Python 继承

OOP（面向对象编程）的三大特性——数据封装、继承和多态

类

OOP（）即所谓面向对象编程，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记类是抽象的模板，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

假设我们要创建一个Student类，在Python中，定义类是通过class关键字：

```
class Student(object):
    pass
```

class后面紧接着是类名，即Student，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来的，继承的概念我们后面再讲，通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类。

定义好了Student类，就可以根据Student类创建出Student的实例，创建实例是通过类名+()实现的：

```
>>> bart = Student()
>>> bart
<__main__.Student object at 0x10a67a590>
>>> Student
<class '__main__.Student'>
```

可以看到，变量bart指向的就是一个Student的实例，后面的0x10a67a590是内存地址，每个object的地址都不一样，而Student本身则是一个类。

可以自由地给一个实例变量绑定属性，比如，给实例bart绑定一个name属性：

```
>>> bart.name = 'Bart Simpson'
>>> bart.name
'Bart Simpson'
```

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的__init__方法，在创建实例的时候，就把name，score等属性绑上去：

```
class Student(object):

    def __init__(self, name, score):
        self.name = name
        self.score = score
```

注意：特殊方法“__init__”前后分别有两个下划线！！！

注意到__init__方法的第一个参数永远是self，表示创建的实例本身，因此，在__init__方法内部，就可以把各种属性绑定到self，因为self就指向创建的实例本身。

有了__init__方法，在创建实例的时候，就不能传入空的参数了，必须传入与__init__方法匹配的参数，但self不需要传，Python解释器自己会把实例变量传进去：

```
>>> bart = Student('Bart Simpson', 59)
>>> bart.name
'Bart Simpson'
>>> bart.score
59
```

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量self，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区别，所以，你仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

继承

什么是继承?

继承是一种创建类的方法，在python中，一个类可以继承来自一个或多个父类。原始类称为基类或超类。

```
class ParentClass1: #定义父类
    pass

class ParentClass2: #定义父类
    pass

class SubClass1(ParentClass1): #单继承，基类是ParentClass1，派生类是SubClass
    pass

class SubClass2(ParentClass1,ParentClass2): #python支持多继承，用逗号分隔开多个继承的类
    pass
```

查看继承：

```
>>> SubClass2.__bases__
(<class '__main__.ParentClass1'>, <class '__main__.ParentClass2'>)
```

什么时候用继承?

假如已经有几个类，而类与类之间有共同的变量属性和函数属性，那就可以把这几个变量属性和函数属性提取出来作为基类的属性。而特殊的变量属性和函数属性，则在本类中定义，这样只需要继承这个基类，就可以访问基类的变量属性和函数属性。可以提高代码的可扩展性。

继承和抽象（先抽象再继承）

抽象即提取类似的部分。基类就是抽象多个类共同的属性得到的一个类。

```
class Riven:
    camp='Noxus'
    def __init__(self,nickname,
                 script,
                 aggressivity=54,
                 life_value=414,
                 ):
        self.nickname = nickname
        self.aggressivity = aggressivity
        self.life_value = life_value
        self.script=script
    def attack(self,enemy):
        print(self.script)
        enemy.life_value -= self.aggressivity

class Garen:
    camp='Demacia'
    def __init__(self,nickname,
                 script,
                 aggressivity=58,
                 life_value=455,
                 ):
        self.nickname = nickname
        self.aggressivity = aggressivity
        self.life_value = life_value
        self.script = script
    def attack(self,enemy):
        print(self.script)
        enemy.life_value -= self.aggressivity

g1=Garen("德玛西亚之力","人在塔在")
g1.camp="诺克萨斯"
r1=Riven("瑞雯","断剑重铸之日，骑士归来之时")
g1.attack(r1)
print(r1.life_value)
#结果:
人在塔在
356
```

Garen类和Riven类都有nickname、aggressivity、life_value、script四个变量属性和attack()函数属性，这里可以抽象出一个Hero类，里面有里面包含这些属性。

```
class Hero:
    def __init__(self, nickname,
                 script,
                 aggressivity,
                 life_value,
                 ):
        self.nickname = nickname
        self.aggressivity = aggressivity
        self.life_value = life_value
        self.script = script
    def attack(self,enemy):
        print("Hero.attack")
        enemy.life_value -= self.aggressivity

class Riven(Hero):
    camp='Noxus'
    def __init__(self,nickname,script,aggressivity=54,life_value=414):
        Hero.__init__(self,nickname,script,aggressivity,life_value)
    def attack(self,enemy):
        print(self.script)
        Hero.attack(self,enemy)

class Garen(Hero):
    camp='Demacia'
    def __init__(self,nickname,script,aggressivity=58,life_value=455):
        Hero.__init__(self, nickname, script, aggressivity, life_value)    def attack(self,enemy):
        print(self.script)
        Hero.attack(self, enemy)

g1=Garen("德玛西亚之力","人在塔在")
g1.camp="诺克萨斯"
r1=Riven("瑞雯","断剑重铸之日，骑士归来之时")
g1.attack(r1)
print(r1.life_value)

#结果:

人在塔在
Hero.attack
356
```

严格来说，上述Hero.init(self,...)，不能算作子类调用父类的方法。因为我们如果去掉（Hero）这个继承关系，代码仍能得到预期的结果。

总结python中继承的特点：

1. 在子类中，并不会自动调用基类的init()，需要在派生类中手动调用。
2. 在调用基类的方法时，需要加上基类的类名前缀，且需要带上self参数变量。
3. 先在本类中查找调用的方法，找不到才去基类中找。

继承的优缺点探讨

子类化内置类型的缺点

1. 内置类型的方法不会调用子类覆盖的方法

内置类可以子类化，但是内置类型的方法不会调用子类覆盖的方法。下面以继承dict的自定义子类重写__setitem__为例说明：

```
class ModifiedDict(dict):
    def __setitem__(self, key, value):
        super().__setitem__(key,[value]*2)
a=ModifiedDict(one=1)
a["two"]=2
print(a)
a.update(three=3)
print(a)#输出{'one': 1, 'two': [2, 2], 'three': 3}
```

从输出可以看到，键值对one=1和three=3存入a时均调用了dict的__setitem__，只有[]运算符会调用我们预先覆盖的方法。

问题的解决方式在于不去子类化dict，而是子类化colections.UserDict。

2、子类化collections中的类

用户自定义的类应该继承collections模块，如UserDict，UserList，UserString。这些类做了特殊设计，因此易于拓展。子类化UserDict的代码如下：

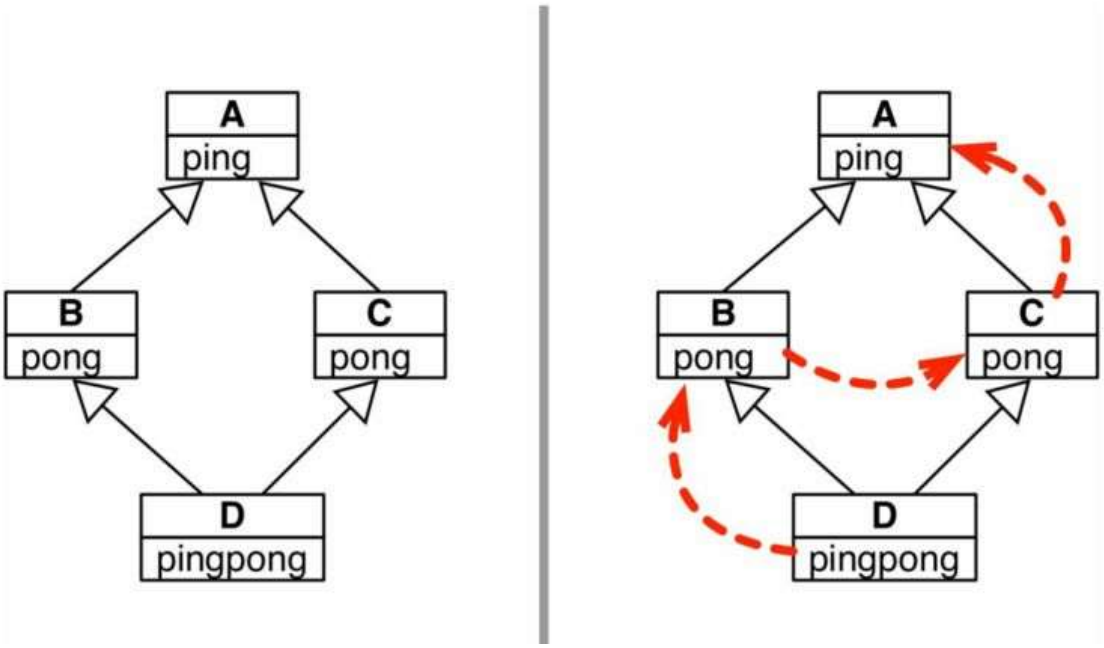
```
from collections import UserDict
class ModifiedDict(UserDict):
    def __setitem__(self, key, value):
        super().__setitem__(key,[value]*2)
b=ModifiedDict(one=1)
b["two"]=2
b.update(three=3)
print(b)#输出{'one': [1, 1], 'two': [2, 2], 'three': [3, 3]}
```

小结：上述问题只发生在C语言实现的内置类型子类化情况中，而且只影响直接继承内置类型的自定义类。相反，子类化使用Python编写的类，如UserDict或MutableMapping就不会有此问题。

多重继承

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

在多重继承中存在不相关的祖先类实现同名方法引起的冲突问题，这种问题称作“菱形问题”。Python依靠特定的顺序遍历继承图，这个顺序叫做方法解析顺序。如图，左图是类的UML图，右图中的虚线箭头是方法解析顺序：



2、super

提到类的属性__mro__，就会提到super:

super 是个类,既不是关键字也不是函数等其他数据结构。

作用：super是子类用来调用父类方法的。

语法：super(a_type, obj);

a_type是obj的__mro__，当然也可以是__mro__的一部分，同时issubclass(obj,a_type)==true

举个例子, 有个 MRO: [A, B, C, D, E, object]

我们这样调用:super(C, A).foo()

super 只会从 C 之后查找，即: 只会在 D 或 E 或 object 中查找 foo 方法。

下面构造一个菱形问题的多重继承来深化理解：

```
class A:
    def ping(self):
        print("A-ping:",self)
class B(A):
    def pong(self):
        print("B-pong:",self)
class C(A):
    def pong(self):
        print("C-PONG:",self)
class D(B, C):
    def ping(self):
        print("D-ping:",self)
        super().ping()
    def pingpong(self):
        self.ping()
        super().ping()
        self.pong()
        super(B,D).pong(self)
d=D()
d.pingpong()
print(D.mro())
```

输出如下：

```
D-ping: <__main__.D object at 0x000001B77096EAC8>
A-ping: <__main__.D object at 0x000001B77096EAC8>#前两行对应self.ping()。
A-ping: <__main__.D object at 0x000001B77096EAC8>#此处super调用父类的ping方法。
B-pong: <__main__.D object at 0x000001B77096EAC8>
C-PONG: <__main__.D object at 0x000001B77096EAC8>#此处从B之后搜索父类的pong()
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
#类D的__mro__，数据以元组的形式存储。
```

分析：d.pingpong()执行super.ping()，super按照MRO查找父类的ping方法，查询在类B到ping之后输出了B.ping()。

3. 处理多重继承的建议

（1）把接口继承和实现继承区分开；

- 继承接口：创建子类型，是框架的支柱；
- 继承实现：通过重用避免代码重复，通常可以换用组合和委托模式。

（2）使用抽象基类显式表示接口；

（3）通过混入重用代码；
混入类为多个不相关的子类提供方法实现，便于重用，但不会实例化。并且具体类不能只继承混入类。

（4）在名称中明确指明混入；
Python中没有把类声明为混入的正规方式，Luciano推荐在名称中加入Mixin后缀。如Tkinter中的XView应变成XViewMixin。

（5）抽象基类可以作为混入，反过来则不成立；
抽象基类与混入的异同：

- 抽象基类会定义类型，混入做不到；
- 抽象基类可以作为其他类的唯一基类，混入做不到；
- 抽象基类实现的具体方法只能与抽象基类及其超类中的方法协作，混入没有这个局限。

（6）不要子类化多个具体类；
具体类可以没有，或者至多一个具体超类。
例如，Class Dish(China，Japan，Tofu)中，如果Tofu是具体类，那么China和Japan必须是抽象基类或混入。

（7）为用户提供聚合类；
聚合类是指一个类的结构主要继承自混入，自身没有添加结构或行为。Tkinter采纳了此条建议。

（8）优先使用对象组合，而不是类继承。
优先使用组合可以令设计更灵活。
组合和委托可以代替混入，但不能取代接口继承去定义类型层次结构。