

约束，反射，内置方法

讲师：尚玉杰

本章目录

- 类的约束
- 类中的其他方法
- 反射

约束

- 约束：指的是对类的约束
 - 在一些重要的逻辑, 与用户数据相关等核心部分, 要建立一种约束, 避免发生此类错误
- 类的约束有两种解决方式:
 - 方式一：在父类建立一种约束（通过抛出异常）
 - 方式二：引入抽象类的概念

约束

- 举例：
 - 公司让小明给他们的网站完善一个支付功能，小明写了两个类，QQ支付和支付宝支付
 - 但是上面这样写不太方便，也不合理，老大说让他整改，统一一下付款的方式，小明开始加班整理，使用归一化设计
 - 后来小明接了大项目，公司招了一个新的程序员春哥接替小明的工作，老大给春哥安排了任务，让他写一个微信支付的功能，春哥随意给微信支付类中的付钱方法起了一个方法名
 - 后来接手这个项目的程序员，也可能随意起名

约束

- 解决方式一：
 - 提取父类，在父类中定义好方法，在这个方法中抛一个异常。这样所有的子类都必须重写这个方法。否则，访问的时候就会报错

```
class Payment:
    def pay(self, money):
        raise Exception('子类必须实现pay方法')
class QQpay(Payment):
    def pay(self, money):
        print(f'使用QQ支付 {money} ')
class Alipay(Payment):
    def pay(self, money):
        print(f'使用Ali支付 {money} ')
class Wechat(Payment):
    def zhifu(self, money):
        print(f'使用微信支付 {money} ')
def pay(obj, money): # 归一化设计
    obj.pay(money)
obj1 = QQpay()
obj2 = Alipay()
obj3 = Wechat()
pay(obj3, 300)
```

约束

- 解决方式二：
 - 用抽象类(制定一种规范)的概念, 建立一种约束
 - 基类如下设置, 子类如果没有定义pay方法, 在实例化对象时就会报错

```
from abc import ABCMeta, abstractmethod
class Payment(metaclass=ABCMeta):
    @abstractmethod
    def pay(self, money):
        pass
# 设置一个类的metaclass (元类) 是ABCMeta
# 那么这个类就变成了一个抽象类(接口类)
# 这个类的功能就是建立一个规范
# 由于该方案来源是java和c#. 所以不常用
```

约束

- 解决方式二：

- 详解：

- Python本身不提供抽象类和接口机制，要想实现抽象类，可以借助abc模块。ABC是Abstract Base Class（抽象父类）的缩写
 - `abc.ABCMeta`是用来生成抽象基础类的元类。由它生成的类可以被直接继承，但是抽象类不能直接创建对象（只能被继承）
 - `@abstractmethod`表明抽象方法，如果子类不具备`@abstractmethod`的方法，那么就会抛出异常

类方法

- 指一个类中通过`@classmethod`修饰的方法
 - 第一个参数必须是当前类对象，该参数名一般约定为“cls”，通过它来传递类的属性和方法（不能传实例的属性和方法）
 - 调用：实例对象和类对象都可以调用
 - 类方法是将类本身作为对象进行操作的方法

类方法

- 使用场景分析：
 - 假设我有一个学生类和一个班级类，要实现的功能为：
 - 执行班级人数增加的操作、获得班级的总人数
 - 学生类继承自班级类，每实例化一个学生，班级人数都能增加
 - 最后，我想定义一些学生，获得班级中的总人数
 - 因为我实例化的是学生，但是如果我从学生这一个实例中获得班级总人数，在逻辑上显然是不合理的
 - 同时，想要获得班级总人数，如果生成一个班级的实例也是没有必要的

类方法

```
class Student:
    __num = 0
    def __init__(self, name, age):
        self.name = name
        self.age = age
        Student.addNum() # 写在__new__方法中也合适
    @classmethod
    def addNum(cls):
        cls.__num += 1
    @classmethod
    def getNum(cls):
        return cls.__num
```

静态方法

- 使用@staticmethod修饰的方法
 - 参数随意，没有“self”和“cls”参数，方法体中不能使用类或实例的任何属性和方法
 - 实例对象和类对象都可以调用
- 详解：静态方法是类中的函数，不需要实例。静态方法主要是用来存放逻辑性的代码，逻辑上属于类，但是和类本身没有关系，也就是说在静态方法中，不会涉及到类中的属性和方法的操作。可以理解为，静态方法是个**独立的、单纯的**函数，它仅仅托管于某个类的名称空间中，便于使用和维护

静态方法

- 譬如，我想定义一个关于时间操作的类，其中有一个获取当前时间的函数

```
import time
class TimeTest(object):
    def __init__(self, hour, minute, second):
        self.hour = hour
        self.minute = minute
        self.second = second
    @staticmethod
    def showTime():
        return time.strftime("%H:%M:%S", time.localtime())
print(TimeTest.showTime())
t = TimeTest(2, 10, 10)
nowTime = t.showTime()
print(nowTime)
```

静态方法

- 详解：
 - 上页中使用了静态方法（函数），然而方法体中并没使用（也不能使用）类或实例的属性（或方法）
 - 若要获得当前时间的字符串时，并不需要实例化对象，此时对于静态方法而言，所在类更像是一种名称空间
 - 其实，我们也可以在类外面写一个同样的函数来做这些事，但是这样做就打乱了逻辑关系，也会导致以后代码维护困难

property

- property: 是一种特殊的属性，访问它时会执行一段功能（方法）然后返回值
- 举例：
 - BMI指数（bmi是计算而来的，但很明显它听起来像是一个属性而非方法，如果我们将它做成一个属性，更便于理解）
 - 成人的BMI数值：
 - 过轻：低于18.5
 - 正常：18.5-23.9
 - 过重：24-27
 - 肥胖：28-32
 - 非常肥胖，高于32
 - 体质指数（BMI）= 体重（kg）÷ 身高²（m）
 - shang: $65\text{kg} \div (1.82 \times 1.82) = 19.623233908948194$

property

```
class People:
    def __init__(self, name, weight, height):
        self.name=name
        self.weight=weight
        self.height=height
    @property
    def bmi(self):
        return self.weight / (self.height**2)

p1=People('egon', 75, 1.85)
print(p1.bmi)
```

property

- 为什么要用property:
 - 将一个类的方法定义成属性以后，对象再去使用的时候obj.name, 根本无法察觉自己的name是执行了一个函数然后计算出来的
 - 这种特性的使用方式**遵循了统一访问的原则**

property

- 属性一般具有三种访问方式，获取、修改、删除
 - 我们可以根据这几个属性的访问特点，分别将三个方法定义为对同一个属性的获取、修改、删除
 - 只有在属性定义property后才能定义setter, deleter

```
class Foo:
    @property
    def AAA(self):
        print('get的时候运行我')
    @AAA.setter
    def AAA(self, value):
        print('set的时候运行我')
    @AAA.deleter
    def AAA(self):
        print('delete的时候运行我')
```

property

- 第二种方式:

```
class Foo:
    def get_AAA(self):
        print('get的时候运行我啊')

    def set_AAA(self, value):
        print('set的时候运行我啊')

    def delete_AAA(self):
        print('delete的时候运行我啊')
    AAA=property(get_AAA, set_AAA, delete_AAA)
#内置property三个参数与get, set, delete一一对应
```

property

- 使用场景举例：

```
class Student(object):  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score  
# 当我们想要修改一个 Student 的 score 属性时，可以这么写  
s = Student('Bob', 59)  
s.score = 60  
# 但是也可以这么写  
s.score = 1000  
# 显然，直接给属性赋值无法检查分数的有效性
```

property

- 如果将score设置为私有属性并提供取值设值方法是可以解决该问题的

```
def get_score(self):  
    return self.__score
```

```
def set_score(self, score):  
    if score < 0 or score > 100:  
        raise ValueError('invalid score')  
    self.__score = score
```

property

- 但是：但是写 `s.get_score()` 和 `s.set_score()` 没有直接写 `s.score` 来得直接

@property

```
def score(self):  
    return self.__score
```

@score.setter

```
def score(self, score):  
    if score < 0 or score > 100:  
        print("数据有误")  
    self.__score = score
```

@score.deleter

```
def score(self):  
    del self.__score
```

property

- 总结：
- 只有@property定义只读，加上@setter定义可读可写，加上@deleter定义可读可写可删除

反射

- 反射的概念是由Smith在1982年首次提出的，主要是指程序可以访问、检测和修改它本身状态或行为的一种能力（自省）
- python面向对象中的反射：通过字符串的形式操作对象相关的属性。python中的一切事物都是对象（都可以使用反射）
- 有四个可以实现自省的函数：
hasattr, getattr, setattr, delattr

反射

- 应用于对象的反射

```
class Foo:
```

```
    f = '类变量'
```

```
    def __init__(self, name, age):
```

```
        self.name=name
```

```
        self.age=age
```

```
    def say_hi(self):
```

```
        print('hi, %s'%self.name)
```

```
obj=Foo('小明', 73)
```

```
#检测是否含有某属性
```

```
print(hasattr(obj, 'name'))
```

```
print(hasattr(obj, 'say_hi'))
```


反射

- 应用于对象的反射

#获取属性

```
n=getattr(obj, 'name')
```

```
print(n)
```

```
func=getattr(obj, 'say_hi')
```

```
func()
```

```
print(getattr(obj, 'aaaaaaaaa', '不存在啊')) #报错
```

反射

- 应用于对象的反射

#设置属性, 该属性不一定是存在的

```
setattr(obj, '呵呵', True)
```

```
setattr(obj, 'show_name', lambda self: self.name+'呵呵')
```

#综合用法

```
getattr(obj, "age", setattr(obj, "age", "18"))
```

#age属性不存在时, 设置该属性

```
print(obj.__dict__)
```

```
print(obj.show_name(obj))
```

反射

- 应用于对象的反射

#删除属性

```
delattr(obj, 'age')
```

```
delattr(obj, 'show_name')
```

```
# delattr(obj, 'show_name111')#不存在, 则报错
```

```
print(obj.__dict__)
```

反射

- 应用于类的反射

```
class Foo(object):  
    staticField = "old boy"  
    def __init__(self):  
        self.name = 'wupeiqi'  
    def func(self):  
        return 'func'  
    @staticmethod  
    def bar():  
        return 'bar'  
print(getattr(Foo, 'staticField'))  
print(getattr(Foo, 'func'))  
print(getattr(Foo, 'bar'))
```

反射

- 应用于当前模块的反射

```
import sys
```

```
def s1():
```

```
    print('s1')
```

```
def s2():
```

```
    print('s2')
```

#每当程序员导入新的模块，sys.modules (是一个字典) 都将记录这些模块

```
this_module = sys.modules[__name__]
```

```
print(hasattr(this_module, 's1'))
```

```
a = getattr(this_module, 's2')
```

```
a()
```

反射

- 反射的应用举例：

- 当我们打开浏览器，访问一个网站，单击登录就跳转到登录界面，单击注册就跳转到注册界面
- 但是，你单击的其实是一个个的链接，每一个链接都会有一个函数或者方法来处理

```
class User:
    def login(self):
        print('欢迎来到登录页面')

    def register(self):
        print('欢迎来到注册页面')

    def save(self):
        print('欢迎来到存储页面')
```

反射

- 没学反射之前的解决方式:

```
while True:
```

```
    choose = input('>>>').strip()
```

```
    if choose == 'login':
```

```
        obj = User()
```

```
        obj.login()
```

```
    elif choose == 'register':
```

```
        obj = User()
```

```
        obj.register()
```

```
    elif choose == 'save':
```

```
        obj = User()
```

```
        obj.save()
```

反射

- 学了反射之后的解决方式:

```
user = User()
while True:
    choose = input('>>>').strip()
    if hasattr(user, choose):
        func = getattr(user, choose)
        func()
    else:
        print('输入错误。。。。')
```