**1、面向过程与面向对象的对比**

**面向过程的程序设计**的核心是过程（流水线式思维），过程即解决问题的步骤，面向过程的设计就好比精心设计好一条流水线，考虑周全什么时候处理什么东西。

优点是：极大的降低了程序的复杂度   
缺点是：一套流水线或者流程就是用来解决一个问题，生产汽水的流水线无法生产汽车，即便是能，也得是大改，改一个组件，牵一发而动全身。   
应用场景：一旦完成基本很少改变的场景，著名的例子有Linux內核，git，以及Apache HTTP Server等。

**面向对象的程序设计**的核心是对象（上帝式思维），要理解对象为何物，必须把自己当成上帝，上帝眼里世间存在的万物皆为对象，不存在的也可以创造出来。对象是特征和技能的结合，其中特征和技能分别对应对象的数据属性和方法属性。   
优点是：解决了程序的扩展性。对某一个对象单独修改，会立刻反映到整个体系中，如对游戏中一个人物参数的特征和技能修改都很容易。   
缺点：可控性差，无法向面向过程的程序设计流水线式的可以很精准的预测问题的处理流程与结果，面向对象的程序一旦开始就由对象之间的交互解决问题，即便是上帝也无法预测最终结果。于是我们经常看到一个游戏人某一参数的修改极有可能导致阴霸的技能出现，一刀砍死3个人，这个游戏就失去平衡。   
应用场景：需求经常变化的软件，一般需求的变化都集中在用户层，互联网应用，企业内部软件，游戏等都是面向对象的程序设计大显身手的好地方。

**2、id、type和value的概念**

在python当中一切皆对象,每产生一个对象会对应三个属性:id、类型type和数值   
id可以理解为在内存当中的位置(其实不是,id实际指向的是对象的地址)   
is是身份运算符,其中id对应的就是身份。   
id相同,数值肯定相同;id不相同,数值一定不相同吗？不是。   
代码示例：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

x = 10

print(id(x))

print(type(x))

print(x)

y = 10

print(id(y))

print(type(y))

print(y)

#判断x和y的内存地址是否相同

print(x is y)

#判断x和y的数值是否相同

print(x == 7)

[复制代码](javascript:void(0);)

运行结果：

[复制代码](javascript:void(0);)

1547159920

<class 'int'>

10

1547159920

<class 'int'>

10

True

False

Process finished with exit code 0

[复制代码](javascript:void(0);)

**3、类和对象的概念**

1>把一类事物的静态属性和动态可以执行的操作组合在一起所得到的这个概念就是类   
2>类的一个个体就是对象,对象是具体的,实实在在的事物   
3>对象是特征与技能的结合体，其中特征和技能分别对应对象的数据属性和方法属性   
4>对象（实例）本身只有数据属性，但是python的class机制会将类的函数绑定到对象上，称为对象的方法，或者叫绑定方法，绑定方法唯一绑定一个对象，同一个类的方法绑定到不同的对象上，属于不同的方法，内存地址都不会一样   
在类内部定义的属性属于类本身的,由操作系统只分配一块内存空间,大家公用这一块内存空间   
5>创建一个类就会创建一个类的名称空间，用来存储类中定义的所有名字，这些名字称为类的属性：而类中有两种属性：数据属性和函数属性，其中类的数据属性是共享给所有对象的，而类的函数属性是绑定到所有对象的。   
6>创建一个对象（实例）就会创建一个对象（实例）的名称空间，存放对象（实例）的名字，称为对象（实例）的属性   
7>在obj.name会先从obj自己的名称空间里找name，找不到则去类中找，类也找不到就找父类…最后都找不到就抛出异常。   
8>类的相关方法：

[复制代码](javascript:void(0);)

类的相关方法(定义一个类,也会产生自己的名称空间)

类名.\_\_name\_\_ # 类的名字(字符串)

类名.\_\_doc\_\_ # 类的文档字符串

类名.\_\_base\_\_ # 类的第一个父类(在讲继承时会讲)

类名.\_\_bases\_\_ # 类所有父类构成的元组(在讲继承时会讲)

类名.\_\_dict\_\_ # 类的字典属性、名称空间

类名.\_\_module\_\_ # 类定义所在的模块

类名.\_\_class\_\_ # 实例对应的类(仅新式类中)

[复制代码](javascript:void(0);)

9>其余概念：

[复制代码](javascript:void(0);)

1.创建出类会产生名称空间,实例化对象也会产生名称空间。

2.用户自己定义的一个类,实际上就是定义了一个类型,类型与类是统一的。

3.用户先是从自己的命名空间找,如果找不到,在从类的命名空间找。

student1.langage = "1111"

print(student1.\_\_dict\_\_) ===>先是从自己的命名空间找

print(Student.\_\_dict\_\_) ===>然后在从类的命名空间找

4.通过类来访问,访问的是函数,通过对象来访问,访问的是方法,在类内部定义的方式实际上是绑定到对象的身上来用的。

<function Student.fun at 0x000000000267DAE8>

<bound method Student.fun of <\_\_main\_\_.Student object at 0x0000000002684128>>

<function Student.fun at 0x00000000025CDAE8>

<bound method Student.fun of <\_\_main\_\_.Student object at 0x00000000025D4160>>

<bound method Student.fun of <\_\_main\_\_.Student object at 0x00000000025D4198>>

5.总结：类的数据属性是大家共有的，而且大家的内部地址是一样的,用的就是一个

类的函数属性是绑定到大家身上的，内部地址不一样，绑定方法指的是绑定到对象身上。

绑定方法：绑定到谁的身上,就是给谁用的,谁来调用就会自动把自己当做第一个参数传入。

\*\*定义在类内部的变量,是所有对象共有的,id全一样,

\*\*定义在类内部的函数,是绑定到所有对象的,是给对象来用的,obj.fun()会把obj本身当做

一个参数来传递。

6.在类内部定义的函数虽然可以由类来调用,但是并不是为了给类用的,在类内部定义的函数的目的就是为了绑定到对象身上的。

7.在类的内部来说,\_\_init\_\_是类的函数属性,但是对于对象来说,就是绑定方法。

8.命名空间的问题:先从对象的命名空间找,随后在从类的命名空间找,随后在从父类的命名

空间找。

print(student1.x)

9.在定义类的时候,可以想什么先写什么。

[复制代码](javascript:void(0);)

示例程序1：编写一个学生类，产生一堆学生对象，要求有一个计数器（属性），统计总共实例了多少个对象。

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

class Student():

#在类内部定义的属性属于类本身的,由操作系统只分配一块内存空间,大家公用这一块内存空间。

count = 0

def \_\_init\_\_(self,name,age):

self.name = name

self.age = age

Student.count +=1

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

student1 = Student("lidong",25)

print(student1.\_\_dict\_\_)

student2 = Student("wangwu",28)

print(student2.\_\_dict\_\_)

print(Student.count)

[复制代码](javascript:void(0);)

运行结果：

{'age': 25, 'name': 'lidong'}

{'age': 28, 'name': 'wangwu'}

2

Process finished with exit code 0

示例程序2：（对象之间的交互，重点）

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

#定义盖伦类和瑞文类,并进行互相残血

#对象之间的交互问题(面向对象之间互相交互)

class Garen:

camp = "Demacia"

#定义一个对象的时候,指定了这个对象的生命值和杀伤力

def \_\_init\_\_(self,nickname,life\_value=200,aggre\_value=100):

self.nickname = nickname

self.life\_value = life\_value

self.aggre\_value = aggre\_value

def attack(self,enemy):

enemy.life\_value = enemy.life\_value - self.aggre\_value

class Riven:

camp = "Demacia"

# 定义一个对象的时候,指定了这个对象的生命值和杀伤力

def \_\_init\_\_(self, nickname, life\_value=100, aggre\_value=200):

self.nickname = nickname

self.life\_value = life\_value

self.aggre\_value = aggre\_value

def attack(self, enemy):

#python为弱类型语言

enemy.life\_value = enemy.life\_value - self.aggre\_value

g = Garen("盖伦")

r = Riven("瑞文")

print("盖伦的生命值是%s"%g.life\_value)

print("瑞文的生命值是%s"%r.life\_value)

g.attack(r)

print("瑞文的生命值是%s"%r.life\_value)

[复制代码](javascript:void(0);)

运行结果：

盖伦的生命值是200

瑞文的生命值是100

瑞文的生命值是0

Process finished with exit code 0

**4、初始化构造函数\\_\_init\_的作用**

所谓初始化构造函数就是在构造对象的同时被对象自动调用,完成对事物的初始化,一个类只要生成一个类对象,它一定会调用初始化构造函数.   
特点:   
1>一个类中只能有一个初始化构造函数   
2>不能有返回值   
3>可以用它来为每个实例定制自己的特征   
示例程序：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

class Student():

def \_\_init\_\_(self,name,age):

self.name = name

self.age = age

print(self.name,self.age)

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

#在构造对象的时候会自动调用初始化构造函数

student = Student("Alex",100)

[复制代码](javascript:void(0);)

运行结果：

Alex 100

Process finished with exit code 0

**5、self关键字的用法**

为了辨别此时此刻正在处理哪个对象,self指针变量指向当前时刻正在处理的对象,即构造出来的对象   
在构造方法中self代表的是：self指针变量指向当前时刻正在创建的对象   
构造函数中self.name = name 的含义：将局部变量name的数值发送给当前时刻正在创建的对象中的name成员   
示例程序：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

class Student():

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

print("当前对象的地址是:%s"%self)

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

student1 = Student()

student2 = Student()

[复制代码](javascript:void(0);)

运行结果：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

class Student():

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

print("当前对象的地址是:%s"%self)

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

student1 = Student()

print(student1)

student2 = Student()

print(student2)

[复制代码](javascript:void(0);)

运行结果：

当前对象的地址是:<\_\_main\_\_.Student object at 0x00000000025ACF28>

<\_\_main\_\_.Student object at 0x00000000025ACF28>

当前对象的地址是:<\_\_main\_\_.Student object at 0x00000000025D4048>

<\_\_main\_\_.Student object at 0x00000000025D4048>

Process finished with exit code 0

在上面的程序中，student1、student2、self实际上都是指针变量，存放的是地址，指定当前时刻正在调用的那个对象。

**6、继承的概念**

1、一个类从已有的类那里获得其已有的属性与方法，这种现象叫做类的继承   
2、方法重写指在子类中重新定义父类中已有的方法，这中现象叫做方法的重写   
3、若A类继承了B类，则aa对象既是A，又是B，继承反映的是一种谁是谁的关系，只有在谁是谁的情况下,才能用继承解决代码冗余的问题。   
4、寻找属性和方法的顺序问题：先从对象自己的命名空间中找，然后在自己的类中，最后在从父类当中去找   
5、在python3当中，所有的类都是新式，所有的类都直接或者间接的继承了Object   
6、在python中，新建的类可以继承一个或多个父类

示例代码1：继承和方法重写

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

class People:

def \_\_init\_\_(self,name,age,sex):

self.name = name

self.age = age

self.sex = sex

def tell(self):

print("%s-%s-%s"%(self.name,self.age,self.sex))

class Student(People):

def \_\_init\_\_(self,name,age,sex,salary):

# self.name = name

# self.age = age

# self.sex = sex

People.\_\_init\_\_(self,name,age,sex)

self.salary = salary

def tell(self):

print("%s是最棒的!"%self.name)

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

student = Student("alex",20,"man",2000)

student.tell()

[复制代码](javascript:void(0);)

运行结果：

alex是最棒的!

Process finished with exit code 0

代码示例2：属性的搜索顺序问题

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

class People:

def \_\_init\_\_(self,name,age):

self.name = name

self.age = age

self.tell()

def tell(self):

print("%s---%s"%(self.name,self.age))

class Student(People):

def tell(self):

print("呵呵!")

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

student = Student("alex",20)

[复制代码](javascript:void(0);)

运行结果：

呵呵

示例代码3：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

class People:

def \_\_init\_\_(self,name,age):

self.name = name

self.age = age

self.tell()

def tell(self):

print("%s---%s"%(self.name,self.age))

class Student(People):

def tell(self):

print("呵呵!")

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

student = Student("alex",20)

#查看Student所有的父类

print(Student.\_\_bases\_\_)

#查看最近的父类

print(Student.\_\_base\_\_)

#student既是Student类,又是People类

print(isinstance(student,Student))

print(isinstance(student,People))

[复制代码](javascript:void(0);)

运行结果：

[复制代码](javascript:void(0);)

呵呵!

(<class '\_\_main\_\_.People'>,)

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

True

True

Process finished with exit code 0

[复制代码](javascript:void(0);)

**7、组合的概念**

1、一个类的属性可以是一个类对象，通常情况下在一个类里面很少定义一个对象就是它本身,实际意义很少   
2、将另外一个对象作为自己的属性成员（自己的一个属性来自于另外一个对象），这就是组合   
3、组合也可以解决代码冗余的问题，但是组合反应的是一种什么是什么的关系。   
示例代码1：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

class Date:

def \_\_init\_\_(self,year,month,day):

self.year = year

self.month = month

self.day = day

def tell(self):

print("%s--%s--%s"%(self.year,self.month,self.day))

class People:

def \_\_init\_\_(self,name,age):

self.name = name

self.age = age

class Student(People):

def \_\_init\_\_(self,name,age,sex,year,month,day):

People.\_\_init\_\_(self,name,age)

self.sex = sex

#下面这一步骤就是组合

self.birth = Date(year,month,day)

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

student = Student("alex",25,"man",2015,12,31)

print("student的birth成员指向了一个Date对象!")

print("%s"%student.birth)

student.birth.tell()

[复制代码](javascript:void(0);)

运行结果：

student的birth成员指向了一个Date对象!

<\_\_main\_\_.Date object at 0x0000000002604358>

2015--12--31

Process finished with exit code 0

**8、接口的概念**

1、通过接口可以实现不相关类的相同行为，可以起到一个标志的作用.   
2、接口提供了不同的类进行相互协作的平台   
3、在python中根本就没有一个叫做interface的关键字，如果非要去模仿接口的概念，可以借助第三方模块   
4、raise：主动抛出异常,本来没有错,主动抛出错。   
raise TypeError(“类型错误”)   
示例程序：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

#模拟Java中接口的概念

class S1:

def read(self):

raise TypeError("类型错误")

def write(self):

raise TypeError("类型错误")

class S2(S1):

def read(self):

print("from S2")

def write(self):

print("from S2")

class S3(S1):

def read(self):

print("from S3")

def write(self):

print("from S3")

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

s2 = S2()

s2.read()

s3 = S3()

s3.read()

[复制代码](javascript:void(0);)

运行结果：

from S2

from S3

Process finished with exit code 0

在上面的程序中存在着一个问题，在S2和S3中如果不实现read和write方法，仍然可以实例化，如何解决，看抽象类的概念。

**9、抽象类的概念**

抽象类是为了更好的对类加以分类,抽象类通常情况下是作为一个类族的最顶层的父类,如植物,并用最底层的类来描述现实世界中的具体的事物.

1>Python中抽象方法定义的方式：利用abc模块实现抽象类，在Java当中如果一个方法没有执行体就叫做抽象方法，而在Python中不是以执行体的有无作为标准，而是以一个方法是否有@abc.abstractmethod装饰器作为标准，有则是抽象方法   
2>抽象方法通过子类的实现可以变成普通的方法   
3>抽象方法不存在所谓重写的问题,却存在着实现的问题   
4>含有抽象方法的类一定是抽象类,但是抽象类不一定含有抽象方法,此时也就没有什么意义了   
5>抽象类是一个介于类和接口直接的一个概念，同时具备类和接口的部分特性，可以用来实现归一化设计   
示例程序：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-  
  
#抽象类加抽象方法就等于面向对象编程中的接口

import abc

class File(metaclass=abc.ABCMeta):

@abc.abstractmethod

def read(self):

pass

#抽象类中可以有普通方法

def write(self):

print("11111")

class B(File):

#如果写pass,也是可以的,此时子类将会覆盖掉父类

def read(self):

pass

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

bb = B()

bb.read()

bb.write()

[复制代码](javascript:void(0);)

运行结果：

11111

Process finished with exit code 0

**10、属性与方法的遍历问题（MRO列表）**

python到底是如何实现继承的，对于你定义的每一个类，python会计算出一个方法解析顺序(MRO)列表，这个MRO列表就是一个简单的所有基类的线性顺序列表，例如：

>>> F.mro() #等同于F.\_\_mro\_\_

[<class '\_\_main\_\_.F'>, <class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.E'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]

为了实现继承,python会在MRO列表上从左到右开始查找基类,直到找到第一个匹配这个属性的类为止。   
而这个MRO列表的构造是通过一个C3线性化算法来实现的。我们不去深究这个算法的数学原理,它实际上就是合并所有父类的MRO列表并遵循如下三条准则:   
1.子类会先于父类被检查   
2.多个父类会根据它们在列表中的顺序被检查   
3.如果对下一个类存在两个合法的选择,选择第一个父类   
示例程序：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

class A:

def fun(self):

print("aaaa")

class B(A):

def fun(self):

print("bbbb")

class C:

def fun(self):

print("cccc")

class D(C):

def fun(self):

print("dddd")

class F(B,D):

def fun(self):

print("ffff")

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

print(F.mro())

#F==>B==>A==>D==>C===>Object

ff = F()

ff.fun()

[复制代码](javascript:void(0);)

运行结果：

[<class '\_\_main\_\_.F'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.D'>, <class '\_\_main\_\_.C'>, <class 'object'>]

ffff

Process finished with exit code 0

**11、super关键字的使用**

1、super关键字产生的原因：在子类当中可以通过使用super关键字来调用父类的中相应的方法，简化代码。   
2、使用super调用的所有属性，都是从MRO列表当前的位置往后找，千万不要通过看代码去找继承关系，一定要看MRO列表。   
示例代码：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

class Foo:

def test(self):

print("from foo")

class Bar(Foo):

def test(self):

#Foo.test(self)

super().test()

print("bar")

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

bb = Bar()

bb.test()

[复制代码](javascript:void(0);)

运行结果：

from foo

bar

Process finished with exit code 0

**12、多态的概念**

1、所谓多态指的是一个父类的引用既可以指向父类的对象，也可以指向子类的对象，它可以根据当前时刻指向的不同，自动调用不同对象的方法，这就是多态的概念。（当然，Python中的多态没必要理解的这么复杂，因为Python自带多态的性能）   
2、多态性依赖于同一种事物的不同种形态   
3、Python是一门弱类型的语言,所谓弱类型语言指的是对参数没有类型限制，而这是我们可以随意传入对象的根本原因   
示例程序：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

#模拟Java中接口的概念

class People:

def fun(self):

print("1111")

class Student(People):

def fun(self):

print("2222")

class Teacher(People):

def fun(self):

print("3333")

def g(aa):

aa.fun()

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

print("只要传入的对象是people的子类即可")

g(People())

g(Student())

g(Teacher())

[复制代码](javascript:void(0);)

运行结果：

只要传入的对象是people的子类即可

1111

2222

3333

Process finished with exit code 0

**13、封装的概念**

1、在面向对象中,所有的类通常情况下很少让外部类直接访问类内部的属性和方法,而是向外部类提供一些按钮,对其内部的成员进行访问,以保证程序的安全性，这就是封装   
2、在python中用双下划线的方式实现隐藏属性，即实现封装   
3、访问控制符的用法\_\_\_包括两种:在类的内部与在类的外部   
1>在一个类的内部,所有的成员之间彼此之间都可以进行相互访问,访问控制符\_\_是透明的,失效的   
2>在一个类的外部,通过\_类名\_对象的方式才可以访问到对象中的\_成员   
综上:内部之间可以直接访问，在类的外部必须换一种语法方式进行访问   
4、在python当中如何实现一个隐藏的效果呢？答案：在Python里面没有真正意义上的隐藏,只能   
从语法级别去实现这件事。   
5、在子类定义的\_\_x不会覆盖在父类定义的\_\_x，因为子类中变形成了：子类名\_\_x,而父类中变形成了：父类名\_\_x，即双下滑线开头的属性在继承给子类时，子类是无法覆盖的   
示例程序1：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

class Student:

def \_\_init\_\_(self,name,age):

self.\_\_name = name

self.\_\_age = age

def setter(self,name,age):

if not isinstance(name,str):

raise TypeError("名字必须是字符串类型")

if not isinstance(age,int):

raise TypeError("年龄必须是整数类型")

self.\_\_name = name

self.\_\_age = age

def tell(self):

print("学生的信息是:%s\t%s"%(self.\_\_name,self.\_\_age))

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

student = Student("Alex",25)

student.tell()

student.setter("Alex\_sb",40)

student.tell()

[复制代码](javascript:void(0);)

运行结果：

学生的信息是:Alex 25

学生的信息是:Alex\_sb 40

Process finished with exit code 0

**14、@property的用法**

1、property是一种特殊的属性，访问它时会执行一段功能（函数）然后返回值   
2、将一个类的函数定义成特性以后，对象再去使用的时候obj.name,根本无法察觉自己的name是执行了一个函数然后计算出来的，这种特性的使用方式遵循了统一访问的原则（使用的原因）   
3、一旦给函数加上一个装饰器@property,调用函数的时候不用加括号就可以直接调用函数了   
代码示例：

[复制代码](javascript:void(0);)

class Student:

@property

def fun(self):

print("1111111")

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

student = Student()

student.fun

[复制代码](javascript:void(0);)

示例程序2：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

class Student:

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

self.\_\_name = name

@property

def name(self):

return self.\_\_name

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

student = Student("alex")

student.\_Student\_\_name = "sb\_alex"

print(student.name)

###本节需要补

[复制代码](javascript:void(0);)

**15、绑定方法与非绑定方法**

绑定方法的使用：   
1、在类内部定义的方法,在没有被任何装饰器修饰的情况下,就是为了绑定到对象给对象用的,self关键字含有自动传值的过程,不管写不写self关键子。   
2、默认情况下,在类内部定义的方法都是绑定到对象的方法。   
3、绑定方法绑定到谁的身上，谁就作为第一个参数进行传入。   
4、绑定到类的方法给对象使用是没有任何意义的。

非绑定方法：   
statimethod不与类或对象绑定，谁都可以调用，没有自动传值效果，python为我们内置了函数staticmethod来把类中的函数定义成静态方法   
不与类或对象绑定，类和对象都可以调用，但是没有自动传值那么一说。就是一个普通工具而已   
只有绑定方法才存在自动传值的说法。

总结：   
绑定到对象的方法,调用的时候会将对象参数自动传入, ===>方法上面什么也不加   
绑定到类的方法,调用的时候会将类作为参数自动传入, ===>方法上面加classmethod   
非绑定方法不与类或对象绑定，类和对象都可以调用，但是没有自动传值那么一说。===>static

使用场景：   
看使用什么调用？类对象or类or什么参数都不需要.

模拟数据库登陆的场景。   
(只有类才有实例化的说法)

创建数据库的时候加上一个id属性,指定是哪一个链接。   
每次数据库实例话的时候都要赋值一个id。   
示例程序1：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

#造成id的方式,用hash算法

import time

import hashlib

def create\_id():

m = hashlib.md5(str(time.clock()).encode("utf-8"))

return m.hexdigest()

#time.clock()计算的是cpu真实的时间

print(create\_id())

print(create\_id())

print(create\_id())

print(create\_id())

[复制代码](javascript:void(0);)

运行结果：

3a92235c44873cbcf618a132a2781157

aa0968a1f467ba19f1595c70efb2c3ee

abc4e2f842e0283b2186d459c069d301

28e76bcb1724833a847851e8f2286e65

Process finished with exit code 0

示例程序2：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

import settings

import time

import hashlib

class MySQL:

def \_\_init\_\_(self,host, port):

self.id = self.create\_id()

self.host = host

self.port = port

print("connecting.....")

def select(self): # 绑定到对象的方法

print(self)

print("select function")

# 绑定到类的方法,从配置文件中获取主机名和端口号,用户默认链接数据库的一种方式

@classmethod

def from\_conf(cls):

# 实例话的结果得到了一个类对象

# 通过绑定对象的方法间接的去创建了一个类对象

return cls(settings.HOST, settings.PORT) # 相当于MySQL("127.1.1.1",3306)

#工具包既不依赖于类,也不依赖于对象

#非绑定方法就是类中普通的工具包,不依赖于self和cls的参数

@staticmethod

def create\_id():

m = hashlib.md5(str(time.clock()).encode("utf-8"))

return m.hexdigest()

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

conn = MySQL("192.168.80.100", 3306)

conn.select()

conn2 = MySQL.from\_conf()

conn2.select()

print(conn.id)

print(conn2.id)

[复制代码](javascript:void(0);)

运行结果：

[复制代码](javascript:void(0);)

connecting.....

<\_\_main\_\_.MySQL object at 0x0000000002584438>

select function

connecting.....

<\_\_main\_\_.MySQL object at 0x0000000002584EB8>

select function

3a92235c44873cbcf618a132a2781157

294e6de59d0e8d4a75717d5bb20f92e0

Process finished with exit code 0

[复制代码](javascript:void(0);)

**16、staticmethod与classmethod的区别**

staticmethod与classmethod的区别：前者是非绑定方法,后者是绑定到类的方法   
示例程序1：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

import settings

import time

import hashlib

class MySQL:

def \_\_init\_\_(self,host, port):

self.host = host

self.port = port

print("connecting.....")

@staticmethod

def from\_conf():

return MySQL(settings.HOST,settings.PORT) #相当于MySQL("127.1.1.1",3306)

def \_\_str\_\_(self):

return "父类"

class Mariab(MySQL):

def \_\_str\_\_(self):

return "子类"

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

conn = MySQL.from\_conf()

print(conn.host)

conn1 = Mariab.from\_conf()

print(conn1.host)

#本来想获取Mariab的一个对象,但是现在获取的是MySQL的一个对象,这是子类继承的一个问题

print(conn1)

[复制代码](javascript:void(0);)

运行结果：

[复制代码](javascript:void(0);)

connecting.....

127.1.1.1

connecting.....

127.1.1.1

父类

Process finished with exit code 0

[复制代码](javascript:void(0);)

示例程序2：基于1的改进

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

import settings

import time

import hashlib

class MySQL:

def \_\_init\_\_(self,host, port):

self.host = host

self.port = port

print("connecting.....")

@classmethod

def from\_conf(cls):

return cls(settings.HOST,settings.PORT) #相当于MySQL("127.1.1.1",3306)

def \_\_str\_\_(self):

return "父类"

class Mariab(MySQL):

def \_\_str\_\_(self):

return "子类"

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

conn = MySQL.from\_conf()

print(conn.host)

conn1 = Mariab.from\_conf()

print(conn1.host)

#本来想获取Mariab的一个对象,但是现在获取的是MySQL的一个对象,这是子类继承的一个问题

print(conn1)

[复制代码](javascript:void(0);)

运行结果：

[复制代码](javascript:void(0);)

connecting.....

127.1.1.1

connecting.....

127.1.1.1

子类

Process finished with exit code 0

[复制代码](javascript:void(0);)

**17、总和应用的一个小例子**

要求：   
定义MySQL类   
1.对象有id、host、port三个属性   
2.定义工具create\_id，在实例化时为每个对象随机生成id，保证id唯一   
3.提供两种实例化方式，方式一：用户传入host和port 方式二：从配置文件中读取host和port进行实例化   
4.为对象定制方法，save和get，save能自动将对象序列化到文件中，文件名为id号，文件路径为配置文件中DB\_PATH;get方法用来从文件中反序列化出对象。   
代码示例：

[复制代码](javascript:void(0);)

#!/usr/bin/python

# -\*- coding:utf-8 -\*-

import time

import hashlib

import settings

import random

import pickle

import os

"""

HOST = "127.1.1.1"

PORT = 3306

DB\_PATH = r"D:\Python Work Location\Python 0507\day07\db"

"""

class MySQL:

@staticmethod

def create\_id():

m = hashlib.md5(str(time.clock()).encode("utf-8"))

return m.hexdigest()

def \_\_init\_\_(self,host,port):

#为每一个对象创建了一个ID

self.id = self.create\_id()

self.host = host

self.port = port

#从配置文件中读取在这里用到了classmethod

@classmethod

def from\_conf(cls):

return cls(settings.HOST,settings.PORT)

def save(self):

file\_path = r"%s%s%s"%(settings.DB\_PATH,os.sep,self.id)

#将这个对象以二进制的形式写到硬盘当中

pickle.dump(self,open(file\_path,"wb"))

def get(self):

#在这里面通过id的方式保证了文件的名字是唯一的

file\_path = r"%s%s%s" % (settings.DB\_PATH, os.sep, self.id)

return pickle.load(open(file\_path,"rb"))

#对于Python中的对象json无法进行序列化

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

conn1 = MySQL("172.1.2.1","3306")

print(conn1.id)

conn1.save()

result = conn1.get()

print(result.id)

#通过os.listdir命令可以浏览某一个目录下面有哪些文件,然后循环的反序列化

print(os.listdir(r"D:\Python Work Location\Python 0507\day07\db"))

[复制代码](javascript:void(0);)

运行结果：

85b8a467159e14ec4b2d16bff39ed199

85b8a467159e14ec4b2d16bff39ed199

['85b8a467159e14ec4b2d16bff39ed199', 'd0cff574feed705df3655a209c79c7ef']

Process finished with exit code 0