打开文件后，若没有读取操作，则光标默认在开头，此时如果马上写入数据，会覆盖原数据  
 w 模式  
 1、如果文件存在，打开文件，清空原数据，光标置于开头，可写入新数据；  
 2、如果文件不存在，则创建文件；  
 3、文件的数据不可读；  
 w+ 模式  
 效果同 w 模式，差异在于，可读取写入的数据  
 a 模式  
 1、如果文件存在，打开文件，原数据保留，光标置于末尾，可追加写入新数据；  
 2、如果文件不存在，则创建文件；  
 3、文件数据不可读  
 a+ 模式  
 效果同 a 模式，差异在于，文件数据可读注：打开文件后，由于光标在末尾，此时马上读取数据会为空，因为光标所在位置之后没有数据，此时可先移动光标的位置到开头后再读取；  
 'r' open for reading (default)  
 'w' open for writing, truncating the file first  
 'x' create a new file and open it for writing  
 'a' open for writing, appending to the end of the file if it exists  
 'b' binary mode（二进制读取）  
 't' text mode (default)  
 '+' open a disk file for updating (reading and writing)  
 'U' universal newline mode (deprecated)  
其他参数都是带默认值的可以参考官方文档  
myfile = open('C:\\Users\\28646\\Desktop\\myfile.txt', 'r+')  
  
  
读取文件  
以只读或者读写文件，有三个读方法  
read([size])如果不指定参数，那就一直读到结束如果传入参数，那就是用于指定字节数  
如果指定的字节数小于0字节数时读到文件尾  
如果指定的字节数大于文件尾的字节数时读到文件尾  
try:  
 myfile = open('C:\\Users\\28646\\Desktop\\myfile.txt', 'r+', encoding='utf-8')  
 print(myfile.read(3))  
except Exception as e:  
 print(e)  
finally:  
 myfile.close()  
运行结果：这是我  
注意：如果文件较大，如果直接调取read()方法会导致内存占用较大，所以最好多次调用read()给传入参数，最好不要超过默认缓冲区的大小，否则  
可能并不会读取到我们指定的字节数  
通过标准库中的io查看默认缓冲区的大小  
import io  
print(io.DEFAULT\_BUFFER\_SIZE)  
运行结果：  
8192  
在通过read()设置读取的字节的时候最好不要超过缓冲区的大小8192  
  
  
readline():如果不传入参数，读到行尾  
如果不传入参数是读到行尾（一行的结尾），如果指定参数size读取指定的字节数  
 当指定的字节数小于读到行尾的字节数时，读取指定的字节数  
 当指定的字节数大于读到行尾的字节数时，或者指定的字节数小于0时，读到行尾  
try:  
 myfile = open('C:\\Users\\28646\\Desktop\\myfile.txt', 'r+', encoding='utf-8')  
 while True:  
 line = myfile.readline()  
 print(line,end='')  
 if not line:  
 break  
 ():这是我自己编写的myfile文件  
 (7):这是我自己编写  
 (90):这是我自己编写的myfile文件  
except Exception as e:  
 print(e)  
finally:  
 myfile.close()  
  
  
readlines():不传参数，读到文件尾，如果指定参数，那size指定字节数,z  
注意：返回一个列表，传入的参数是，字节而不是行数，传入的参数也是对应的行数，而不是只会读取到字节  
import pprint  
try:  
 myfile = open('C:\\Users\\28646\\Desktop\\myfile.txt', 'r+', encoding='utf-8')  
 line = myfile.readlines()  
 pprint.pprint(line)  
 ():这是我自己编写的myfile文件  
 (7):这是我自己编写  
 (90):这是我自己编写的myfile文件  
except Exception as e:  
 print(e)  
finally:  
 myfile.close()  
  
  
写文件  
write(text)用于将字符串写入到文件中。调用方法之后只是先到了缓存中，需要调用flush()（刷新）close()(关闭)或者当写入的文件大于缓存才  
会被写入到文件  
方法的返回值就是写入字符串的长度  
try:  
 myfile = open('C:\\Users\\28646\\Desktop\\myfile.txt', 'w', encoding='utf-8')  
 i = myfile.write(str(dir()))  
 if type(i) is int:  
 print('文件写入成功')  
 myfile.flush()  
except Exception as e:  
 print(e)  
finally:  
 myfile.close()  
   
writelines(sep)将指定的字符串序列写入文件中 【】或者（） 注意：这里面所有的内容包括数字都是要带引号的  
注意：这个方法是没有返回值的  
try:  
 myfile = open('C:\\Users\\28646\\Desktop\\myfile.txt', 'w', encoding='utf-8')  
 myfile.writelines(['11111111111111111111111\n','24242424','iuyaguayd']) 这里如果不换行，在文件中就不会换行  
except Exception as e:  
 print(e)  
finally:  
 myfile.close()  
  
  
  
关闭文件

因为文件对象会占用操作系统的资源，并且操作系统在某一刻打开的文件也是有限的  
这里这么干  
try:  
 myfile = open('C:\\Users\\28646\\Desktop\\myfile.txt', 'w', encoding='utf-8')  
 myfile.writelines(['11111111111111111111111\n','24242424','iuyaguayd']) 这里如果不换行，在文件中就不会换行  
except Exception as e:  
 print(e)  
finally:  
 myfile.close()  
   
由于文件对象实现了\_\_enter\_\_()，和\_\_exit\_\_（）,所以文件对象可以作为上下文管理器，其中特殊方法\_\_enter\_()返回打开的文件对象  
\_\_exit\_\_()中关闭打开的文件  
格式：  
with 打开文件 as file:  
 读文件或者写文件  
注意：在特殊方法\_\_exit\_\_()中会自动的关闭文件，所以使用with语句不用每次都手动的关闭文件了  
with open('C:\\Users\\28646\\Desktop\\myfile.txt', 'w', encoding='utf-8') as file:  
 file.writelines({'11111111111111111111111\n':'1','24242424':'2','iuyaguayd':'3','自己编写的文件':'23'})  
注意：使用字典也可以写入，但是写入之后只有key的值  
  
  
  
  
文件指针

用于指向文件的某个位置  
读写文件时，是从文件当前开始写的，在读写过程中，文件指针会随后移动  
  
打开文件指针的位置：  
 以追加方式打开时，文件指针指向文件结尾位置  
 以其他文件打开时候，文件指针指向文件开始位置  
tell()返回文件指针位置  
代码：  
with open('C:\\Users\\28646\\Desktop\\myfile.txt', 'r+', encoding='utf-8') as file:  
 file.writelines({'11111111111111111111111\n','24242424','iuyaguayd','自己编写的文件'})  
 print(file.tell())  
运行结果：  
63  
代码：  
with open('C:\\Users\\28646\\Desktop\\myfile.txt', 'r', encoding='utf-8') as file:  
 print(file.tell())  
运行结果：  
0  
  
  
读写文件时文件指针的移动过程  
代码：  
with open('C:\\Users\\28646\\Desktop\\myfile.txt', 'r+', encoding='utf-8') as file:  
 file.read(11)  
 print(file.tell())  
运行结果：  
15  
代码：  
with open('C:\\Users\\28646\\Desktop\\myfile.txt', 'r+', encoding='utf-8') as file:  
 file.read(2)  
 print(file.tell())  
 file.read(4)  
 print(file.tell())  
 file.read(17)  
 print(file.tell())  
 file.read()  
 print(file.tell())  
运行结果：  
2  
6  
37  
63  
  
  
自由移动文件指针  
seek(offset,[whence])   
offset:表示偏移量，负数就表示向后偏移  
whence：表示偏移位置，有三个取值  
 os.SEEK\_SET：相对文件的起始位置，值为0，默认值  
 os.SEEK\_CUR：相对文件的当前位置值为1  
 os.SEEK\_END：相对文件的结尾位置，值为2  
 注意：只有当以二进制打开文本时才支持后两种取值，不是二进制打开只支持第一种取值  
import os  
with open('C:\\Users\\28646\\Desktop\\myfile.txt', 'r+', encoding='utf-8') as file:  
 print(file.tell())  
 file.seek(3, os.SEEK\_SET)  
 file.seek(3)  
 print(file.tell())  
  
  
msg = "北京"  
print(msg.encode(encoding = "utf-8"))unicode编码转换为utf-8编码  
print(msg.encode(encoding = "utf-8").decode(encoding = "utf-8"))unicode编码转换为utf-8编码，再转化为unicode编码

# 函数（高级篇）

在python中一切皆对象，函数也是一个对象，从而函数也可以被赋予给变量  
一个函数可以作为另外一个函数的实参  
一个函数可以作为另外一个函数的返回值  
注意：在这些操作时候不能带小括号，因为如果带上小括号那就等于调用了这个函数  
  
def eval\_square():  
 def inner():  
 print('This is inner')  
 return inner  
eval\_square()()  
运行结果：  
This is inner  
  
  
lambda表达式  
当函数体只有一行return时就可以使用lambda表达式来代替  
格式：lamdba [形式参数1，形式参数2，....形式参数n]：关于形式参数的表达式  
 没有函数名  
 没有关键字  
 没有小括号  
 关于形式参数的表达式相当于函数的返回值  
(lamdba表达式就是匿名简化版的函数)  
def add(num1, num2):  
 return num1 + num2  
print(add(5,2))  
print((lambda num1, num2 : num1 + num2)(5,2))  
lm = lambda num1, num2 : num1 + num2  
print(lm(5,2))  
  
在python中一切皆是对象，所以lamdba表达式也是对象，也可以赋给变量  
因为lamdba时一个函数。所以也可以作为一个实参  
代码：  
result = map(lambda i:i\*\*i,[1,2,3,4])  
print(list(result))  
运行结果：  
[1, 4, 27, 256]  
  
所以对于函数的特性lamdba表达式也是适用的  
  
偏函数  
定义的时候可以给形参设置默认值，这样就简化了调用，因为只有与默认值不符合的形参才需要额外的形参  
偏函数也可以简化函数的调用  
from functools import partial  
def f(a,b = 5):  
 print('a =',a,'b =',b)  
  
f\_new = partial(f, 2) 2就会传递给a，就是说之后的调用f\_new传入的参数只是传递给b了等于是给a设置了一个默认值2  
f\_new()  
f\_new(6)  
f\_new(b = 6)  
代码：  
from functools import partial  
def eval\_sum(\*agrs):  
 s = 0  
 for i in agrs:  
 s += i  
 return s  
s = partial(eval\_sum,1,1)(1,1,1,1,1,1,1,1,1)  
print(s)  
运行结果：  
11  
代码：  
from functools import partial  
def eval\_sum(\*agrs):  
 s = 0  
 for i in agrs:  
 s += i  
 return s  
eval\_sum\_new = partial(eval\_sum,1,1) partial第一个参数时函数名，之后的参数是相遇eval\_sum内部传递的默认值  
print(eval\_sum\_new(11111)) 其实调用的还是转换前的函数  
代码：  
from functools import partial  
def f1(a, b = 5, \*args, \*\*kwargs):  
 print('a =', a, 'b =', b, 'args =', args, 'kwargs =', kwargs)  
  
f1\_new = partial(f1, 1, 3, 6, m = 8)  
print(f1\_new(2, 4, n=9))  
推断a = 1 b = 5 args = 3,6,2,4 kwargs = [m = 8,n = 9] b会被传入给3  
a = 1 b = 3 args = (6, 2, 4) kwargs = {'m': 8, 'n': 9}  
  
def f2(a, b=5,\*,c,\*\*kwargs): (a, b=5,\*,c,\*\*kwargs)\*就表示后面的参数都是可变的了  
 pass  
  
可以将已有的函数，转化为新的函数。  
再转化过程中，可以指定若干个位置实参关键字实参，位置实参和关键字实参就可以了。转换后的新函数被称为新函数的偏函数

闭包

闭包满足的三个条件  
 如果在一个函数定义了另外一个函数（姑且称之为内函数和外函数）  
 在内函数中引用了外函数的变量  
 并且外函数的返回值是一个内函数，这样就变成了一个闭包  
   
 通常情况下，在函数调用结束后，函数内定义的变量将不再可用  
 但是对于闭包而言，在外函数调用结束以后,外函数中被内函数引用的变量仍然是可用的，因为在外函数中被内函数的引用的变量会被绑定到  
内函数的特殊属性\_\_closure\_\_  
  
代码：  
def outer():   
 i = 10 对于i 在外函数调用之后仍然是可用的，因为构成了一个闭包，所以i被赋予给了\_\_closure\_\_，而通过  
 outer\_result.\_\_closure\_\_[0].cell\_contents就能访问到  
 def inner():  
 print(i)  
 return inner  
  
outer\_result = outer()  
outer\_result()  
outer()()  
print(outer\_result.\_\_closure\_\_)  
print(outer\_result.\_\_closure\_\_[0].cell\_contents)  
运行结果：  
10  
10  
(<cell at 0x008DE568: int object at 0x71A8C840>,)  
10  
  
代码：  
def outer():  
 i = 10  
 def inner():  
 print(i)  
 return inner  
  
print(outer().\_\_closure\_\_[0].cell\_contents)  
总结：所以如果形成闭包的三个条件，那就可以访问函数内的值  
  
在内函数  
def outer():  
 a = 10  
 def inner():  
 a = 10 这样等于是又重新定义了一个变量a  
 a += 1这样等于是又重新定义了一个变量a 把外面的给屏蔽了  
 pass  
 return inner  
  
代码：  
def outer():  
 a = [3]  
 def inner3():  
 a = [1] 这样等于是又重新定义了一个变量a  
 a[0] = 9 是因为a引用的是一个（可变）类型的对象，  
 print(a)  
 return inner3  
outer()()  
运行结果：[9]  
总结：在默认情况下，在内函数中是不可以修改外函数的内容的，但是对于可变类型的是可以修改它的值的

关键字nonlocal  
如果想在内函数中引用外函数的变量那就可以使用关键字nonlocal  
nonlocal:在内函数中对外函数的变量进行声明，说明不是局部的  
代码：  
def outer():  
 a = 10  
 def inner():  
 nonlocal a  
 a = 15  
 print(a)  
 return inner  
outer()()  
运行结果：  
15  
  
  
代码：  
def outer1():  
 a = 16  
 print('outer1中a的值为：',a)  
 def outer():  
 nonlocal a  
 a = 10  
 print('outer中a的值为：', a)  
 def inner():  
 nonlocal a  
 a = 15  
 print('inner中a的值为：', a)  
 return inner  
 return outer  
outer1()()()  
运行结果：  
outer1中a的值为： 16  
outer中a的值为： 10  
inner中a的值为： 15  
  
def outer1():  
 a = 16  
 print('outer1中a的值为：',a)  
 def outer():  
 nonlocal a  
 a = 10  
 print('outer中a的值为：', a)  
 def inner():  
 nonlocal a  
 a = 15  
 print('inner中a的值为：', a)  
 return inner  
 return outer  
outer1()()()

变量的作用域

是由定义变量的位置所决定的  
 局部作用域(Local)：每次调用某个函数时就会创建一个局部作用域，局部中定义的就做局部作用域  
 是从函数调用开始起作用，在函数结束之后所有的变量都会被销毁  
 嵌套作用域（Enclosing）：  
 每次调用嵌套函数中的外函数时，就会创建一个嵌套作用域  
 当在外函数内定义变量时，该作用域为：从定义变量开始到函数结束  
 外函数调用结束后，对其对应的桥套作用域都会被销毁  
 全局作用域(Global):  
 每次运行模块时都会创建一个全局作用域  
 全局作用域：从定义变量开始到模块结束  
 程序结束后，全局变量都会被销毁  
 内置作用域(Built-in)  
 每次启动python解释器都会自动加载内置模块，从而创建一个内置作用域  
 内置模块中的函数（内置函数），可以在程序中直接使用  
 停驶解释器后，内置作用域中的所有变量都会被销毁  
print(id(123))  
id = "Global"  
def outside():  
 id = "Enclosing"  
 def inside():  
 id = "Local"  
 print(id)  
 inside()  
outside()  
查找顺序：LEGB依次搜索，找到了就停止搜索，如果没有找到就会抛出NameError  
 1,id = "Local"  
 2,id = "Enclosing"  
 3,id = "Global"  
 4,print(id(123))  
因此，如果根据LEGB的顺序，前面定义的变量会屏蔽掉后面定义的变量  
注意：全部变量在局部中是不能被访问的  
代码：  
class MyClass(object):  
 a = 18  
 def \_\_init\_\_(self):  
 print(a)  
MyClass()  
运行结果：NameError: name 'a' is not defined（a没有定义），这里在类中是可以通过self在局部变量中访问的  
  
如果想在局部作用域中使用全局地变量可以使用global进行声明  
a = 49  
def s\_sth():  
 global a  
 print(a)  
s\_sth()  
运行结果：  
49  
  
对于流程控制语句，和异常处理语句  
if,for-in,while，try - extept  
这写里面定义的变量在结束之后，都是可以访问的  
  
内置函数locals()和globals()  
 命名空间：某个作用域内所有名字值和名字的映射，用字典来表示  
 locals():返回其局部作用域的命名空间  
 globals():返回全局作用域的命名空间  
代码：  
def outer(a):  
 b = 8  
 def inner(c):  
 d = 3  
 print(locals())  
 return inner  
 print(locals())  
outer(8)(5)  
运行结果：{'c': 5, 'd': 3}  
代码：  
def outer(a):  
 b = 8  
 def inner(c):  
 d = 3  
 print(locals())  
 print(locals())  
outer(8)  
运行结果：{'a': 8, 'b': 8, 'inner': <function outer.<locals>.inner at 0x030D56E8>}  
  
代码：  
a = 6  
def outer(a):  
 b = 8  
 def inner(c):  
 d = 3  
 print('在inner中',locals())  
 print('在outer中',globals())  
print('在模块中',globals())  
outer(8)  
运行结果：  
在模块中 {  
'\_\_name\_\_': '\_\_main\_\_',   
'\_\_doc\_\_': '\n变量的作用域\n',   
'\_\_package\_\_': None,   
'\_\_loader\_\_': <\_frozen\_importlib\_external.SourceFileLoader object at 0x00EFE718>,   
'\_\_spec\_\_': None, '\_\_annotations\_\_': {},   
'\_\_builtins\_\_': <module 'builtins' (built-in)>,   
'\_\_file\_\_': 'D:/Pythonproject/venv/Include/Python高级/变量的作用域.py',   
'\_\_cached\_\_': None,   
'a': 6,   
'outer': <function outer at 0x00ED7808>}  
在outer中 {  
'\_\_name\_\_': '\_\_main\_\_',   
'\_\_doc\_\_': '\n变量的作用域\n',   
'\_\_package\_\_': None,   
'\_\_loader\_\_': <\_frozen\_importlib\_external.SourceFileLoader object at 0x00EFE718>,   
'\_\_spec\_\_': None,   
'\_\_annotations\_\_': {},   
'\_\_builtins\_\_': <module 'builtins' (built-in)>,   
'\_\_file\_\_': 'D:/Pythonproject/venv/Include/Python高级/变量的作用域.py',   
'\_\_cached\_\_': None,   
'a': 6,   
'outer': <function outer at 0x00ED7808>}

locals()  
locals()返回的只是一个值的拷贝，如果修改某个值对于实际的值是没有任何影响的，但是可以添加一个映射  
a = 5  
locals()['b'] = 5  
print(locals())  
print(b)  
运行结果：  
{'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': '\n变量的作用域\n', '\_\_package\_\_': None, '\_\_loader\_\_': <\_frozen\_importlib\_external.SourceFileLoader object at 0x01BBE718>, '\_\_spec\_\_': None, '\_\_annotations\_\_': {}, '\_\_builtins\_\_': <module 'builtins' (built-in)>, '\_\_file\_\_': 'D:/Pythonproject/venv/Include/Python高级/变量的作用域.py', '\_\_cached\_\_': None, 'a': 5, 'b': 5}  
5  
总结：使用locals()查找到的值添加之后是起效果的，而且还能够使用  
  
  
globals()返回的是一个实际的命名空间，对globals()修改就是对实际的命名空间进行修改  
代码：  
globals()['\_\_doc\_\_'] = "作用域"  
print(globals())  
运行结果：  
{  
'\_\_name\_\_': '\_\_main\_\_',   
'\_\_doc\_\_': '作用域',   
'\_\_package\_\_': None,   
'\_\_loader\_\_': <\_frozen\_importlib\_external.SourceFileLoader object at 0x0294E718>,   
'\_\_spec\_\_': None,   
'\_\_annotations\_\_': {},   
'\_\_builtins\_\_': <module 'builtins' (built-in)>,   
'\_\_file\_\_': 'D:/Pythonproject/venv/Include/Python高级/变量的作用域.py',   
'\_\_cached\_\_': None}  
代码：  
globals()['\_\_doc\_\_'] = "作用域"  
print(\_\_doc\_\_)  
运行结果：

作用域  
可以在局部作用域或者嵌套作用域访问被屏蔽的全局变量的  
代码：  
f = 10  
def f1():  
 f = globals()['f']  
 print(f)  
f1()  
运行结果：  
10  
  
代码：  
f = 10  
def f1():  
 f = 19  
 print(globals()['f'])  
f1()  
运行结果：  
10  
  
  
通过这种方法就可以查看一个函数的帮助信息  
>>> help(vars)  
Help on built-in function vars in module builtins:  
帮助内建模块的功能变量:  
vars(...)  
 vars([object]) -> dictionary  
 var([对象])- >字典  
 Without arguments, equivalent to locals().  
 没有参数，等价于locality()。  
 With an argument, equivalent to object.\_\_dict\_\_.  
 如果传入一个对象那就等同于object.\_\_dict\_\_.  
  
函数装饰器  
  
  
 对于某个函数，我们不希望改变函数代码的前提下，函数增加额外的功能，那么就可以使用装饰器来装饰该函数  
 装饰器本身就是一个函数：  
 装饰器接收一个函数作为参数（传入实参是被装饰的函数，）装饰器的内部嵌套定义着另一个函数，内函数会引用函数的装饰器的参数，并且装饰器  
的返回值是内函数，这样就构成了一个闭包，为了让内函数wrapper接收任意类型的函数，所以将wrapper形参定义为(\*args,\*\*kwargs)。在函数中  
首先完成的是新添加的功能，然后调用被装饰的函数  
 把装饰器应用到装饰器的语法为：在被装饰函数前面添加一个“@被装饰的函数名”。在被装饰函数sum添加了@log之后就相当于执行了sum = log(sum)  
首先，被装饰的函数sum会作为实参传递给装饰器log，然后，返回装饰器的内函数wrapper，最后，将内函数wrapper赋值给名为sum（）的变量  
代码：  
def log(func): 将sum作为实参传递给了log  
 def wrapper(\*args,\*\*kwargs): \*args,\*\*kwargs用于接收所有的关键字实参和实参  
 print("函数%s被调用了" % func.\_\_name\_\_)  
 print("在wrapper内部：", 'args =', args, 'kwargs =',kwargs)  
 return func(\*args,\*\*kwargs) 执行接收到的func这个函数，并且将参数放到里面（等同于执行了以前的函数）  
 return wrapper 返回一个wrapper就相当于调用了这个函数  
@log 添加这样一个标记就相当于执行了sum = log(sum)  
def sum(sum1, sum2):  
 print(sum1, sum2)  
 return sum2 + sum1  
print(sum(12,13))  
  
  
print(sum.\_\_name\_\_) wrapper  
如果希望被装饰的函数名字为sum那可以添加另一个装饰器4  
@functools.wraps(func) 这是标准库中的模块funtools的wraps函数  
代码：  
import functools  
def log(func): 将sum作为实参传递给了log  
 @functools.wraps(func) 这是标准库中的模块funtools的wraps函数  
 def wrapper(\*args,\*\*kwargs): \*args,\*\*kwargs用于接收所有的关键字实参和实参  
 return func(\*args,\*\*kwargs) 执行接收到的func这个函数，并且将参数放到里面（等同于执行了以前的函数）  
 return wrapper 返回一个wrapper就相当于调用了这个函数  
@log 添加这样一个标记就相当于执行了sum = log(sum)  
def sum(sum1, sum2):  
 return sum2 + sum1  
print(sum.\_\_name\_\_)  
运行结果：  
sum  
  
把装饰器应用到被装饰函数时，还可以传递额外的参数，此时需要编写一个三层嵌套的函数  
@log('6月', '18日')就相当于执行了语句sum = log('6月', '18日')(sum)  
代码：  
import functools  
def log(month, day):  
 def decorator(func):  
 @functools.wraps(func)  
 def wrapper(\*args,\*\*kwargs):  
 print('%s%s,函数%s被调用了' % (month, day, func))  
 return func(\*args,\*\*kwargs)  
 return wrapper  
 return decorator  
@log('6月', '18日')  
def sum(sum1, sum2):  
 return sum2 + sum1  
print(sum(1,1))  
运行结果：  
6月18日,函数<function sum at 0x00A859B8>被调用了  
2