**Level II：Property（长得像Attribute的Method）（推荐使用）**

**Property**的设计初衷：

- 代码复用
- 延迟计算
- 更加规范的对象属性访问管理

场景：我要减肥，需要监控BMI指标，但是只能测量体重，每天更新体重，隔几天看一次BMI指数

```
class X:
    def __init__(self, w, h):
        self.w = w
        self.h = h
        self.BMI = w / h ** 2


def main():
    x = X(75, 1.83)
    print(x.BMI)
    x.w = 74
    x.w = 73
    x.w = 72
    print(x.BMI)

if __name__ == "__main__":
    main()
```

输出：

```
22.395413419331717
22.395413419331717
```

---

改进一：

```
class X:
    def __init__(self, w, h):
        self.__w = w
        self.__h = h
        self.BMI = w / h ** 2

    def update_w(self, w):
        self.__w = w
```

```
        self._update_bmi()

    def _update_bmi(self):
        self.BMI = self.__w / self.__h ** 2


def main():
    x = X(75, 1.83)
    print(x.BMI)
    x.update_w(74)
    x.update_w(73)
    x.update_w(72)
    print(x.BMI)

if __name__ == "__main__":
    main()
```

输出：

```
22.395413419331717
21.49959688255845
```

分析：

1. w变为私有，更新需要通过对象方法类执行，并将BMI的更新放于其中，实现功能逻辑
2. BMI属性依旧可以被外部访问和修改
3. 与w相关的代码全部被更改
4. 无论BMI属性是否被访问，每次w更新均会更新BMI，造成一定的计算资源浪费

---

改进二：

```
class X:
    def __init__(self, w, h):
        self.w = w
        self.h = h

    def get_bmi(self):
        return self.w / self.h ** 2


def main():
    x = X(75, 1.83)
    print(x.get_bmi())
    x.w = 74
    x.w = 73
    x.w = 72
    print(x.get_bmi())
```

```
if __name__ == "__main__":
    main()
```

分析：

1. 保持w和h属性可以随意更改，bmi指数仅在被访问时实时计算出结果
2. 访问BMI的方式由属性改为方法，造成一定程度上的代码修改
3. 在w更新频率高于BMI访问频率时，节省了计算资源
4. 当w未更新，却多次调用BMI指数时，造成了重复计算

---

改进三：

```
class X:
    def __init__(self, w, h):
        self._w = w
        self._h = h
        self._bmi = w / h ** 2

    def get_w(self):
        return self._w

    def set_w(self, value):
        if value <= 0:
            raise ValueError("Weight below 0 is not possible.")
        self._w = value
        self._bmi = self._w / self._h ** 2

    def get_bmi(self):
        return self._bmi

    w = property(get_w, set_w)
    BMI = property(get_bmi)


def main():
    x = X(75, 1.83)
    print(x.BMI)
    x.w = 74
    x.w = 73
    x.w = 72
    print(x.BMI)


if __name__ == "__main__":
    main()
```

分析：

1. 通过**Property**对象显式的控制属性的访问
2. 仅在w被更改的时候更新BMI，充分避免了重复计算
3. 很容易的增加了异常处理，对更新属性进行预检验
4. 完美复用原始调用代码，在调用方不知情的情况完成功能添加

说明：

1. **Property**对象声明必须在访问控制函数之后，否则无法创建**Property**对象
2. 这里的写法是为了深入剖析**Property**对象，其实一般不这样写，有更加优雅的写法

---

改进四：

```
class X:
    def __init__(self, w, h):
        self._w = w
        self._h = h
        self._bmi = w / h ** 2

    @property
    def w(self):
        return self._w

    @w.setter
    def w(self, value):
        if value <= 0:
            raise ValueError("Weight below 0 is not possible.")
        self._w = value
        self._bmi = self._w / self._h ** 2

    @property
    def BMI(self):
        return self._bmi


def main():
    x = X(75, 1.83)
    print(x.BMI)
    x.w = 74
    x.w = 73
    x.w = 72
    print(x.BMI)


if __name__ == "__main__":
    main()
```

说明：

1. 从改进三中我们发现，传给**Property**对象的其实是函数名，那么最优雅的方式当属**Decorator**

了

> 回顾一下**Property**的设计初衷：

- 代码复用
- 延迟计算
- 更加规范的对象属性访问管理

**关于Property的用法**

- `property(fget=None, fset=None, fdel=None, doc=None)`
- 使用`@Property`默认实现了可读
- 被`@Property`装饰过的`method`可以通过`@method.setter`继续装饰单输入参数方法实现可写

---

## Cross-Cutting and Duck Typing

**单继承 vs 多态**

- 单继承保证了纵向的复用和一致性
- 多态保证了跨类型的复用和一致性

**传统OOP vs 鸭子类型**

- 传统OOP基于类别进行设计，从类别出发逐步扩展
- 鸭子类型仅考虑功能，从需要满足的功能出发进行设计

**传统OOP的多态 vs 鸭子类型的多态**

- 传统OOP中的多态大多基于共同的基类进行设计
- Python中的鸭子类型无需考虑继承关系，实现了某个通用的接口就可以完成多态设计（Special Method）

**MixIn：基于鸭子类型的视角看Multi-Inheritance**

```
class X:
    def f1():
        pass

class Y(X):
    def f2():
        pass

class A:
    def f3():
        pass

def do_f1(x):
    x.f1()

def do_f2(x):
```

```
    x.f2()

def do_f3(x):
    x.f3()

class Z(Y, A):
    pass
```

> MixIn的设计用中文讲比较贴切的应该是：**赋能**

**Cross-Cutting：基于鸭子类型的视角看Decorator与Special Method**

- 通过给一个类实现一个个的**Special Method**，你就让这个类越来越像Python的**Built-in Class**
- 实现**Special Method**是从语言衔接层面为你的Class赋能
- 实现**Decorator**是从通用的函数功能层面为你的Class赋能
- 通过**Multi-Inheritance**，利用**MixIn**的理念，你可以为你的Class批量化的赋能

**Python语言设计的三个关键词（其实是一件事的三个不同视角）**

- **Duck Typing**
- **Consistency**
- **赋能**

---

## 总结

- 本系列课程最艰难的一节终于结束了，后面的课程虽然功能强大，但你会感到很轻松
- 本节课仅仅是抛砖引玉，Python中关于OOP的知识还有90%没讲
- 但其实做AI的话，到这里就足够了，如果你想卷起袖子写框架，那还有很长的路要走
- 经验：从一个编程语言的设计初衷来学习，你会事半功倍，否则你会永远在高手的围城之外