目录

[Python中常见的设计模式 2](#_Toc43494473)

[一、简述 2](#_Toc43494474)

[二、设计模式类别 2](#_Toc43494475)

[三、代码实现 3](#_Toc43494476)

[1.结构型模式： 3](#_Toc43494477)

[2.行为型模式 8](#_Toc43494478)

[3.创建型模式： 11](#_Toc43494479)

[四、总结 14](#_Toc43494480)

[五、参考文献 14](#_Toc43494481)

# Python中常见的设计模式

## 一、简述

本学期，学校开设了“Python与数据挖掘”课程。在课程里，我深入浅出的了解了许多Python语言相关的特性，从它的基础到一些常见的语言特性（比如：面向对象，修饰器等）。Python给我的印象是它的对数据类型的要求并不是很严格，我想这一特性也正是它在大数据相关领域使用如此广泛的一个重要原因。再就是它封装了很多他人已经优化的很好的函数库，使的初学者能很快上手，使用Python进行相关软件开发。

这篇文章，我将会从设计模式角度出发，谈谈我的设计模式的一些认识，以及如何利用Python来简单实现一些设计模式。设计模式：它是一种在软件开发中的高层抽象，注重于模块与模块间的整合、解偶和组织关系。它更看重软件未来的扩展，维护以及健壮性；并不关注软件的具体实现细节（包括数据结构，算法等）。而为什么会产生设计模式，什么时候要使用它？我在《精通Python设计模式》这本书中得到了相关的一些解答。

首先，设计模式并不是万能的“灵丹妙药“，并不是在所有软件开发中就一定要套用一个设计模式的。设计模式是在已有的方案上发现一个更好的方案，它考虑的更多的是在已有的项目上进行优化和重构。当一个项目变得难以扩展、维护时，可以考虑根据其业务逻辑来套用相应的设计模式。设计模式的优势更多的凸显在软件后期的维护，重构；大型项目更是如此，设计模式使得整个项目的结构变得更清晰，模块与模块之间的耦合度更低，未来项目的扩展性更好。我想这才是设计模式真正发挥它作用的地方。

## 二、设计模式类别

1.设计模式分为三大类：创建型模式，结构型模式，行为型模式。

（1）创建型模式：对象实例创建相关的设计模式，其优势在于模块复用和对象的追踪，使用者和对象创建方式解耦，使用者完全不需要知道创建对象的细节等。

（2）结构型模式：该模式更多的是考虑模块与模块间的组织关系，如何更好的解耦，使得整个项目结构更加清晰；在将来项目进行扩展，维护时更加简单。

（3）行为型模式：该模式，我对它的理解是如何组织对象/模块之间的关联关系，它比较注重对象内部实现的方法。这类模式更像是针对特定业务逻辑而总结出来的一套解决方案，而不是一定说它有什么特别好的地方。

#### 2.三大类设计模式包含的具体设计模式

创建型模式：

工厂模式，建造者模式，原型模式

结构型模式：

适配器模式，修饰器模式，外观模式，享元模式，模型-视图-控制器模式，代理模式

行为型模式：

责任链模式，命令模式，解释器模式，观察者模式，状态模式，策略模式，模板模式

## 三、代码实现

下面，我将分别从三大类设计模式中挑选出四个我理解的比较透彻的例子来说明自己的观点和代码实现。

### 1.结构型模式：

（1）在该模式下，我主要来说明模型-试图-控制器(以下简称MVC)和适配器模式。

（2）MVC模式，该模式被认为是一种架构模式，注重点在于模块分离。我在Web编程中使用这种模式最多，也对它相对更了解。该模式秉承着‘关注点分离’原则把应用切分为不同部分[1]，每个部分各司其职且功能单一，尽量做到松耦合高内聚。

在MVC中，主要有三个部分：

模型(Model)：模型是MVC中的核心，它是应用的数据来源，负责与数据库交互，数据处理逻辑。

视图(View)：视图属于MVC应用中的表现层，它主要负责向控制器传递数据和对来自控制器的数据进行渲染，然后展示给用户。

控制器(Controller)：控制器属于链接模型和视图的连接层。

它即接受来自模型的数据也接受来自使徒的数据，数据在这进行中转和简要处理后在分别发往模型/视图。它负责应用的业务逻辑（转发，验证等）。

（3）代码实现：

在该代码实例中：我使用main、show这两个方法来模拟视图层，只负责显示数据，控制器类：Controller，模型类：Model。利用Python的dict()方法得到一个字典来模拟数据库访问，使用Python的namedtuple包来封装用户信息。最后主要目的是模拟用户登陆，注册过程的MVC实现。

Model:

**class** **Model:**

**def** \_\_init\_\_**(**self**):**

**global** users

**global** number

number **=** number **+** 1

**def** register**(**self**,** name**,** sex**,** age**,** password**):**

**for** key **in** users**:**

**if** name **is** users**.**get**(**key**).**name**:**

**return** **False**

self**.**user **=** User**(**name**=**name**,** sex**=**sex**,** age**=**age**,** password**=**password**)**

users**[**number**]** **=** self**.**user

**return** **True**

**def** check\_user**(**self**,** name**,** password**):**

**for** key **in** users**:**

**if** users**[**key**].**password **==** password **and** users**[**key**].**name **==** name**:**

**return** **{True,** users**[**key**]}**

**return** **{False,** **None}**

该类只负责与数据库进行交互。

全局变量number作为表项主键。

register方法负责将收集来的数据，在进行用户存在校验后，决定是否插入数据库。

check\_user方法负责在数据库中查找对应用户，并返回查找结果（找到/未找到）

Controller:

**class** **Controller:**

**def** \_\_init\_\_**(**self**):**

self**.**model **=** Model**()**

**def** to\_register**(**self**,** name**,** sex**,** age**,** password**):**

**return** self**.**model**.**register**(**name**,** sex**,** age**,** password**)**

**def** to\_login**(**self**,** name**,** password**):**

**return** self**.**model**.**check\_user**(**name**,** password**)**

控制器类：

to\_register：该方法接收视图传递过来的数据，并调用模型方法进行用户注册。最后返回注册成功与否的标志（True/False）

to\_login：该方法调用模型方法，对登录用户进行校验。

View:

**def** show**(**user**):**

**print(**'Welcome you {}, this is home page'**.**format**(**user**.**name**))**

**while** **True:**

leave **=** input**(**'Enter [L]eave to leave there'**)**

**if** leave **==** 'L'**:**

**break**

**def** main**():**

**while** **True:**

want **=** input**(**'What you want, [R]egister or [L]ogin (Others to leave)?'**)**

controller **=** Controller**()**

**if** want **==** 'R'**:**

name **=** input**(**'Input your name for register: '**)**

sex **=** input**(**'Input your sex for register: '**)**

age **=** input**(**'Input your age for register: '**)**

password **=** input**(**'Input your password for register: '**)**

**if** controller**.**to\_register**(**name**,** sex**,** age**,** password**):**

**print(**'Register Success!!!'**)**

**else:**

**print(**'The user not exit in database'**)**

**elif** want **==** 'L'**:**

name **=** input**(**'Input your name for login: '**)**

password **=** input**(**'Input your password for login: '**)**

flag**,** user **=** controller**.**to\_login**(**name**,** password**)**

**if** flag**:**

show**(**user**)**

**else:**

**print(**'The {} is not in database, please to register'**.**format**(**name**))**

**else:**

**print(**'Bye...'**)**

**break**

这两个方法为视图方法：

main方法：用户选择登录或注册，若为登录，则提供登录信息，由控制器进行登陆校验并返回登录是否成功标志；若用户选择注册，则提供注册信息后，由控制器调用模型方法，并根据返回结果进行校验。

show方法：用户登录成功后跳转的视图。

在这两个方法中，使用return关键字模拟视图跳转，并且这两个视图方法唯一的作用就是向用户展示信息。

（4）适配器模式

适配器模式也属于结构型模式中的一种，它的主要关注点在于模块之间的兼容。就是怎么在不改变原有代码的情况下使原本不兼容的两个接口之间兼容。常见情况是：将老组件用在新系统或新组件用在老系统时，出现的不兼容情况；并且修改原有代码不切实际。面对这种情况最好的解决办法就是写一个适配器类去兼容双方。主要思路是用适配器去包装我们要使用的某个类，在不修改它的前提下，使该类能够兼容现有API。

（5）代码实现：

该代码是我仿照之前提到的《精通Python设计模式》书中例子，结合书本后边给读者留下的问题，进行自己的实现。

该代码共两个类A，B。A代表当前API中使用的一个老类，当前API（main方法）只知道调用A中名为speak的方法，现在要做的就是在不改变A类和当前API的情况下，如何兼容新类B。在这里的实现用到了Python的特性：类内部自带属性字典‘\_\_dice\_\_’，它的主要用途就是保存该类实例各种属性方法。使用该特性来对B进行兼容。

**class** **A:**

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

self**.**name **=** name

self**.**zb **=** 'zoubin'

**def** \_\_str\_\_**(**self**):**

**return** self**.**name

**def** speak**(**self**):**

**return** 'my name is {}'**.**format**(**self**.**name**)**

**class** **B:**

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

self**.**name **=** name

**def** \_\_str\_\_**(**self**):**

**return** self**.**name

**def** execute**(**self**):**

**return** 'They name is {}'**.**format**(**self**.**name**)**

类A和B，这是两个简单的类。可以看到这两个类第三个方法的方法名是有区别的。

**class** **Adapter:**

**def** \_\_init\_\_**(**self**,** obj**,** adapted\_method**):**

self**.**obj **=** obj

self**.**\_\_dict\_\_**.**update**(**adapted\_method**)**

**def** \_\_str\_\_**(**self**):**

**return** str**(**self**.**obj**)**

这就是适配器类，它的构造器传入一个obj(对象)参数并赋值给类内部的obj变量，之后使用类自带的\_\_dict\_\_属性来建立一个方法名和实际方法的映射，这就是适配器兼容的关键，之后的方法调用就可以使用统一的方法名来进行不同的方法调用。

**def** main**():**

objects **=** **[**A**(**'Alis'**)]**

b **=** B**(**'Bob'**)**

objects**.**append**(**Adapter**(**b**,** dict**(**speak**=**b**.**execute**,** name**=**b**.**name**)))**

**for** ob **in** objects**:**

**print(**'{} {}'**.**format**(**ob**.**name**,** ob**.**speak**()))**

**print(**dir**(**objects**[**1**])**

该方法模拟适配器兼容，就是使用一个对象数组来收集不同类型的对象，之后对该数组进行遍历，从数组中取出对象，对象以统一的方法调用打印相关信息。

在for循环中，数组objects中每个取出的对象都调用speak方法，但是之前被Adapter类封装的B类实例，以及Adapter类都没有speak方法，他们之所以能调用该方法，就是因为Adapter类做了一个speak方法名和实际B类实例调用方法execute的映射。这样，objects中取出的Adapter类实例调用speak方法，实际上是之前给Adapter类实例传入的B类实例在调用其execute方法。

这样，书中提到的问题就迎刃而解了，为什么按照书中代码直接在for循环中写ob.name会报错？就是因为A类有该属性，而Adapter类没有该属性。解决办法就是在Adapter类的\_\_dict\_\_属性中添加关于name属性的映射即可。

### 2.行为型模式

（1）该模式中，我主要来说明观察者模式。

（2）观察者模式中有两个角色：观察者和发布者，观察者在消息发布者那里进行注册；之后，消息发布者相关状态发生改变，就可以被观察者看到并做出响应。观察者模式也可以套用是MVC模式来理解，在MVC中Model是消息发布者，View是观察者。观察者模式也遵从关注点分离，观察者和消息发布者尽量做到对对方的API最小依赖。

现实中视频软件就是个很好的例子，当用户关注了某些其他用户后，该被关注用户只要在该视频平台上发布视频，关注他的用户就会无差别收到该视频发布通知。

（3）代码实现

在该例子中，利用了Python的面向对象的特征，将发布者必须拥有的属性，方法抽象到一个父类中，继承它的子类就可以在此基础上进行功能的扩展了。该简单程序模拟的是：当发布者的一个私有属性\_data发生了改变，他就调起注册了到他的观察者，观察者做出响应改变。在这里，观察者把改变后的发布者的\_data属性以不同进制打印出来即可。

**class** **Publisher:**

**def** \_\_init\_\_**(**self**):**

self**.**observes **=** **[]**

**def** add**(**self**,** observe**):**

**if** observe **not** **in** self**.**observes**:**

self**.**observes**.**append**(**observe**)**

**else:**

**print(**'Failed to add: {}'**.**format**(**observe**))**

**def** remove**(**self**,** observe**):**

**try:**

self**.**observes**.**remove**(**observe**)**

**except** ValueError**:**

**print(**'Failed to remove: {}'**.**format**(**observe**))**

**else:**

**print(**'Success to remove: {}'**.**format**(**observe**))**

**def** notify**(**self**):**

**for** o **in** self**.**observes**:**

o**.**notify**(**self**)**

该父发布者：\_\_init\_\_方法负责初始化一个用于观察者注册的数组，add方法就是观察者用于注册的方法，remove是模拟当该系统中观察者下线后，由该发布者来删除离线观察者。上述两个方法均作了简单的错误校验，add方法禁止插入已存在对象，remove方法禁止删除不存在对象，并以if判断方式和异常处理方式进行错误响应。最后notify方法负责当有消息发布时，来唤醒观察者。

**class** **DefaultFormatter(**Publisher**):**

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

Publisher**.**\_\_init\_\_**(**self**)**

self**.**name **=** name

self**.**\_data **=** 0

**def** \_\_str\_\_**(**self**):**

**return** "{}: '{}' has data = {}"**.**format**(**type**(**self**).**\_\_name\_\_**,** self**.**name**,** self**.**\_data**)**

*@property*

**def** data**(**self**):**

**return** self**.**\_data

**def** set\_data**(**self**,** new\_value**):**

**try:**

self**.**\_data **=** int**(**new\_value**)**

**except** ValueError **as** e**:**

**print(**'Error: {}'**.**format**(**e**))**

**else:**

self**.**notify**()**

该类就是扩展后的发布者，他在set\_data方法中定义了唤醒观察者的时机和机制：既当其私有属性\_data发生改变时唤醒。

**class** **HexFormatter:**

**def** notify**(**self**,** publisher**):**

**print(**"{}: '{}' has now hex data = {}"**.**format**(**type**(**self**).**\_\_name\_\_**,**publisher**.**name**,** hex**(**publisher**.**data**)))**

**class** **BinaryFormatter:**

**def** notify**(**self**,** publisher**):**

**print(**"{}: '{}' has now binary data = {}"**.**format**(**type**(**self**).**\_\_name\_\_**,**publisher**.**name**,**bin**(**publisher**.**data**)))**

以上两个类就是注册在观察者类DefaultFormatter中的两个观察者，当发布者发布消息，就唤醒观察者做出响应，在这里观察者只是通过打印发布者\_data属性的不同进制来模拟响应机制。

**def** main**():**

df **=** DefaultFormatter**(**'default formatter'**)**

**print(**df**)**

hf **=** HexFormatter**()**

bf **=** BinaryFormatter**()**

df**.**add**(**hf**)**

df**.**add**(**bf**)**

df**.**add**(**bf**)**

df**.**set\_data**(**10**)**

df**.**remove**(**hf**)**

df**.**remove**(**hf**)**

最后，该方法模拟启动发布者，并对发布者的数据做出改变来唤醒观察者并观察现象。同时，有意添加已存在对象和删除不存在对象来测试观察者的错误处理机制。

### 3.创建型模式：

（1）在该模式下，我主要来说明抽象工厂模式。

（2）抽象工厂模式更多的是工厂模式的泛化，工厂模式是可以动态的根据需要，利用工厂类的工厂方法创建一种类对象实例，而抽象工厂则是其内部有一组这样的方法，每种方法能根据提供的特定信息来创建不同类的对象实例。这样，使用对象的创建对其使用者透明而且便于对象追踪。生活中，产品生产过程也可以理解为一个抽象工厂。产品的不同部件在不同的机床上按需生产，但是生产车间是作为一个整体存在的，想要什么产品只需要向该公司下单即可。

（3）代码实现

在该代码中，我使用了两组基础对象，分别是：第一组：人(Person)和朋友(PFriend)，第二组：动物(Animal)和朋友(AFriend)；每组的对象相互交互。使用了两个抽象工厂类用于按需要生成上述类，分别是：PRelation (描述人与朋友关系)，ARelation(描述动物与朋友关系)。最后使用类(World)，主要目的是动态的生成上述两个抽象工厂，并开始对象交互过程。该函数主要运行方式是：用户决定选择是角色人还是角色动物，并为所选角色起个名字。选定后，便决定了使用哪个抽象工厂。由World类实例负责调用自身\_\_init\_\_方法负责初始化该抽象工厂并启动go\_on方法模拟对象交互。

**class** **Person:**

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

self**.**name **=** name

**def** \_\_str\_\_**(**self**):**

**return** self**.**name

**def** play**(**self**,** friend**):**

**print(**'A person {} with {} to {}'**.**format**(**self**,** friend**,** friend**.**action**))**

**class** **PFriend:**

**def** \_\_str\_\_**(**self**):**

**return** 'they friend'

*@property*

**def** action**(**self**):**

**return** 'play basketball'

**class** **Animal:**

**def** \_\_init\_\_**(**self**,** kind**):**

self**.**kind **=** kind

**def** \_\_str\_\_**(**self**):**

**return** self**.**kind

**def** play**(**self**,** friend**):**

**print(**'A {} with {} to {}'**.**format**(**self**,** friend**,** friend**.**action**))**

**class** **AFriend:**

**def** \_\_str\_\_**(**self**):**

**return** 'it friend'

*@property*

**def** action**(**self**):**

**return** 'get food'

以上四个类是分为两组的基础类，主要描述组内类实例之间的交互动作（Person和Animal中的play方法）。

**class** **PRelation:**

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

self**.**name **=** name

**def** get\_a\_creature**(**self**):**

**return** Person**(**self**.**name**)**

**def** get\_a\_friend**(**self**):**

**return** PFriend**()**

**class** **ARelation:**

**def** \_\_init\_\_**(**self**,** kind**):**

self**.**kind **=** kind

**def** get\_a\_creature**(**self**):**

**return** Animal**(**self**.**kind**)**

**def** get\_a\_friend**(**self**):**

**return** AFriend**()**

这两个类就是抽象工厂类，他们提供不同的方法来创建不同的类实例。

**class** **World:**

**def** \_\_init\_\_**(**self**,** abstract\_factory**):**

self**.**creature **=** abstract\_factory**.**get\_a\_creature**()**

self**.**friend **=** abstract\_factory**.**get\_a\_friend**()**

*@property*

**def** go\_on**(**self**):**

self**.**creature**.**play**(**self**.**friend**)**

该类负责接收用户输入，根据用户输入来使用不同的抽象工厂创建不同对象。

## 四、总结

看完《精通Python设计模式》这本书给我最大的感受就是设计模式在日常生活中到处都存在，并不是只局限在软件工程中。

它并不是一种死板的设计样板，并不是说拿到一个项目只要硬套设计模式就可以了；尽管设计模式提供了对待特定问题的解决方案，但这绝不意味着非此不可。我想它更鼓励的是：在需求明确并已经有了对项目的总体设计实现方案，按照该方案进行相关实现后，发现项目出现了后期难以维护或扩展问题时；或者对老代码进行优化重构时，再使用设计模式。

同时，设计模式的优点会随着软件规模的增大而越发显露。通过合适的设计模式，会使得代码的逻辑结构一目了然，不仅在开发过程中调错容易，而且写出来的代码更加模块化，易于复用，耦合度低。

但是，我想设计模式之所以难也在于并不是每个设计模式都是单独拿出来用的。现实中，更多的是将一些设计模式组合到一起使用，而如何进行这种组合，使得项目更优化才是灵活运用设计模式的难的地方。

## 五、参考文献

1. 《精通Python设计模式》[荷]Sakis Kasampalis 著 夏永锋 译