OOP七大设计原则-里氏替换原则

概述：

在面向对象的语言中 继承是必不可少的，非常优秀的语言机制，它有如下优点：
1)代码共享，减少创建类的工作量，每个子类都拥有父类的方法和属性.
2)提高代码的重用性
3)子类可以形似父类，但是又异于父类，子类不但拥有了父类的所有功能，还可以添加自己的功能.
4)提高代码的可扩展性，实现父类的方法就可以了，许多开源框架的扩展接口都是通过继承父类来完成.

但是所有的事物都有二面性，继承除了有上述优点，也有下面缺点：
1)继承是侵入性的，只要继承，就必须拥有父类的所有方法和属性.
2)降低了代码的灵活性，子类必须拥有父类的属性和方法，让子类有了一些约束.
3)增加了耦合性，当父类的常量 变量和方法被修改了，就需要考虑子类的修改，这种修改可能带来非常糟糕的结果，要重构大量的代码.

如何扬长避短呢？方法是引入里氏替换原则.

什么是里氏替换原则?

定义1：如果对每一个类型为S的对象o1，都有类型为T的对象o2，使得以T定义的所有程序P在所有的对象o1都代换成o2时，程序P的行为没有发生变化，那么类型S是类型T的子类型.
o1 = S()
o2 = T()
在代码中把所有的o1替换成o2的时候，程序的行为没有发生变化，那么称为S是T的子类. 把所有子类出现的位置都替换为父类，程序的功能不变.

定义2：所有引用基类的地方必须能透明地使用其子类的对象，反之不行.

定义3：里氏替换原则通俗的来讲就是，子类可以扩展父类的功能，但不能改变父类原有的功能.

里氏规则定义的代码示例：

In [5]:
```
class t(object):
    def foo(self):
        print('this is foo')

    def bar(self):
        print('this is bar')

class s(t):
    def talk(self):
        print('i am talking')

if __name__ == '__main__':
    o2 = t()
    o1 = s()
    o2.foo() # o1.foo()  也可以，符合引用基类的地方可以透明的时候子类
    o2.bar() # o2.bar()  也可以，符合引用基类的地方可以透明的时候子类
    print('#' * 10)
    o1.foo()
    o1.bar()
    o1.talk()
    print('#' * 10)
    o2.talk() ##报错，引用子类的地方不可以透明的使用其父类，因为子类可能对父类做了扩展
```

```
this is foo
this is bar
##########
this is foo
this is bar
i am talking
##########

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-5-e24fb4ace859> in <module>
     20     o1.talk()
     21     print('#' * 10)
---> 22     o2.talk() ##报错
     23
     24

AttributeError: 't' object has no attribute 'talk'
```

如何遵守里氏替换原则?
1)子类必须完全实现父类的抽象方法，但不能覆盖父类的非抽象方法.
2)子类可以实现自己特有的方法.
3)当子类覆盖或实现父类的方法时,方法的前置条件(即方法的形参)要比父类方法的输入参数更宽松.
4)当子类的方法实现父类的抽象方法时，方法的后置条件(即方法的返回值)要比父类更严格.
5)子类的实例可以替代任何父类的实例，但反之不成立.
6)子类派生的重点在于不影响父类原功能，而不是不覆盖原方法
7)尽量把父类设计为接口或抽象类，让子类实现接口或继承父类，并实现父类中声明的但是未实现方法.

代码示例(假设有如下代码)：

In [8]:
```
class A:
    def func(self, a, b):
        return a - b

if __name__ == '__main__':
    a = A()
    print("100 - 80 = %s" %a.func(100,80))
    print("100 - 50 = %s" %a.func(100,50))
```

```
100 - 80 = 20
100 - 50 = 50
```

后来，我们需要增加一个新的功能：完成两数相加然后再与100求和，由类B来负责. 即类B需要完成两个功能，1是两数相减，2是两数相加，然后再加100.
由于类A已经实现了第一个功能，所以类B继承类A后，只需要再完成第二个功能就可以了.

代码如下：

In [9]:
```
class A:
    def func1(self, a, b):
        return a - b

class B(A):
    def func1(self, a, b):
        return a + b

    def func2(self, a, b):
        return self.func1(a, b) + 100

if __name__ == '__main__':
    b = B()
    print("100 - 80 = %s" %b.func1(100,80))
    print("100 - 50 = %s" %b.func1(100,50))
    print("100 - 20 = %s" %b.func2(100,20))
```

```
100 - 80 = 180
100 - 50 = 150
100 - 20 = 220
```

我们发现原本运行正常的相减功能发生了错误，原因就是类B在给方法起名时无意中重写了父类的方法，造成所有运行相减功能的代码全部调用了类B重写后的方法，造成原本运行正常的功能出现了错误．在本例中，引用基类A完成的功能，换成子类B之后，发生了异常．在实际编程中，我们常常会通过重写父类的方法来完成新的功能，这样写起来虽然简单．但是整个继承体系的可复用性会比较差，特别是运用多态比较频繁时，程序运行出错的几率非常大．如果非要重写父类的方法，比较通用的做法是：原来的父类和子类都继承一个更通俗的基类，原有的继承关系去掉，采用依赖、聚合，组合等关系代替．

代码示例：

In [10]:
```
from abc import ABCMeta, abstractmethod

class ab(object):
    __metaclass__ = ABCMeta
    @abstractmethod
    def func1(self, a, b):
        pass
    @abstractmethod
    def func2(self, a, b):
        pass

class A(ab):
    def func1(self, a, b):
        return a - b

class B(ab):
    def func1(self, a, b):
        return a + b

    def func2(self, a, b):
        return self.func1(a, b) + 100
```