类成员包括字段，方法和属性。所以成员修饰符号，可以用于修饰这三个。

在JAVA中对类成员的访问可以使用private, protected这样的类成员修饰符，在Python中不存在这样的关键字修饰符。我们使用下划线\_进行标识。

规则1：

**在成员前加入两个下划线\_\_，那么这个成员便是私有的。不可以在外部访问，只可以使用类中定义的方法去访问。 私有的属性方法也不会被子类继承访问。**

例如:

class Student(object):

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

self.\_\_name = name

self.\_\_score = score

def print\_score(self):

print("%s: %s"%(self.\_\_name, self.\_\_score))

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问实例变量.\_\_name和实例变量. \_\_score了。达到了封装的效果。如果我们想从外部获取name和score, 或者设置name和score的值，那么我们将需要为此类添加getter和setter方法。封装以后的好处在于，当在外部进行设置获取读取的时候，我们可以通过类中getter和setter方法中定义的逻辑，对读取操作进行控制。

例如：

def get\_name(self):

return self.\_\_name

def get\_score(self):

return self.\_\_score

def set\_score(self, score):

if 0 <= score <= 100:

self.\_\_score = score

else:

raise ValueError('bad score')

规则2：

**在成员前加入一个下划线\_，那么这个成员便是受保护的。这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思是，"虽然我可以被访问，但是，请把我视为私有变量，不要随意访问。"**

规则3：

**在Python中，变量名类似\_\_XX\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量时可以直接访问的，不是private变量。**

特殊的类成员: （特殊的成员是前后都有\_\_,但是虽然前面有\_\_但是不是私有变量，可以外部调用）

|  |  |
| --- | --- |
| \_\_init\_\_ | 构造方法，当建立实例对象的时候将执行的函数 |
| \_\_del\_\_ | 析构方法，在对象被垃圾回收之前将执行的函数。 |
| \_\_doc\_\_ | 类的文档说明 |
| \_\_module\_\_ | 表示当前操作的对象在哪个模块中 |
| \_\_class\_\_ | 表示当前操作的对象在哪个类名中 |
| \_\_call\_\_ | 当对一个对象进行调用时，将执行的方法。例如:  Obj = Example()  Obj() #此时调用类中定义的\_\_call\_\_方法 |
| \_\_str\_\_ | 当对象进行print(obj)操作，或者str(obj)操作的时候将输出的字符串（默认情况下将会直接输出对象的内存地址） |
| \_\_dict\_\_ | 返回对象中的所有字段并且以字典的形式返回  或者返回类中的所有定义的方法并以字典形式返回 |
| \_\_getitem\_\_ | 当对象执行obj[key] 或者obj[index:index] 的时候，即是以中括号对对象进行取值的时候，将执行类中\_\_getitem\_\_的方法。如果我要以key或者[index:index]都能进行取值，那么我们在\_\_getitem\_\_中要进行判断而已。但是调用的都是用\_\_getitem\_\_方法。  (例如Python中的dict便是Dict类的对象，其中便定义了\_\_getitem\_\_方法，\_\_setitem\_\_,\_\_delitem\_\_方法) |
| \_\_setitem\_\_ | 当对象执行obj[key] = value 或者obj[index:index] = list的时候，即是以中括号对对象进行赋值的时候，将执行类中\_\_setitem\_\_的方法。如果我要以key或者[index:index]都能进行赋值，那么我们在\_\_setitem\_\_中要进行判断而已。但是调用的都是用\_\_setitem\_\_方法。  (例如Python中的dict便是Dict类的对象，其中便定义了\_\_getitem\_\_方法，\_\_setitem\_\_,\_\_delitem\_\_方法) |
| \_\_delitem\_\_ | \_\_delitem\_\_和\_\_setitem\_\_和\_\_getitem\_\_ 的原理一样。 |
| \_\_iter\_\_ | 如果我们想要对一个对象进行循环，例如for i in obj, 那么便要求这个obj需要有返回生成器的能力。如果没有则报错。当对一个对象进行循环时，变会寻找类中的\_\_iter\_\_方法去寻找生成器。（Python自带的可循环对象，list, dict,set类中便有定义\_\_iter\_\_方法） |

**规则4：在类中定义的函数中使用修饰符@classmethod，则该函数变为类方法，该函数只能访问到类的数据属性，不能获取实例的数据属性**

例如

@classmethod

def cal\_info(self):

print (“This is %s” %self.cal\_name)

**规则5：在类中定义的函数中使用@staticmethod代表静态方法，类或实例均可调用。静态方法函数里不传入self**

@staticmethod

def cal\_test(a,b,c):

print(a,b,c)