PYTHON程序设计

计算机学院 王纯

六 面向对象程序设计

六 面向对象程序设计

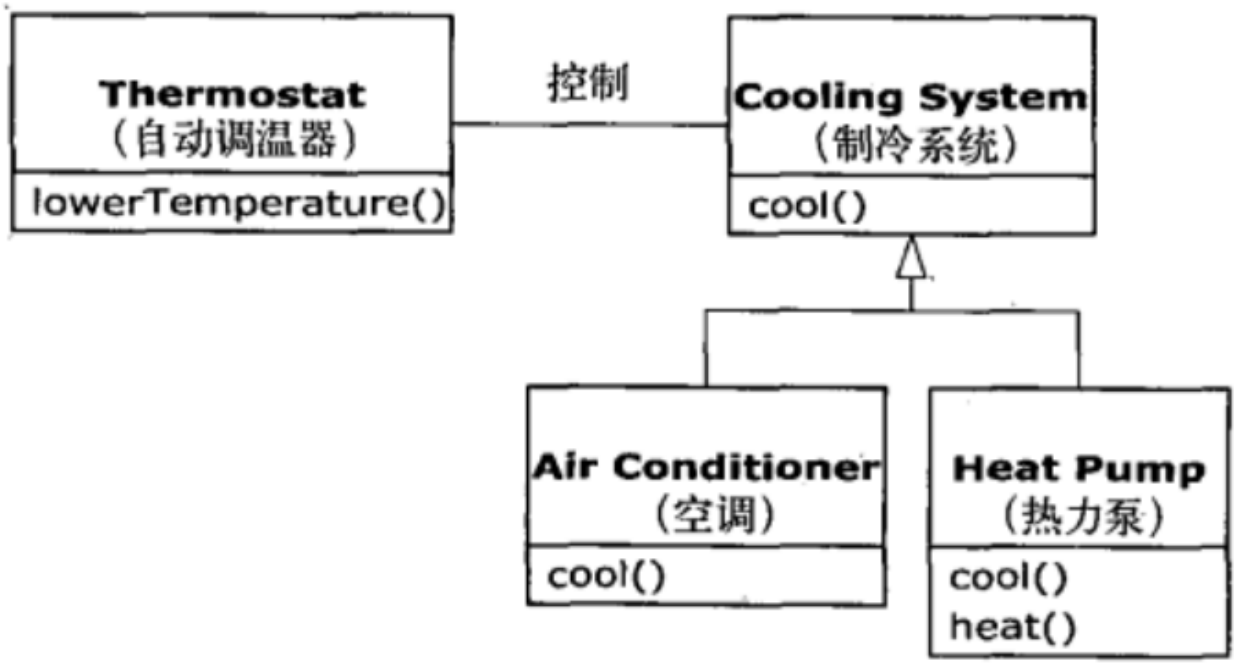
- 理解面向对象
- 类与实例
- 属性
- 继承与多态
- 设计实例
- 封装和类型检查
- 小结

理解面向对象

面向对象程序设计

- ✓是一种建立在以对象为中心的思维方式基础上的编程范式
- ✓是一种更灵活和更强大的语言抽象，允许根据问题来描述问题
- ✓对象具有状态、行为和标识（***Booch***）

面向对象对问题的抽象



理解面向对象

面向对象的主要特征

- ✓ 封装性：将数据和操作捆绑在一起，创造出一个新的类型。实现“高内聚、低耦合”的“最佳状态”
- ✓ 继承性：子类可以对基(父)类的行为进行扩展、覆盖、重定义
- ✓ 多态性：一个接口形式，多种实现方法

Python中的面向对象

- ✓ 一切皆对象
- ✓ 完全采用面向对象程序设计思想，是真正面向对象的高级语言
- ✓ 支持多种编程范式

类与实例

类的定义

类是一种*数据类型*

最简单的定义类的语法：

class <类名>:

<语句块>

当类定义正常结束时，会创建出一个类对象

引用和实例化

- ✓ 类对象支持属性的引用和实例化两种操作
- ✓ **属性的引用**：***<类名>.<属性名>***
- ✓ 实例是根据类创建出来的一个个具体的“对象”，即实例对象
- ✓ **类的实例化**，使用的是*函数表示法*
- ✓ **实例对象只支持属性的引用**。有两种有效的属性名称，即数据属性和方法。数据属性可以理解为属于对象的变量，方法可以理解为属于对象的函数

```
#类定义
class Teacher:
    """一个简单的教师类示例"""
    profession = 'education'

    def show_info(self):

        return 'This is a teacher’

#属性应用
>>> Teacher.profession
 'education'

#实例化
teacher_zhang = Teacher()
teacher_wang = Teacher()
```

属性

私有属性和公有属性

- ✓ 默认都是公有的
- ✓ 属性的名字以一个下划线 _ 开头，标明它是一个受保护(*protected*)的变量， 原则上不允许直接访问，但实际上在外部还是可以访问到这个变量的。大家约定俗成当做是私有变量
- ✓ 属性的名字以两个下划线 __ 开头，则表示是私有变量（*private*），只有内部可以访问，外部不能访问
- ✓ 没有对私有属性提供严格的访问保护机制
- ✓ 类似__***xxx***__的，即以双下划线开头且以双下划线结尾的是特殊属性或者方法

```
class Student:
    def __init__(self, name, age):
        self._name = name
        self.__age=age

stu = Student('zhangsan',30)
print(stu.age) #30
print(stu.name) #zhangsan
```

属性：下划线使用约定

模式	举例	含义
单前导下划线	_var	命名约定，仅供内部使用。 通常不会由Python解释器强制执行（通配符导入除外），只作为对程序员的提示。
单末尾下划线	var_	按约定使用以避免与Python关键字的命名冲突。
双前导下划线	__var	当在类上下文中使用时，触发"名称修饰"。 由Python解释器强制执行。
双前导和双末尾下划线	__var__	表示Python语言定义的特殊方法。 避免在你自己的属性中使用这种命名方案。
单下划线	_	有时用作临时或无意义变量的名称（"不关心"）。 也表示Python REPL中最近一个表达式的结果。

属性：数据属性

- ✓ 用来说明对象的一些特性的
- ✓ 分为**属于对象**的数据属性和**属于类**的数据属性两类
- ✓ 不需要声明。像局部变量一样，在第一次被赋值时产生
- ✓ 在一个类中，在所有方法之外定义的属性是**属于类**的**数据属性**
- ✓ **类**的**数据属性**通过类名或对象名都可以访问
- ✓ 在类的外部，**实例**的**数据属性**只能通过对象名访问

```
class Teacher: # Teacher类
    profession = 'education' #属于类的数据属性
    def __init__(self, name, age ):
        self.name = name #属于对象的数据属性
        self.age = age #属于对象的数据属性

t1 = Teacher('zhang',30) #实例化对象
t2 = Teacher('li',40)
print(t1.name,t1.age) #访问对象的数据属性
print(Teacher.profession) #访问类的数据属性
t2.title = 'lecturer' #动态添加属于对象的数据属性
```

属性：方法

- ✓ 用来描述对象所具有的行为
- ✓ 分为**实例方法、类方法和静态方法**三类：
  - **实例方法**一般都以 ***self*** 做为第一个参数，代表实例对象自身
  - 在定义**类方法**之前，需添加**@classmethod** 进行说明必须以 ***cls*** 作为第一个参数， ***cls*** 代表类本身
  - 在定义**静态方法**前，需添加**@staticmethod** 进行说明
- ✓ 静态方法和类方法都可以通过类名和对象名调用，但实例方法只能供对象名调用
- ✓ 静态方法类似函数，只是定义在类的命名空间里，因此无法使用类的属性和方法

	类	实例
实例方法	✗	✓
类方法	✓	✓
静态方法	✓	✓

属性：方法

```
class Teacher:
    profession = ‘education’ #属于类的数据属性
    def __init__(self, name, age ):
        self.name = name #属于对象的数据属性
        self.age = age #属于对象的数据属性
    def showName(self): #实例方法
        print(self.name)

    @classmethod #修饰器，声明类方法
    def showProfession(cls): #类方法
        print(cls.profession)

    @staticmethod #修饰器，声明静态方法
    def showHello(): #静态方法
        print('hello teacher')
```

t1 = Teacher('zhang',30) #实例化对象

*Teacher.showname() #**通过类调用实例方法，错误！！！***

Teacher.showProfession() #通过类调用类方法 ***education***

Teacher.showHello() #通过类调用静态方法 ***hello teacher***

t1.showname() #通过对象调用实例方法 ***zhang***

t1.showProfession() #通过对象调用类方法 ***education***

t1.showHello() #通过对象调用静态方法 ***hello teacher***

属性：类方法应用

✓静态模板

```
class Man:
    id = 0  # 类变量

    def __init__(self, name):
        self.name = name
        self.id = self.id_number()

    @classmethod
    def id_number(cls):
        cls.id += 1
        return cls.id

a = Man('A')
print(a.id)
b = Man('B')
print(b.id)
```

```
# 类方法和数据属性的使用
class Goods:
    __discount = 1

    def __init__(self, name, price):
        self.__name = name
        self.__price = price

    @property
    def price(self):
        return self.__price * self.__discount

    @classmethod
    def change_discount(cls, new_discount):
        cls.__discount = new_discount


# 改变折扣
Goods.change_discount(0.8)
apple = Goods('苹果', 10)
print(apple.price)
banana = Goods('香蕉', 15)
print(banana.price)
```

属性：类方法应用

- ✓ 设计模式的特殊构造要求
  - **抽象工厂模式(*Abstract Factory Pattern*)**
    用于创建复杂的对象，这种对象由许多小对象组成，这些小对象都属于某个特定的系列。比如示例中的“**示意图**”工厂，由图、形状、文字对象组成。

```
+--------------------------+
|   +------------------+   |
|   |%%%%%%%%%%%%%%%%%%|   |
|   |%%Abstract Factory%%|   |
|   |%%%%%%%%%%%%%%%%%%|   |
|   +------------------+   |
+--------------------------+
```

纯文本格式和SVG格式的示意图

```
class DiagramFactory:
    @classmethod
    def make_diagram(cls, width, height):
        return cls.Diagram(width, height)

    @classmethod
    def make_text(cls, x, y, text, fontsize=12):
        return cls.Text(x, y, text, fontsize)
...

class SVGDiagramFactory(DiagramFactory):
...

def create_diagram(factory):
    diagram = factory.make_diagram(30,7)
    text = factory.make_text(7, 3, ‘Abstract Factory’)
    diagram.add(text)
    return diagram

def main():
    txt_diag = create_diagram(DiagramFactory)
    svg_diag = create_diagram(SVGDiagramFactory)
...
```

属性：内置的特殊方法

*__new__()*方法用于创建一个类的新实例

*__init__()*方法是对象的初始化方法，大致相当于*C++*里的构造函数

*__del__()*方法在实例将被销毁时调用，大致相当于*C++*里的析构函数

*__str__()*方法需要返回一个字符串，是对这个对象的描述

*__call__()*方法使对象可以像函数一样直接被调用执行，即**callable**

继承与多态：继承的概念

- ✓ 继承是面向对象程序设计的最重要的特征之一，是解决软件可重用性的重要措施。
- ✓ 定义派生类的语法格式为：

class <类名>(基类名):

<语句块>

- ✓ 子类继承父类所有的公有数据属性和方法。
- ✓ 父类私有的属性和方法，子类不能继承。
- ✓ 先在本类中查找调用的方法，如果找不到，才会到基类中去查找。

```
class Person: #定义一个父类
    name=''    #公有变量，子类可以继承
    __nick = '' #私有变量，子类不能访问
    def get_name(self):
        return self.name
class Male(Person): #定义一个子类
    def get_name_plus(self):
        return self.name+'男'
zhangsan=Male()
zhangsan.name='张三' #该属性继承自父类
print(zhangsan.__nick) #出错，父类的属性不能继承
print(zhangsan.get_name_plus())#'张三_男'，子类自己的方法
print(zhangsan.get_name())#'张三'，从父类继承方法
```

继承与多态：继承示例

```
class Person(object): #object是所有类的最顶层基类名，可不写
    __count = 0 #定义一个私有变量，子类不能访问
    def __init__(self, name, age): #定义类的初始化函数
        self.age = age
        self.name = name
        Person.__count += 1  #初始化一次，count数字加1
        self.__show_count() #调用本类私有方法，输出count数字
    def get_age(self):      #定义一个返回年龄的实例方法
        return self.age
    def get_name(self):     #定义一个返回姓名的实例方法
        return self.name
    def set_age(self, age):  #定义一个设置年龄的实例方法
        self.age = age
    def set_name(self, name=""): #定义一个设置姓名的实例方法
        self.name = name
```

```
    def show_info(self):
        #定义一个打印输出信息的实例方法
        print('Person: ', self.name, ' age ', self.age)

#装饰器，将下面的方法声明为类的方法
    @classmethod
    def __show_count(cls):
        #私有的类的方法，子类不能访问
        print(cls.__count)
        #打印类的私有属性，输出实例化个数

#__两个下划线开头的类的私有方法，只能在类的内部调用 （类内部别的方法可以调用他），不能在类地外部调用
```

继承与多态：继承示例

```
class Teacher(Person):#定义一个Person类的子类
    def __init__(self, name, age, title=None):
        #显式调用父类__init__()方法进行初始化
        Person.__init__(self, name, age)
        self.title = title #单独对子类属性进行初始化
    def change_title(self, title): #给子类新增1个方法
        self.title = title
    def show_info(self):
        #重写父类的同名方法，多打印出来子类自己的属性
        print('Teacher:',self.name,' age:',self.age,'title:
',self.title)
```

```
zhang_san = Person('张三', 30)  #把张三实例化为Person类的对象
>>> 1  ### 输出 1 实例化第一次
li_si = Teacher('李四', 35, '博士')  #把李四实例化为Teacher类的对象
>>> 2  ### 输出 2 Teacher类继承自Person类，所以是实例化第二次

zhang_san.show_info()  #打印张三的基本信息
>>>Person:  张三  age  30
li_si.show_info()  #打印李四的基本信息
>>> Teacher: 李四  age: 35 title:  博士

li_si.set_age(40)      #通过继承的父类方法，修改李四的年纪
li_si.change_title('讲师')  #通过子类的方法，修改李四的title

li_si.show_info()  #打印李四的基本信息
>>> Teacher: 李四  age: 40 title:  讲师
```

继承与多态：如何判断继承

- ✓ Python 与其他语言不同，当定义一个 *class* 的时候，实际上也就定义了一种数据类型。新定义的数据类型和Python自带的数据类型，比如*str*、*list*、*dict* 没什么区别。
- ✓ Python 有两个判断继承的函数：***isinstance()*** 用于检查实例类型；***issubclass()*** 用于检查类继承。
- ✓ 类的内置私有属性***__mro__***(*method resolution order*)也记录了类的继承关系，同时给出类的**调用顺序。**

```
class Person(object):#定义一个父类
    pass            #什么内容也不填
class Child(Person): # Child继承自Person
    pass            #也略过

xiao_m = Child()
lao_w = Person()
print(isinstance(xiao_m,Child))#True,小明是Child的实例
print(isinstance(xiao_m,Person))#True,小明也是父类实例
print(isinstance(lao_w,Child)) #False,老王不是子类实例
print(isinstance(lao_w,Person)) #True,老王是父类实例
print(issubclass(Child,Person))#True,Child是Person的子类
Person.__mro__  # (<class '__main__.Person'>, <class 'object'>)
Child.__mro__   #(<class '__main__.Child'>, <class '__main__.Person'>,
<class 'object'>)
```

继承与多态：多态的概念

1）多态性也是面向对象程序设计的最重要的特征之一。

2）让具有不同功能的函数可以使用相同的函数名，这样就可以用一个函数名调用不同内容(功能)的函数。

a）只关心对象的实例方法是否同名，不关心对象所属的类型；

b）对象所属的类之间，继承关系可有可无；

c）多态的好处可以增加代码的外部调用灵活度，让代码更加通用，兼容性比较强；

d）多态是调用方法的技巧，不会影响到类的内部设计。

2）总结：一个接口,多种实现。

继承与多态：多态示例

类之间**有继承关系**的情况。不同子类，定义相同的方法名字，但是内部的逻辑不同。

```
class Animal(): #同一类事物:动物
    def talk(self):
        pass

class Cat(Animal): #动物的形态之一:猫
    def talk(self):
        print('喵喵') #重写了talk函数

class Dog(Animal): #动物的形态之二:狗
    def talk(self):
        print('汪汪')#重写了talk函数

c = Cat()
d = Dog() #分别实例化一个猫和狗的对象

def func(obj): #定义一个新方法，传入对象，调用该对象talk方法
    obj.talk()
func(c)
func(d) #'喵喵'、'汪汪'；同样的方法，传入不同对象，输出就不同
```

继承与多态：多态示例

类之间**没有继承关系**的情况。完全没有关系的类，定义相同的方法名字，内部的逻辑也完全不同。

```
class Animal(): #同一类事物:动物
    def talk(self):
        pass

class Cat(Animal): #动物的形态，猫
    def talk(self):
        print('喵喵') #重写了talk函数

class Child(): #和动物完全没有关系，也不是继承自Animal
    def talk(self):
        print('balabala')#只是Child类里面也有talk函数

c = Cat()
d = Child() #分别实例化一个猫和Child的对象

def func(obj): #定义一个新方法，传入对象，调用该对象talk方法
    obj.talk()
func(c)
func(d) #'喵喵'、'balabala'；同样的方法，传入不同对象，输出就不同
```

继承与多态：多态示例

增加了程序灵活性,不论对象千变万化，使用者都是同一种形式去调用，如*func(obj)*。

增加了程序的可扩展性,通过继承animal类创建了一个新的类，比如猪，无需更改，还是用*func(obj)*去调用。

继承与多态：多态-关于鸭子类型DUCK TYPING

1）鸭子类型是动态类型的一种风格。一个对象有效的语义，是由**"当前方法和属性的集合"决定**。

2）这个概念来源于诗人詹姆斯·惠特科姆·莱利（James Whitcomb Riley,1849- 1916）的诗句：*When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck*。

3）在不支持鸭子类型的语言中，编写一个函数，接受鸭子对象，调用鸭子的“走”、“叫”函数。支持鸭子类型的语言中，该函数，可以接受任何类型的对象，只要它有“走”、“叫”函数。

继承与多态：多态-关于鸭子类型DUCK TYPING

4）***Duck Typing*** 作为程序设计中的一种**类型推断风格**，适用于大部分脚本语言 / 动态语言 (如 *Python*、*Ruby*、*Perl*、*JavaScript* 等) 和某些静态语言 (如 Golang，通常静态类型语言在编译前便已显式指定变量类型，而 *Golang* 却则在编译时推断变量类型)。支持 *duck typing* 的语言，其解释器/编译器 将会在解释/解析 (*Parse*) 或编译时推断对象类型。

5）鸭子类型可以灵活地实现多态的“一个接口多种实现”

6）***Duck Typing*** 没有任何**静态检查**，通常得益于不检查方法和函数中参数的类型，而**依赖文档、清晰的代码和测试**来确保正确使用，充分体现动态语言的灵活性。

```
class Duck:
    "Duck 对象具有 fly 方法 (鸭子的飞行行为) 和 run 方法 (鸭子的奔跑行为)"
     def fly(self):
                print("Duck flying")
     def run(self):
                print("Duck running")

class Chick:
    "Chick 对象具有 fly 方法 (小鸡的飞行行为) 和 run 方法 (小鸡的奔跑行为)"
    def fly(self):
                print("Chick flying")
    def run(self):
                print("Chick running")

class Plane:
    "Plane 对象具有 fly 方法 (飞机的飞行行为)"
    def fly(self):
                print("Plane flying")

class Fish:
    "Fish 对象具有 swim 方法 (鱼儿的游泳行为)"
    def swim(self):
                print("FIsh Swimming")

def go(entity):
    "令传入实例对象 entity 执行飞行和奔跑操作 (不论什么东西能飞、能跑就行)"
    entity.fly()  # 令传入实例对象 entity 执行 fly 方法
    entity.run()  # 令传入实例对象 entity 执行 run 方法
```

继承与多态：多态-关于鸭子类型DUCK TYPING

```
>>> duck = Duck()  # 实例化 Duck 对象
>>> chick = Chick()  # 实例化 Chick 对象
>>> plane = Plane()  # 实例化 Plane 对象
>>> fish = Fish()  # 实例化 Fish 对象

>>> go(duck)  # 有 fly 和 run 方法就能正常运行, 不关心实例对象的类型是什么 (不论什么东西, 能飞和跑就行)
Duck flying
Duck running

>>> go(chick)  # 有 fly 和 run 方法就能正常运行, 不关心实例对象的类型是什么 (不论什么东西, 能飞和跑就行)
Chick flying
Chick running
```

```
>>> go(plane)  # 没有 run 方法就报错, 不关心实例对象的类型是什么 (不论什么东西, 不能跑就不行)
Plane flying
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    go(plane)
  File "<pyshell#13>", line 3, in go
    entity.run()
AttributeError: 'Plane' object has no attribute 'run'

>>> go(fish)  # 没有 fly 方法就报错, 不关心实例对象的类型是什么 (不论什么东西, 不能飞就不行)
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    go(fish)
  File "<pyshell#35>", line 2, in go
    entity.fly()
AttributeError: 'Fish' object has no attribute 'fly'
```

设计实例：几何形状-设计一个基本几何形状的基类

```
class Shape: #基类Shape
    def __init__(self,name):
        self.__name=name
    def __str__(self): #特殊方法，存入描述信息
        return '图形是：'+self.__name+','
    def area(self): #几何形状，有求面积的方法
        return
    def perimeter(self):#几何形状，有求周长的方法
        return
```

设计实例：几何形状-定义一个圆形的子类

```
class Circle(Shape): #子类Circle
    def __init__(self,r):
        Shape.__init__(self,'圆形')#调用父类的初始化，圆形
        self.r=r    #然后再增加圆形的特殊属性——半径
    def perimeter(self): #重写圆形的周长公式，2πr
        return 2*3.14*self.r
    def area(self):#重写圆形的面积公式，πrr
        return 3.14*self.r**2
    def __str__(self):#重写特殊方法，追加半径信息
        return Shape.__str__(self)+'半径为：'+str(self.r)
```

设计实例：几何形状-定义一个三角形的子类

```
class Triangle(Shape): #子类Triangle
    def __init__(self,a,b,c):
        Shape.__init__(self,'三角形')#调用父类的初始化，三角形
        self.a=a#定义三条边
        self.b=b#定义三条边
        self.c=c#定义三条边
    def perimeter(self):#重写三角形的周长公式，a+b+c
        return self.a+self.b+self.c
    def area(self):#重写三角形的面积公式，海伦公式
        s=(self.a+self.b+self.c)/2
        return math.sqrt(s*(s-self.a)*(s-self.b)*(s-self.c))
    def __str__(self):#重写特殊方法，追加三条边信息
        return Shape.__str__(self)+'三条边长为:
'+str(self.a)+','+str(self.b)+','+str(self.c)
```

设计实例：几何形状-定义一个长方形的子类

```
class Rectangle(Shape): #子类Rectangle
    def __init__(self,a,b):
        Shape.__init__(self,'矩形')#调用父类的初始化，矩形
        self.a=a#定义长宽
        self.b=b#定义长宽
    def perimeter(self):#重写矩形的周长公式，2（长+宽）
        return 2*(self.a+self.b)
    def area(self):#重写矩形的面积公式，长*宽
        return self.a*self.b
    def __str__(self):#重写特殊方法，追加长宽信息
        return Shape.__str__(self)+'长和宽为：'+str(self.a)+','+str(self.b)
```

设计实例：几何形状-测试输出

```
t1=Triangle(3,4,5)  #实例化三角形
t2=Triangle(9,8,6)  #实例化三角形
c1=Circle(3)        #实例化圆形
c2=Circle(7)        #实例化圆形
r1=Rectangle(3,9)   #实例化长方形
r2=Rectangle(7,6)   #实例化长方形
shapes=[t1,t2,c1,c2,r1,r2] #构建一个不同对象类型组成的列表
for s in shapes: #输出每个图形的面积和周长
    print(f'{s} 面积是：{s.area():.2f}，周长是{s.perimeter():.2f}')
#调用每个子类的求面积和求周长方法进行计算

#如没有面向对象，圆形、三角形、长方形，之间并没有什么关系，各自定义各自的求面积和求周长的函数。
#但现实世界中，这三类形状，确实都是几何形状，符合数学视角，也更符合和人们看待世界的方式。
#从扩展性上看，采用面向对象之后，扩展也更方便。比如，给所有几何形状增加“求质心”的方法。给长方形，增加求对角线的方法等。给三角形增加等腰、等边、直角三角形等子类。
```

封装和类型检查：问题

Python的面向对象语法简洁，结合动态类型的特点，编写程序灵活便捷，但对于数据封装的限制多为约定性的，类型检查也主要依赖文档、注释等同样是约定性的手段。

*如有合法性检查之类的需求，可以通过语言**内置的特性**和**设计模式**来实现。*

```
class Book:

    def __init__(self, title, isbn, price, quantity):
        self.title = title
        self.isbn = isbn
        self.price = price
        self.quantity = quantity

>>>book = Book("Programming in Go", "ISBN 0321774639", 44.99, 5220)
>>>book.title = ""
```

一个图书类的例子

问题：数据属性赋值的时候，使用***实例.属性=属性值***的方式显然把属性暴露出来了，并且也无法对属性值进行限制检查。

封装和类型检查：

通过对每个数据属性定义*get*、*set*方法，可解决数据暴露和检查的问题。但略显复杂，能否更简洁?

方案一：***@property***修饰器可以使调用方式看起来和直接赋值一样。

方案二：采用**类修饰器**对整个类的数据进行附加处理，也能达到同样的效果。

```
class Book:

    def __init__(self, title, isbn, price, quantity):
        self.set_title(title)
        self.set_isbn(isbn)
        self.set_price(price)
        self.set_quantity(quantity)

    def get_title(self):
        return self.__title

    def set_title(self, value):
        if not isinstance(value, str):
            raise ValueError("Book title must be of type str")
        if not bool(value):
            raise ValueError("Book title may not be empty")
        self.__title = value
```

封装和类型检查：PROPERTY修饰器

Python内置的 ***property*** 修饰器的作用是把方法变成属性，调用的时候不需要加括号。这个修饰器的定义是一个类，包含*getter*、*setter*、*deleter*和*doc*四个方法分别对属性值实现读取、设置、删除和文档说明的功能。

使用***property*** 修饰器就可以使调用看起来和赋值一样，同时把合法性检查等操作悄悄完成了。

@property广泛应用在类的定义中，可以让调用者写出简短的代码，同时保证对参数进行必要的检查，还可以定义属性的读写特征（没有实现setter方法即只读属性）。

```
@property
def title(self):
    return self.__title

@title.setter
def title(self, value):
    if not isinstance(value, str):
        raise ValueError("Book title must be of type str")
    if not bool(value):
        raise ValueError("Book title may not be empty")
    self.__title = value
```

```
>>> abook.title = “Python in Practice“

>>>print (abook.title)
```

封装和类型检查：类修饰器

函数修饰器可以使被修饰的函数功能更强，是一种非常有用的模式。

修饰器机制可用来修饰函数和方法，同样可以修饰类。

```
@cls_decorator
class  Some_Sort:
 # 相当于 Some_Sort=cls_decorator(Some_Sort)
def cls_decorator(cls):
    ....
    return cls
```

参数化 修饰器通常只有一个参数，就是被修饰的函数、方法或类。实际上在原有修饰器实现的基础上增加一层多参数外壳就可以实现参数化。

```
def  parameterize(*args):
    def cls_decorator(cls):
        ....   #这里可以使用args参数列表
        return cls
    return cls_decorator
```

这样就得到了一个参数化的类修饰器。

封装和类型检查：类修饰器

针对问题，我们构建一个修饰器完成合法性检查的工作。经过这个修饰器的修饰，目标类的定义有变化。

对于每一个需要检查的数据属性，在原属性名前加__（双下划线）产生一个等同的私有属性。

为每个私有属性定义一个getter（读取）和一个setter（设置）的方法，其中setter方法中调用外部传入的类型相关的验证函数validate。

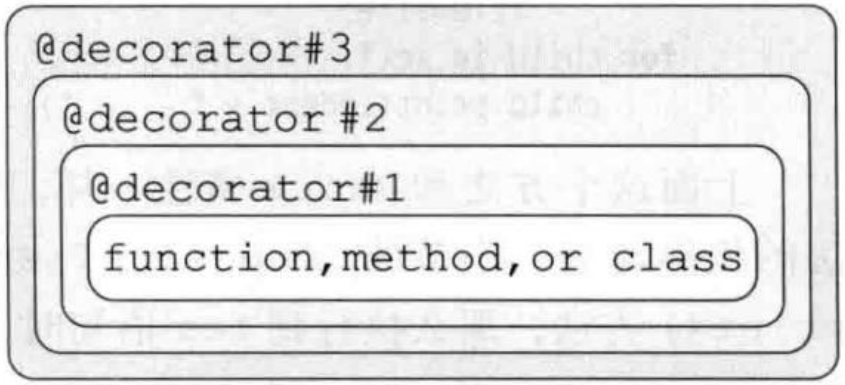
```
def ensure(name, validate, doc=None):
    """This class decorator factory needs a property name and a validate
    function, and will accept a property docstring
    The validate function should return None on success and raise a
    ValueError on failure.
    """
    def decorator(Class):
        privateName = "__" + name
        def getter(self):
            return getattr(self, privateName)
        def setter(self, value):
            validate(name, value)
            setattr(self, privateName, value)
        setattr(Class, name, property(getter, setter, doc=doc))
        return Class
    return decorator
```

封装和类型检查：类修饰器

编写验证函数，例如检测名称必须是字符串且不为空。

在初始类定义的基础上加上每个属性检验有关的修饰器即可。这样的修饰还可以被其子类继承。

注意：修饰器是可以叠加的，相当于嵌套。




```
def is_non_empty_str(name, value):
    if not isinstance(value, str):
        raise ValueError("{} must be of type str".format(name))
    if not bool(value):
        raise ValueError("{} may not be empty".format(name))

@ensure("title", is_non_empty_str)
@ensure("isbn", is_valid_isbn)
@ensure("price", is_in_range(1, 10000))
@ensure("quantity", is_in_range(0, 1000000))
class Book:
    def __init__(self, title, isbn, price, quantity):
        self.title = title
        self.isbn = isbn
        self.price = price
        self.quantity = quantity
```

小结

Python面向对象和C++的简要对比

- ✓ **封装**
  - 数据属性：C++的静态数据成员和非静态数据成员分别对应Python的类属性和实例属性
  - 方法：C++的静态成员函数和非静态成员函数分别对应Python的类方法和实例方法
- ✓ **继承**
  - C++有*public/protected/private*三种模式，Python只有一种
  - C++的继承建立在虚函数的应用上，Python采用 ***MRO*** 解析继承关系
- ✓ **多态**
  - C++的多态建立在继承的基础上
  - Python-体现动态语言特点的鸭子类型*Duck typing*

六 面向对象程序设计

- 理解面向对象
- 类与实例
- 属性
- 继承与多态
- 设计实例
- 封装和类型检查
- 小结

谢谢