三大特征：

封装：集成打包，实现同一功能所需的数据与方法；为外部提供统一的调用接口和必要的访问机制。

继承：复用已有的功能，在不改变原有功能的基础上扩展自己的功能。

多态：通过继承关系复用已有的功能接口，在不改变被继承对象（父类）的接口功能的前提下，重写该接口的功能。

六大原则：

开闭原则：支持功能扩展，关闭原有代码修改

类的单一职责：类所处理的变化因子要单一（有且只有一个）

类与类，方法不同；对象与对象，数据不同。

职能对应的是功能的区别，如果确实是处理方法差异性很大的功能，应当单独提出一个变化因子做一个类，相应的联系采用关联的方式处理类与类之间的关系。

如：客人在餐厅吃饭，吃的是红烧肉。

我们可以定义一个类 客人，客人具有吃的方法。

再定义一个类 餐厅 ，餐厅具有出菜的方法。我们固然可以把餐厅的菜单固化到类里面，但是这意味着这家餐厅万年如一的口味，如果店家需要更改自家的菜单，则添加新菜的动作就是一个新的变化点，这样也就是说餐厅要做两件事，出菜给客人和研发新菜式。

如果餐厅不研发新菜：

class DiningHall:

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

list\_menu = [“红烧肉”，“糖醋鱼”，......]

def OrderMenu(self,index):

print(“客户要%d”%（self.list\_menu[index]）)

这样如果后面要扩展新菜品，就不得不更改原有代码，从而破坏了“开闭原则”，不如改成

class DiningHall:

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

list\_menu = []

def MenuAdd(self,food):

self.list\_menu.append(food)

def OrderMenu(self,index):

print(“客户要%d”%（self.list\_menu[index]）)

如果菜品的数据相当复杂，且包含很多数据及相关的计价方法，我们则需要构建相关的食物类来处理变化。注意上面的更改虽然是给列表添加变量，但是不要忘记变量本身也是一种类，也就是上述方法，我们一样是做了将多出来的变化提出来。

组合复用原则：使用耦合度相对较低的关联方式替换耦合度最高的泛化方式

重点：继承只用作功能和数据的复用，以及同名功能的扩展。如果需要使用数据的扩展，或者父类不包含的方法扩展，建议将与继承无关的数据提出子类，使他们与子类保持一种关联关系而非 与继承无关的数据被继承关系裹挟，从而造成无关的耦合 。

如程序员类 和 销售员类，它们的只是岗位的差异造成薪资结算方式的不同。如果用员工继承程序员或者销售员类，那么当员工换岗的时候，要么是销毁原有数据再重新新建数据要么是子类重写继承自父类的薪资结算方法，这种改动是方法层面上的改动，需要改动原始的代码，违反了“开闭原则”。

程序如下：

class a：

def \_\_init\_\_(self,value):

self.basic\_salary = value

def get\_salary(self):

return self.basic\_salary

class b(a):

def \_\_init\_\_(self,value,value1,name):

super().\_\_init\_\_(value)

self.deduction\_wage = value1

self.name = name #name属于与继承无关数据

def get\_salary(self):

return super.get\_salary + self.deduction\_wage

class c(a):

def \_\_init\_\_(self,value,value1,name):

super().\_\_init\_\_(value)

self.find\_bug = value1

self.name = name #name属于与继承无关数据

def get\_salary(self):

return super.get\_salary + self.find\_bug\*5

如果，此时实例化一个小王的B类员工，但是小王想转c岗，我们需要最有可能是删除b类员工小王，然后创建一个c类员工小王，这样其实造成了更改过程中多余的内存开销和新增系统垃圾，同样与继承无关的数据被继承关系裹挟。

程序应改成：

class a：

def \_\_init\_\_(self,value):

self.basic\_salary = value

def get\_salary(self):

return self.basic\_salary

class b(a):

def \_\_init\_\_(self,value,value1):

super().\_\_init\_\_(value)

self.deduction\_wage = value1

#self.name = name 将与继承无关数据提出去

def get\_salary(self):

return super.get\_salary + self.deduction\_wage

class c(a):

def \_\_init\_\_(self,value,value1):

super().\_\_init\_\_(value)

self.find\_bug = value1

#self.name = name 将与继承无关数据提出去

def get\_salary(self):

return super.get\_salary + self.find\_bug\*5

class staff：

def \_\_init\_\_(self,name,a\_target):

self.name = name #将原有无关继承数据提出来

self.job = a\_target #工作类与员工类的关系是关联

组合复用原则也是从另一个侧面体现类的单一职责功能，他侧重于如果多出来的变化点与继承无关的情况下出现的处理原则。继承中子类与父类在公有属性上的差异可以通过方法重写来实现修改，但是如果新出现的变化点与继承关系无关，纯粹属于子类自己多出来的，那么为了防止程序改动时出现的小改大动，尽可能的将与继承无关的数据提出去单独成类，并让现有类对象成为新建类的成员。

依赖倒置：使方法的依赖关系不是直接指向具体的对象，而是指向对象变化部分的抽象。

沿用上例说明

class staff：

def \_\_init\_\_(self,name,a\_target):

self.name = name #将原有无关继承数据提出来

self.job = a\_target #工作类与员工类的关系是关联

员工类里面的成员job，关联的是工作类的父类，而非具体的岗位，因为具体的岗位是可能发生变化的，但是无论如何变化还是离不开它是岗位的客观事实，所以关联具体岗位的父类，可以让类的使用者可以尽可能的不受具体的变化点性质改变带来的影响。依赖倒置的好处是：隔离变化

里式替换：类外调用类方法体时，类方法的依赖指向依赖参数（变化）的共性（父类）。以确保当调用者提供参数发生改变时，类方法本身提供的转化依旧稳定适用。实现类与操作对象之间的变化隔离，使得操作对象类型（方法）的变化不会影响类方法本身。

上例的调用：

class staff：

def \_\_init\_\_(self,name,a\_target):#认为要处理父类if isinstance(a\_target,a):

self.name = name #将原有无关继承数据提出来

self.job = a\_target #实际处理的数据是子类J01

j01 = b(10000,5000) #b是a的子类

s01 = staff(“小王”，j01) //实际传子类

个人认为里式替换是父子类作为参数的一种特性，原则是基于这种特性的一种要求，里式替换其实可以视为依赖倒置成立的一种必须，如果传递接收的标准是父类，如果方法体本身操作的不是实际子类的数据不是子类的数据，那么方法和实际子类根本无从建立联系，那样就不是隔离变化而是隔绝变化了。

迪米特法则：尽可能少的提供不必要的数据，或者尽可能只提供必要数据，以实现低耦合

以组合复用原则中出现的例子作为引子：

类中的数据不应该出现不必要的耦合，与实际方法无关的数据应当提出去，因为他们在这个类里面显得太另类，太无关，与类的其他成员之间显得太陌生，不必要的数据和方法应当独立于类的经常变化点，因为它们的改变不频繁。

class a：

def \_\_init\_\_(self,value):

self.basic\_salary = value

def get\_salary(self):

return self.basic\_salary

class b(a):

def \_\_init\_\_(self,value,value1,name):

super().\_\_init\_\_(value)

self.deduction\_wage = value1

self.name = name #name属于与继承无关数据

def get\_salary(self):

return super.get\_salary + self.deduction\_wage

应当把无关于变化点的数据脱离变化点本身，使它更稳定，而不是让它容易受到变化的牵连，变成时变数据。

分类以变化的单一原则来分割并封装。

复用的数据、方法使用继承、多态来提取共性和个性。

无关的数据、方法，应使其尽量不要绑定变化内部，应使用关联而非继承使之相对稳定。

调用依赖关系时，使用父类作为依赖，在增强方法适用性的同时隔离了方法体内部与变化，使方法体内部的输入输出关系相对稳定且纯粹。

流程

分而治之：先分解需求，将需求分解成若干个变化因子单一的小需求块

封装变化：根据需求块，分配所属的方法体，根据方法体所需要的必要数据进行整合、集成、封装

高内聚：剩余的填充数据，单独建立稳定独立的区域。除了纯数据类外，类里面的数据无有多余，都是方法所需，尽量保留大量的功能内聚，尽可能的减少偶然内聚。

低耦合：尽量确保，具体需求的变化，不会造成与具体功能无关的代码部分的修改。

以上是我对于面向对象三大特征、六大原则的见解。