**第11章 面向对象（下）**

Python中的面向对象功能来自于C++和Modula-3中的类机制。Python的类提供了面向对 象编程的所有标准功能：类继承机制允许多重继承，子类可以覆盖其父类的任意方法或嵌套类， 方法能够以相同的名称调用父类中的方法。对象可以包含任意数量和种类的数据。

111 WliWBB三大待fiE

面向对象开发思想具有三个典型特征：封装、继承和多态。

仏封裝 .\*

封装是面向对象方法的一个重要原则，把对象的属性和行为封装起来，不需要让外界关心 内部的具体实现细节。使用封装能隐藏对象的实现细节，使代码更容易维护，同时因为不能直 接调用、修改内部的私有信息，在一定程度上保证了系统的安全。

接下来我们举一个关于封装的例子。例如,我们定义了一个圆形类（Circle） 9它定义了 一个计算圆面积的方法（get\_area） ?那么当需要计算某一个圆形A的面积时，只需要调用 Circle类的get\_area方法，并将圆形A的半径值传入，即可得到A的面积，而不需要关心它的 面积是如何计算出来的。

2继承

面向对象编程语言的一个主要功能就是继承，继承主要描述的是类与类的关系，它可以在 不必重写类的情况下，对原有类的功能进行扩展。例如，有一个轮船类，它具有吨位、时速、 吃水线等属性，具备行驶、停泊等行为，客轮类继承自轮船类歹它会拥有轮船类的全部属性和 行为，还可以增加客轮类自己的特殊属性（如载客量）和行为（如供餐）。

继承不仅增强了代码的复用性，还提高了开发效率，而且为程序后期的维护提供了便利。 3n爹窓

多态与继承紧密相关，是面向对象编程中另一个突岀的特征。对象的多态性是指在父类中 定义的属性或方法被子类继承之后，可以使同一个属性或方法在父类及其各个子类中具有不同 的含义，这称为多态性。例如，动物都有吃饭的方法，但是老鼠的吃饭方法和猫的吃饭方法是

截然不同的。简单来说就是，一种行为产生多种效果。

综上所述，封装可以隐藏实现细节同时包含私有成员，使得代码模块化并增加安全指数； 继承可以扩展已存在的模块，并且可以达到代码重用的效果；多态则是为了保证类在继承和重 写的时候，继承体系中任何类的实例都被正确调用，实现了接口重用。

J 皿。2 MW

通常把隐藏属性、方法与方法实现细节的过程称为封装。为了保护类里面的属性，避免外 界随意赋值，可以采用如下方式解决：

。把属性定义为私有属性，即在属性名的前面加上两个下画线；

•添加可以供外界调用的两个方法，分别用于设置或者获取属性值。

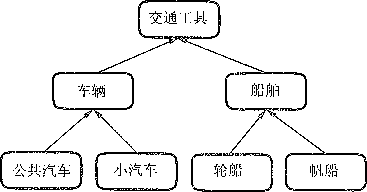
具体案例可参见第1047小节“保护对象的私有属性”的内容。

封装的另一个用途是保护类的方法。为了保护类的方法，可以把它设置为私有方法，只在 类内部可以访问。

I 1L3 «

篥醴鬣

现实世界中的事物是有多种联系的，通常将具有相同特征和行为的事物划分为一个种类， 如动物、植物、水果、人类等。相同种类的事物之间也存在着各种关系,从属关系就是其中的一种O 比如交通工具包含了车辆和船舶，车辆又包含了公共汽车和小汽车，船舶又包含了轮船和帆船 等。接下来，通过一张图来描述，具体如图11』所示。



图交通工具层级图

图反映了交通工具相关的层级关系，从交通工具到小汽车，向下经历了3个层级。低 层级的交通工具具备了高层级交通工具的所有特性和行为，并且还增加了高层级不具备的，自 己特有的特征和行为。所有的小汽车都是车辆，但是车辆却不一定是小汽车。从高层级到低层 级是一个从抽象到具体的过程，从低层级到高层级则是一个从具体到抽象的过程。

在面向对象编程里，所有的事物都对应着一个类，包括交通工具类、车辆类、公共汽车类、 小汽车类等。每个事物都有自己的特征和行为，对应着类的属性和方法。

一个类可以继承自另一个类身并自动拥有了另一个类的属性和方法，并且可以进一步完善, 添加新的特性和方法。当一个类继承自其他类时，继承类叫作子类，被继承类叫作超类或者父类。 例如,类B继承自类A,则类B为子类,类A叫作超类或者父类。

在Python程序中，继承使用如下语法格式标注:

class子类名(父类名):

假设有两个类，类A和类玖 类R继承自类A，示例如下：

class A(object):

class B(A):

如果在定义类的时候没有标注,则默认是继承自object的9例如class Person(object)和 class Person()两者是等价的。

下面的代码定义了一个父类车辆类(Vehicle)和子类小汽车类(Car):

|  |  |
| --- | --- |
| >>> class Vehicle(object):  •.。 def init (selfz tyres=4):  o。\* self»tyres = 4  "。» def run(self):  …o print (”一 开动 一n) | #定义一个车辆类  #构造方法  #轮胎数量  #定义跑的方法 |
| >>> class Car(Vehicle): | # 定义子类Car,继承自Vehicle |
| « .。 pass |  |

上述代码中定义的类Vehicle有一个构造方法,在构造方法中定义了一个对象属性tyres, 还定义了一个方法run(),该方法中输岀了一个字符串。

然后定义了一个类Car,继承自Vehicle类。虽然Car没有新增任何方法和属性，但是它继 承了 Vehicle的构造方法,并且继承了 tyres属性和run()方法。

接下来创建一个myCar对象,并访问它的tyres属性和run()方法，代码如下:

>>> my\_car = Car()

>>> my\_car.run()

一开动一

>>> print(my\_car.tyres)

4

需要说明的是，子类不能继承父类的私有属性和方法，也不能在子类中直接访问。例如， 我们给Vehicle类设置一个私有属性和方法,同样让Bus类继承Vehicle,此时,在子类中访问 父类的私有属性或者方法是不可行的。

|  |  |  |  |
| --- | --- | --- | --- |
| >>> class Vehicle(object): | | # | 定义父类 |
| .… def | \_init\_(self, color = “黑色旳) |  |  |
|  | self. color = ° M & " |  |  |
| … def | get\_vehicle\_color(self): | # | 定义方法在方法中访问私有属性 |
|  | print(self. color) |  |  |
| … def | \_add\_oil(self): | # | 定义私有方法 |
|  | print ("—加油—°) |  | \* |
| >>> class Bus(Vehicle): | | # | 定义子类 |
| …• def | get\_bus\_\_color (self): | # | 在子类方法中访问父类的私有属性 |

print(se1f o color)

在上述代码中，定义了一个类Vehicle,它的属性\_color是私有的,方法\_add\_oil()是私有的, 方法get\_vehicle\_color()是共有的。在get\_bus\_color()方法中打印了私有属性 color的值。

接着定义了 Vehicle类的子类Bus,它定义了一个方法get\_bus\_color(),在该方法中访问了 父类的\_\_color属性。

此时，如果使用子类对象访问父类中的属性和方法，程序会报错，具体如下：

>>> bus = Bus()

>>> bus » get\_bus\_color()

Traceback (most recent call last):

File "<stdin>,? z line 1 z in <module>

File n<stdin>"7 line 37 in getBusColor

AttributeError: 1 Bus \* obj ect has no attribute 5\_Bus color ? »> bus add\_oil ()

Traceback (most recent call last):

File "<stdin>°, line 1, in <module>

AttributeError: 'Bus' obj ect has no attribute f add\_\_oil \* >>> bus o get\_vehicle\_color()

黑色 — —

从运行结果可以看出，bus.get\_bus\_color()出错，说明子类对象无权直接访问父类私有属性; bus. add\_oil()出错,说明子类对象无权直接访问父类的私有方法;只有bus.get\_vehicle\_color() 运行成功，说明子类对象可以通过调用父类方法的途径访问父类私有属性。

我们知道私有属性和方法是被系统改名了，实际上子类也继承了父类的私有元素，子类对 象通过类名 私有元素”可以访问父类的私有属性和私有方法，但是这种访问并不建议。 一般情况下，私有的属性和方法都是不对外公布的，只能用来做其内部的事情。

11-3-2 ® fesybdassf)函数

Python提供了两个和继承相关的函数,分别是isinstance()函数和issubclass()函数。

们 SsinstanceO 函数

isinstance(o5t)函数用于检查对象的类型，它有2个参数，第1个参数是要判断类型的对象 (0),第2个参数是类型(t),如果。是t类型的对象，则函数返回Tnw,否则返回珂屁。例如：

>>> isinstance(bus}Bus)

True

2 issubc!ass()函数

函数issubclass(cls? classinfo)用于检查类的继承关系,它也有2个参数,第1个参数是要判 断的子类类型(cis) 9第2个参数是要判断的父类类型(classinfo),如果cis类型是classinfo 类型的子类,则函数返回True,否则返回Falseo例如:

>>> issubclass(BusrVehicle)

True

如果一个子类只能继承一个父类，这叫作单继承。但 有时，一个子类可以继承自多个不同的父类，这种情况叫 作多继承。

［二If量二］ 车辆

I 55 I "行驶

X X

./■

' 房车 f  
mm

，仟驶

图11-2多继承关系举例

现实生活中，一个子类可能会有多个父类。比如在职 研究生，既是工作者，又是学生。又比如房车，既具有房 屋的居住功能，又具有车辆的行驶功能。多继承的关系举 例如图1L2所示。

从图lb2中可以看出，房车同时拥有了房屋和车辆的特征。

Python语言是支持多继承的，它可以看作单继承的扩展，多继承的基本语法格式如下： class子类(父类1,父类2…)

一旦形成了多继承，那么子类会有多个父类，也就意味着子类同时拥有了多个父类的属性 和方法。

接下来我们创建房车类(RV),让它继承自房屋类(House)和车辆类(Car)两个父类， 代码如下：

class House(): def live(self):

print (房子可以居住)

#定义房屋类

»>

class Car():

>>>

»>

def move(self):

print (汽车可以行驶) class RV(Housef Car):

pass

上述代码中，父类House(房屋)中有一个iive()方法,父类Car(车辆)中有一个move()方法, 子类RV (房车)同时继承了 House类和Car类。

#定义车辆类

#定义房车类

接下来创建一个RV类型的对象rv,调用它继承来的live()方法和move()方法,代码如下:

»> rv = RV( ) #创建房车对象

>>> rv。live()

--房子可以居住--

>>> rv.move()

一汽车可以行驶一

从运行结果可知，子类的对象同时继承了两个父类的方法。

那么问题来了：如果子类的多个父类有一个同名的方法，那么子类的对象继承来的是哪个父 类的方法呢？我们来做个测试,给上述的House类和Car类都添加一个gd\_ook)r()方法，代码如下:

House 类:

def get\_color(self): print ("白色的房子")

Car 类:

def get\_color(self):

print (°红色的汽车n )

然后让rv对象调用get\_color()方法,代码如下:

>>> rv = RV()

»> rv. live ()

—房子可以居住――

»> rv »move ()

—汽车可以行驶

>» rv. get\_color ()

白色的房子

从运行结果可以看岀,rv对象调用的是父类House的get\_color()方法。

实际上，在Python 3中，如果子类继承的多个父类间是平行的关系，子类先继承的哪个类 就会调用哪个类的方法。如果当前类的继承关系非常复杂，Python会使用iwo算法找到合适的 类，算法专门用于在多继承时判断调用的属性和方法的路径，即来自于哪个类。可以通过 类名访问\_mro()\_属性的方式查看该类搜索属性和方法时的先后顺序。

如果多个父类有同名的属性或方法，在子类中要指定使用某个父类的属性或方法时，我们 可以在子类中对该属性或方法名赋值来覆盖继承。下面的例子说明了多继承情况下指定父类元 素的使用。

|  |  |  |
| --- | --- | --- |
| >>> class House(object): | | |
| *…* color = “白色"  •.. def get\_color(self): | | * color 属性 * get\_color 方法 |
|  | print(self»color) |
| »> | class Car(object): |  |
|  | color ="红色 ° | # color属性 |
|  | def get\_\_color (self): | # get\_color 方法 |
|  | print(self.color) |  |
| »> | class RV(House,Car): |  |
| o 。*。* | color = Car.color | #指定使用Car类的color属性 |
|  | get\_\_color = Car . get\_color | #指定使用Car类的get\_color方法 |

上述代码中,定义了两个父类House和Car,它们有一个同名属性color和一个同名方法 get\_color()o子类RV同时继承了 House类和C取类,但是它在定义时指定了使用Car类的属性 和方法。

接下来创建一个rv对象，并调用它的color属性和get\_color方法。代码如下所示:

»> rv = RV()

»> print (rv, color)

红色

»> rv. get\_color ()

红色 ~

从运行结果看岀，显式指定了父类的属性和方法后，就可以忽略继承顺序，直接访问指定

第11章面向対象(下)Q 229 父类的属性和方法。

11d3d4 囂膏 • ’ \* '

子类可以继承父类的属性和方法，但是有些情况下，子类希望对父类提供的属性和方法进 行修改，提供自己的实现，这种行为就叫重写(overriding)。

下面,我们使用子类InServiceStudent (在职生)重写父类Student (学生)的方法，代码如下。

>> class Student(object): def main (self) : # 定义父类方法

… print (n学习卵)

>>> class InServiceStudent(Student):

…• def main (self) : #子类重写父类方法

…o print (??—边工作一边学习° )

>>> in\_service\_student = InServiceStudent()

>>> in\_service\_student.main() 边工作一'边学习

父类Student有一个main()方法,子类InServiceStudent重写了 main()方法。从运行结果可 以看出9在in service student对象调用main()方法时,调用的是子类的main()方法。 ［泠注意：， ： .

*Jr*

重写后的方法必须与父类的方法名称和参数个数相同。

11d3d5 super •

在子类重写了父类的方法后，如果还需要访问父类的同名方法，可以使用Slip就关键字。 super关键字用于在子类中访问父类的方法。

使用supe『关键字的经典场合是在\_\_init()\_方法中。在Python中，如果子类重写了 \_inif()\_构造方法，子类并不会自动调用父类的构造方法。但是子类通常需要父类的构造方法， 包括在构造方法里定义的属性，或者初始化的设置等，所以在子类重写的\_init()\_方法中， 要调用父类的\_init()\_方法。

下面我们看一个例子，具体如下：

»> class Father (object):

•… def init (self) : #父类构造方法

print (°父类构造方法")

>>> class Son(Father):

…. def init (self) : #子类构造方法

... print ("子类构造方法")

在上述代码中,定义了父类Father和子类Son,子类Son重写了 \_init()\_构造方法。

接下来创建一个son对象，代码如下所示。

>>> son = Son()

子类构造方法

从运行结果可以看出，子类重写构造方法以后，并没有调用父类的构造方法。

如果子类需要在构造方法中增加自己的属性，或者扩充其他功能，而需要重写构造方法的 话，一定要记住在最后调用父类的构造方法。比如上述代码中的Son类的构造方法，可以更改为:

>>> class Son (Father):

*.。*. def init (self):

.o . super() . init ();

…。 print (n子类构造方法")

再次创建son对象，代码如下所示。

>>> son = Son ()

父类构造方法

子类构造方法

从运行结果就可以看到，子类调用了父类的构造方法。

1L4多态

多态，是面向对象的程序设计语言最核心的特征，如果一个语言不具备多态特性，那么就 不能称之为面向对象的语言。

在Python中，多态指在不考虑对象类型的情况下使用对象。相比于强类型,Python更推崇“鸭 子类型” O “鸭子类型”是这样推断的：如果一只生物走起路来像鸭子，游起泳来像鸭子，叫 起来也像鸭子，那么它就可以被当作鸭子。也就是说，它不关注对象的类型，而是关注对象具 有的行为。

由于Python是动态类型语言，所以多态随处可见。Python中使用对象，并不需要显式指 定对象的类型，只要对象具有预期的方法和表达式操作符，就可以使用。也可以说，只要对象 支持所预期的“接口”，就可以使用。这使得PythcMi语言使用起来更简易和灵活。

|  |  |  |
| --- | --- | --- |
| 看一个多态的案例，具体如下：  >>> class Animal(object): | | #定义父类 |
|  | def move(self):  pass |
| »> | class Rabbit(Animal):  def move(self):  print ("兔子蹦蹦跳跳") | #定义子类Rabbit |
| »> | class Snail(Animal):  def move(self):  print (°蜗牛缓慢爬行") | #定义子类Snail |
| »> | def test (obj): obj.move() | #test ()函数中调用了对象的move ()方法 |

上述代码创建了 Animal类和它的两个子类Rabbit类和Snail类，它们都有move()方法。 创建函数test()9它接收一个参数obj,并在函数体内调用了 obj的m(g()方法。

接下来，分别创建了 Rabbit类和Sn疝类的对象，作为参数传入test()函数。代码如下所示：

>>> rabbit = Rabbit () # 创建 Rabbit 对象

>>> test(rabbit)

兔子蹦蹦跳跳

>>> snail = Snail () # 建 Snail 对象

>>> test(snail)

蜗牛缓慢爬行

从运行结果可以看出，根据传入参数的对象不同，调用的是不同类型对象的mov©()方法。

实际上，由于化如)并没有规定参数的类型，所以如果传入其他类型的对象也可以，只要 该对象具有move()方法,而不一定要继承自Animal类。

注意：

在强类型语言中，由于严格限制了变量的类型，所以多态适用于把不同的子类对象都当作父 类来看，可以屏蔽不同子类对象之间的差异。赋值之后，父类类型的对象就可以根据当前赋值给 它的子对象的特性以不同的方式运作，也就是，父亲的行为像儿子，而不是儿子的行为像父亲。

1L5运算简蕾载

在Python里，用户可以在自定义类里通过实现特殊方法来重载內建运算符。类可以重载 所有的表达式运算符，以及打印、函数调用、属性点号运算符等内置运算。重载使得类对象的 行为更像内置类型。在类中提供特殊名称的类方法，可以实现对应运算符的重载。

例如，加法运算符(+ )对应\_add\_方法，当调用+进行加法运算的时候，实际上调用 T \_add\_方法。表114列出了部分运算符重载的方法。

表仲1运算符重载方法(部分)

|  |  |  |
| --- | --- | --- |
| 、 方 法 | ， 说 明 | 何时调用方法 |
| \_\_init\_ | 构造函数 | 创建对象时；X = X(args) |
| \_add | 「lllll'llliyIf | 对象加法:x+y5 x+=y |
| \_sub\_ | 减法运算 | 对象减法:x-y, x-=y |
| \_mul\_ | 乘法运算 | 对象乘法：x\*y, x\*=y |
| \_truediv\_ | 除法运算 | 对象除法:x/y, x/=y |
| \_floordiv |  | 对象地板除法:x//y, x//=y |
| \_mod\_ | 求余运算 | 对象求余:x%y, x%=y |
| \_bool\_ ' | 真值测试 | 测试对象是否为真值:bool (x) |
| \_\_repr\_*、* \_str\_\_ | 打印、转换 | print *(x)、*repr (x)、str (x) |
| \_\_contains\_ | 成员测试 | item in x |

续表

|  |  |  |
| --- | --- | --- |
| 方 法 | 说 明 | 何时调用方法 |
| \_ etitem\_ | 索引、分片 | x[i]、x[i:j]、没有 \_iter\_ 的 for 循环等 |
| \_setitem\_\_ | 索引赋值 | x[i]=值、x[i:j]=序列对象 |
| \_delitem\_ | 索引和分片删除 | del x[i]、del x[i:j] |
| \_len\_\_ | 求长度 | len (x) |
| \_iter\_*、* \_next\_\_ | 迭代 | iter (x)、next (x)、for 循环等 |
| \_\_ne\_ | 相等测试、不等测试 | x=y、x!=y |
| \_ge\_. | 大于等于测试、大于测试 | x>=y、x>y |
|  | 小于等于测试、小于测试 | x<=y、x<y |

11.5J

四则运算加、减、乘、除(+, -, \*, /)分别是通过调用―add\_、\_sub\_. \_mul\_和 truRdiv\_方法完成重载的。当两个实例对象执行四则运算时，自动调用的是它们对应的方法。

现在我们创建一个简单的计算器(Calculator*)类,*它能执行四则运算，并且把运算结果记

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 录下来， | 作为下一次运算的起始数字，代码如下所示。 | | | |
| »> | class Calculator(): | |  | #计算器类 |
| 。•。 | def | init (self,number): self.number = number |  | #构造方法 |
|  | def | add (self7 other): |  | #加法运算 |
|  |  | self.number = self.number return self.number | + | other |
|  | def | sub (selfF other): |  | #减法运算 |
|  |  | self.number = self.number return self.number | 一 | other |
|  | def | mul (self7 other): |  | #乘法运算 |
|  |  | self.number = self.number return self»number | 大 | other |
|  | def | \_truediv\_(self7 other): |  | #除法运算 |
|  |  | self.number = self.number return self.number | / | other |

在上述代码中，定义了一个Calculator类，它重载了 \_add\_、\_sub\_、\_mul\_和 truediv\_方法9并将计算结果保存在属性number中。

然后我们创建一个Calculator类的对象，对它进行加减乘除操作。具体代码如下：

»> calculator = Calculator (10)

>>> print(calculator + 10)

20

»> print (calculator - 5)

15

>» print (calculator \* 2)

30

»> print (calculator / 3)

10.0

115.2褰鄭濾翁胯蠶戰

以下是跟索引和分片相关的3个可重载方法：

* getitem :索引、分片取值。
* —setitem :索引、分片赋值。
* \_delitem\_：索引和分片删除。

这三个方法都可以进行重载，具体如下：

在对实例对象执行索引、分片或者fbw迭代操作取值时，会自动调用就血m\_方法，下 面的例子定义了一个记录班级成绩的类ClassScore,在该类中重载 getitem 方法,具体代码 如下。

>>> class ClassScore(object):

#定义班级成绩类

#构造方法

#定义索引”分片运算符重方法

•.• def init (selffnumbers): self. scores = numbers [:]

•.. def getitem (selff index): …• return self.scores[index]

上述代码中定义的ClassScore类在构造方法中添加了 scores属性,用于记录班级所有的成 绩，并重写了索引分片时调用的方法

然后创建一个mathscore对象进行测试，先获取索引为。的值，再通过分片获取部分值， 再使用fo『循环实现迭代。代码如下：

>>> mathscore = ClassScore ( [85z91,*95f*98])

索引返回单个值

>>> print(mathscore[0])

85

分片返回部分值

>>> print(mathscore[1:])

[91, 95, 98]

for循环迭代

for score in mathscore: print(score)

»>

85

91

95

98

2a

setitem 方憑

通过赋值语句给索引或者分片赋值时，调用—setitem \_方法可以实现对序列对象的修改, 下面看一个例子。

>>> class ClassScore(object):

#构造方法

#重载索引、分片赋值运算方法

... def init (self,numbers): self . scores = numbers [:]

... def setitem (self7 index,value): •.• self.scores[index] = value

上述代码中定义的ClassScore类在构造方法中添加了 scores列表，然后重写了索引分片赋 值时调用的方法 setitem 。

接下来创建一个mathscore对象，先显示列表中所有的元素，再通过赋值语句修改列表的 第1个元素，然后把列表中的分片进行替换，分别输出两次修改后的列表。代码如下所示。

>>> mathscore = ClassScore ( [85f 91, 95, 98] ) # 创建实例对象,并用列表初始化

>>> print(mathscore.scores)

[85, 91, 95, 98]

»> mathscore [0] = 100 #修改列表第一个元素

>>> print (mathscore . scores) # 显示修'改后的列表

[100, 91, 95, 98]

»> mathscore [1:3] = [99, 98,97] # 把列表中的分片[M 3]替换为列表[99, 98, 97]

>>> print (mathscore . scores) # 显示修改后的列|表

[100, 99, 98, 97, 98]

11.5.3繼制麺豪鱸寥籥器辭就 ’

重载—str 和 repr—方法可以定义对象转换为字符串的形式,在执行print(). str().

mpH)以及交互模式下直接显示对象时，会调用\_s加\_和\_唧1\_方法。但是只重载某个方法 与把两个方法都重载的效果是不同的。

如果只是重载了 方法，只有print()和str()函数可以调用这个方法进行转换。下面

看一个例子。

>>> class Person: # 定义类

• . ° ' def init (self, name, age) : # 构造方*法*

…。 self . name = name

。。 self o age = age

…” def str (self) : # 重载 str 方法

。… return "我叫%s,今年%s岁"% (self.name7 self. age) #返回自定义字符串

上述代码中定义的Person类重载了 方法，在重载方法中使用return返回自定义的

字符串。

记下来创建一个tom对象，然后分别使用pHiit()、\*()、冀pf()函数输出对象的信息。代码 如下所示。

>>> tom = Person("tom"z 25) >>> print(tom) 我叫tom,今年25岁

»> print (str (tom)) 我叫七om ,今年25岁

>>> print(repr(tom))

< main Person obj ect

调用print输出对象信息

调用str<#转换

调用repr函数,结果显示没有进行转换

at

#

0x00590C90>

2.只蠢载—應p方法 . • ’、

重载\_,泸\_方法，可以保证各种操作下都能正确获得实例对象自定义的字符串形式。现 在我们对上一个案例进行修改，重载方法，具体如下：

>>> class Person: # 定义类

…。 def init (self z name, age) : # 构造方法

\* .o self o name = name

... self o age = age

def —repr—(self) : # 重载 repr *方法*

return "我叫 %s ?今年％s 岁"% (self . namef self . age)

上述代码重写了 \_repr 方法，并使用return返回自定义的字符串。

接下来创建一个tom对象，然后分别使用print()> str()^ r©pr()函数输出对象的信息。代码 如下所示。

>>> tom = Person("Tom"z 25) >>> print(tom)

我叫Tom,今年25岁

>>> print(str(tom))

我叫Torn,今年25*岁*

»> print (repr (tom))

#调用 repr 方法进行转换

我叫Torn,今年25岁

*3.*同蹲重戮或『灘馋方淒

如果同时重载了 str 和 repr 方法，则stT()和print()函数调用的是 str—方法。 交互模式下直接显示对象和repr()函数调用的是\_ropL\_方法，看下面的例子。

>>> class Person: # 定义类

def

init (self, name^ age) : # 构造方法

self.name = name

self.age = age

def

str (self) : # 重载 str 方法

return nstr 转换:我口日 %s今年 岁 °% (self . namef self . age) repr (self) : # 重载 repr 方法

def

return nrepr ##:我叫 %s^ 今年％s 岁 n% (self. name, self . age)

上述代码中,重写的\_str\_方法和\_repr\_方法都返回了自定义的字符串，但是在字符 串中加入了 str或者repi•作为标识。

接下来创建一个fom对象，然后分别使用print(). str().冀p『()函数输出对象的信息。代码 如下所示。

>>> tom = Person("Tom",25)

>>> print(tom)

str转换:我叫Tom,今年25岁

>>> print(str(tom))

str转换:我叫Tom,今年25岁

>>> print(repr(tom))

repr转换:我叫Tom,今年25

ILS \_恥殉錢

Python中object类还有一个內建的方法\_\_new\_9用于创建类的对象。在自定义类时一般 不用重写这个方法，但在特殊场合也会重写(比如下节的单例模式实现中)。

\_iiew\_方法是类的静态方法，即使重写时没有被加上静态方法装饰器。我们看看。可知 类中对 new 方法的定义:

@staticmethod

def new (cisf \*more):

"° ° Create and return a new object。 ""n

pass

关于定义的介绍如下：

* \_new\_\_方法至少要有一个参数cis,代表要实例化的类，此参数在实例化时由Python 解释器自动提供。

« \_\_new\_方法必须要有返回值，返回实例化出来的实例，这点在自己实现\_\_iww\_方 法时要特别注意，可以冀turn父类\_new\_方法返回的实例，或者直接返回object类的 ―new 方法返回的实例。

现在我们借助下面的案例来测试\_\_new\_方法的用法，以及与\_init\_方法的区别。

>>> class Test(object):

。。 def init (self):

… print (卵执行init方法，，)

*…*. print ("self 对象的 id^%s"%id(self))

... def new (cis):

… print ( n执行new方法")

..o demo\_\_obj ect = obj ect. new (cis)

<," . print (°demo\_obj ect 对象的 id 是 %s°%id (demo\_\_obj ect))

... return demo\_obj ect

上述代码中，我们定义了一个继承自object的测试类Te电 为它重写了 方法和

\_new\_方法，在方法中打印了提示信息和创建的对象的ido

接下来创建一个test对象,代码如下所示。

>>> test = Test ()

执行new方法 demo\_object<<^ id< 5836400 执行init方法

seU 对象的id是5836400

从运行结果可以看出，在创建test对象时,首先运行了\_new\_方法，然后才是\_init\_方法。 并且—new 方法返回的对象id与传入 init 方法的对象id相同，证明它们是同一个对象。

—new—方法和—init\_\_方法是有区别的:

* \_new 方法用于创建对象，—init 方法在 new 方法的基础上完成一些其他初始 化的工作，包括设置对象的私有属性等。
* \_而」方法不需要返回值。
* \_>it\_方法的参数Ef,就是\_new\_方法返回的实例。

\_ 1L7单例模式

有些场合需要用到单例，即该类只有一个实例存在，比如工具类、管理类、代表程序本身 的类等，此时可以使用一些方法制造一个单例对象，也就是设计模式中的单例模式。设计模式

就是一套被反复使用的、多数人知晓的、经过分类编目的代码设计经验的总结。使用设计模式 是为了重用代码、让代码更容易被他人理解、保证代码可靠性。

单例模式一般都是在创建对象时进行判断，如果该类型的对象已经存在，则不再创建新的 对象，而是将已存在的对象直接返回

由于Pythoii创建对象都是在\_110¥()\_方法中，所以要在\_\_11削()\_方法中进行判断。在 Python中创建单例的方法如下所示。

|  |  |
| --- | --- |
| >>> class Singleton(object): |  |
| • o o instance = None | #表示Singleton类的唯一实例 |
| \_first\_init = False | #表明单例属性是否经过了设置 |
| def new (cis7 agez name): | #在该方法中创建对象 |
| 。. if not cis. instance: |  |
| 。。。 cis <, instance = obj ect. | , new (cis) |
| …。 return cis » instance |  |
| def init (self z age, name): | #在该方法中设置属性 |
| ... if not self. first\_init: |  |
| 。.. self« age = age |  |
| …。 self. name = name |  |
| .• . Singleton. first\_init = | True |

>>> singleton\_one = Singleton(18,"xiaoMing")

>>> singleton\_two = Singleton(9,"xiaoHong") »> print (id (singleton\_one))

7277616 一

>» print (id (singleton\_two))

7277616 —

在上述代码中，我们创建了一个Singleton类，表示单例类。它有两个类私有属性，其中 ―instance表示Singleton类的唯一实例,默认值为None；而 \_first init表示该实例是否经过初 始化设置，初始值为Falseo

在—new()—方法中判断—instance是否有值，如果没有，说明唯一实例还没有创建， 此时调用父类，也就是object类的object. new (els)方法对唯一实例进行创建并赋值给 instance?然后返回 instance；如果 instance已经存在,则直接返回。由于 instance是类 属性，所以所有的对象方法访问的是同一个\_instanceo

在\_Jnit\_()方法中首先判断\_firstjnit属性的值，如果为FMse,说明没有对唯一实例进 行初始设置，此时就可以进行设置并添加实例的私有属性；如果为Tme?说明已经对唯一实例 进行过初始设置富此时就什么都不做。

接下来我们创建了两个对象singleton\_one和singleton\_two,并打印singleton one和 singleton\_\_two的唯一标识，可以看出，它们的唯一标识相同，都是7277616,说明singleton\_ one和singleton\_two是同一个对象。我们的单例模式就成功实现了。

那么唯一实例的属性值会被改变吗？我们继续进行测试：

>>> print(singleton\_one.ager singleton\_one.name)

18 xiaoMing

»> print(singleton\_two.agez singleton\_two.name)

18 xiaoMing

>>> singleton\_one.age = 19

.>>> print (singleton\_two .. age)

19

在上述代码中,首先打印了 singleton\_one和singleton\_two的私有属性值,发现都是一样的, 并且都是第一次创建对象(对象singleton\_one )时设置的属性值。再次创建对象时(例如对象 singleton\_two ) 9不会改变私有属性的值。

然后在代码中通过对象访问的方式修改属性的值(singleton\_one.age = 19 ) 9并再次访问 对象singleton\_two的值，可以看到，属性age的值已经被修改为19 了，说明通过对象修改唯 一实例的类属性值是可行的。

1L8王厂模戒

工厂模式是常用的设计模式，它用于根据需求创建不同的对象。工厂模式通常有一个工厂 类，它提供一个生产对象的方法，这个方法负责根据传入的参数创建对应的对象。工厂模式可 以增强代码的扩展性和可维护性，当需要增加一个新类型时，不需要修改现有的类，只需要增 加一个新的类型即可。

下面是工厂模式的一个应用，该应用的具体介绍如下：

® 一个手机类(Mobile )作为基类,两个类HuaweiMobile和VivoMobile都是Mobile类的 子类。

® 一个工厂类MobileFactory,它有一个get mobile()方法,根据传入的参数决定创建哪个 类型的对象。

°用户使用工厂类MobileFactory的get\_mobile()方法创建自己要的手机。

|  |  |  |
| --- | --- | --- |
| >>> class Mobile(object): | | #定义手机类 |
|  | def init (self):  self.name = None  self oprice = None def get\_name(self):  return self.name def get\_price(self):  return self.price |
| »> | class HuaWeiMobile(Mobile):  def init (self):  print ("生产一部 HuaWei 手机!") | #定义HuaWei手机类 |
| »> | class VivoMobile(Mobile):  def \_init\_(self):  print ("生产一部Vivo手机!") | #定义Vivo手机类 |
| »> | class MobileFactory:  def get\_mobile(selffname): if name == °HuaWei°:  return HuaWeiMobile() elif name == "Vivo":  return VivoMobile() | #定义手机工厂类 |

上述代码创建了 Mobile 类、HuaweiMobile 类、VivoMobile 类和 MobileFactory 类。 接下来，用户根据自己的需要创建不同的手机。代码如下所示。

>>> factory = MobileFactory () # 创建一个工厂

>>> huawei = factory o get\_mobile("HuaWei")

生产一部HuaWei手机!

>>> vivo = factoryget\_mobile ( "Vivo") 生产一部Vivo手机!

另外，在这个例子中，如果多个地方都要用到手机工厂类，则可以将它设计成一个单例(在 上节介绍过)。