"""

10:50

面向对象三大特征:

封装:

语法:

数据:用一个类将多个变量包装起来.

```
class 类名:
    def __init__(self,参数):
        self.数据1 = 参数
        self.数据2 = 参数
        self.数据3 = 参数
    def 方法1(self):
        操作数据的逻辑
    def __方法2(self):
        操作数据的逻辑
变量 = 类名(参数)
变量.数据1
变量.数据2
变量.数据3
变量.方法1()
```

行为: 对外隐藏实现功能的细节

注意:对外要提供必要的功能

设计:

分而治之,变则疏之.

继承:

语法:

```
class 爸爸:
    def __init__(self,参数1):
        self.数据1 = 参数1
    功能1
class 儿子(爸爸):
    def __init__(self,参数1,参数2):
        super().__init__(参数1)
        self.数据2 = 参数2
    功能2
变量 = 儿子()
变量.功能1()
变量.功能2()
```

设计:

抽象具体事物,统一概念(约束子类),隔离变化

```
    多态:
        语法:依赖重写实现
        重写:儿子具有和爸爸相同的方法,实际执行的是儿子的方法(覆盖)
        做法:重写 + 创建子类对象 --> 调用父执行子
            class 爸爸:
                def 功能1(self):
                    .....
            class 儿子(爸爸):
                def 功能1(self):
                    .....
            # 创建子类对象
            变量 = 儿子()
            变量.功能1() # 执行儿子的功能
            # 不是多态(创建父类对象)
            变量 = 爸爸()
            变量.功能1() # 执行爸爸的功能
        设计:调用父类方法,执行子类方法(创建子类对象).
            # 做函数使用父类
            def 函数(爸爸):
                爸爸.功能1()
            # 用函数使用儿子
            函数(儿子())# 传递到函数中的是子类对象
面向对象设计原则:
    开闭原则:允许增加新功能,不能修改以前的代码.
    单一职责:一个类处理一个变化点
    依赖倒置:调用父,而不调用子.
    组合复用:通过变量(参数/实例变量)调用,而不是通过继承调用.
案例1:
    老张开车/坐飞机...去东北
    # 违背了:开闭原则,依赖倒置
    class 人:
        def 去(飞机):
            if 是飞机:
                ....
            elif 是车:
                ...
    ------------架构师---------------
    class 人:
        def 去(交通工具):
```

```
            # 现象:调用父类,执行子类.
            交通工具.运输()
        class 交通工具:
            def 运输():
                ...
        ------------程序员---------------
        # 做法1:重写
        class 飞机(交通工具):
            def 运输():
                ...
        class 车:
            def 运输():
                ...
         ------------测试---------------
        变量1 = 人()
        # 做法2:创建的是子类对象
        变量2 = 飞机()
        变量1.去(变量2)
    案例2:
        手雷爆炸伤害玩家/敌人.
        封装:手雷/玩家/敌人
        继承:受害者隔离了手雷爆炸的逻辑与玩家或者敌人受伤的逻辑
        多态:手雷爆炸依赖受害者,实际执行玩家与敌人.
        开闭:增加/减少新受害者,手雷不用改变.
        单一:手雷(爆炸) 受害者(隔离变化) 玩家(受伤逻辑) 敌人(受伤逻辑)
        依赖倒置:手雷依赖受害者,不依赖玩家/敌人.
        组合复用:手雷的爆炸方法,通过参数"受害者"访问玩家/敌人"受伤"方法
         ------------架构师---------------
        class 手雷:
            def 爆炸(受害者):
                受害者.受伤()
        class 受害者:
            def 受伤():
                ...
        ------------程序员---------------
        class 玩家(受害者):
            def 受伤():
                ...
```

```
    class 敌人(受害者):
        def 受伤():
            ...
    ------------测试---------------
    变量1 = 手雷()
    变量2 = 玩家()
    变量1.爆炸(变量2)
案例3:
    图形管理器管理圆形/矩形.
    封装:图形管理器/圆形/矩形
    继承:图形隔离了图形管理器计算总面积的逻辑与圆形/矩形获取面积的逻辑
    多态:图形管理器依图形,实际执行圆形,矩形.
    开闭:增加/减少新图形,图形管理器不用改变.
    单一:图形管理器(管理图形) 图形(隔离变化) 圆形(获取面积) 矩形(获取面积)
    依赖倒置:图形管理器依赖图形,不依赖圆形/矩形.
    组合复用:图形管理器的计算总面积方法,通过实例变量"所有图形"访问圆形/矩形"获取面积"方法
    ------------架构师---------------
    class 图形管理器:
        def __init__():
            self.所有图形 = []
        def 计算总面积():
            for 图形 in self.所有图形:
                累加 图形.获取面积()
    class 图形:
        def 获取面积():
            ...
    ------------程序员---------------
    class 圆形(图形):
        def 获取面积():
            ...
    class 矩形(图形):
        def 获取面积():
            ...
    ------------测试---------------
    变量1 = 图形管理器()
    变量2 = 圆形()
    变量1.添加图形(变量2)
```

```
        变量1.计算总面积()
"""
```

设计角度讲

定义

将相关类的共性进行抽象，统一概念，隔离变化。

适用性

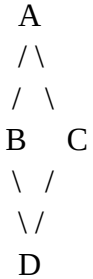
多个类在概念上是一致的，且需要进行统一的处理。

相关概念

父类（基类、超类）、子类（派生类）。
父类相对于子类更抽象，范围更宽泛；子类相对于父类更具体，范围更狭小。
单继承：父类只有一个（例如 Java，C#）。
多继承：父类有多个（例如 C++，Python）。
Object 类：任何类都直接或间接继承自 object 类。

多继承

一个子类继承两个或两个以上的基类，父类中的属性和方法同时被子类继承下来。
同名方法的解析顺序（MRO， Method Resolution Order）：
类自身 --> 父类继承列表（由左至右）--> 再上层父类




```
"""
    多继承
```

```
   17:00
"""
class A:
   def m(self):
      print("A--m")
class B(A):
   def m(self):
      print("B--m")
class C(A):
   def m(self):
      print("C--m")
class D(C, B):
   def m(self):
      # super().m()#C
      # 如果希望调用指定父类的方法,使用类名调用实例方法.
      B.m(self)
      print("D--m")
d = D()
d.m()
# python 同名方法解析顺序
print(D.mro())
```

多态

设计角度讲

定义

父类的同一种动作或者行为，在不同的子类上有不同的实现。

作用

1. 在继承的基础上，体现类型的个性化（一个行为有不同的实现）。
2. 增强程序扩展性，体现开闭原则。

语法角度讲

重写

子类实现了父类中相同的方法（方法名、参数）。
在调用该方法时，实际执行的是子类的方法。

快捷键

Ctrl + O

内置可重写函数

Python 中，以双下划线开头、双下划线结尾的是系统定义的成员。我们可以在自定义类中进行重写，从而改变其行为。

转换字符串

__str__函数：将对象转换为字符串(对人友好的)
__repr__函数：将对象转换为字符串(解释器可识别的)

```
"""
    内置可重写函数
        自定义对象 -->  str
        练习:exercise02.py
"""
class Car:
    def __init__(self, brand="", price=0,max_speed =
0):
        self.brand = brand
        self.price = price
        self.max_speed = max_speed
    # 对人友好的 --> 随心所欲的规定字符串内容
    def __str__(self):
        return "品牌是%s,单价是%d"%(self.brand,self.price)
    # 对解释器友好 --> 根据python语法规定字符串内容
    def __repr__(self):
        return "Car('%s',%d,%d)"%
(self.brand,self.price,self.max_speed)
# 应用:将对象显示出来
c01 = Car("宝马",1000000,260)
```

```
print(c01) # print(str(c01)) --> print(c01.__str__())
print(c01.__str__())
# 将括号中的字符串,作为python代码执行.
# re = eval("1 + 2")# 3
# repr(c01) -->  c01.__repr__()
# 应用:克隆(重新创建与之前对象数据相同的新对象,克隆对象与原对象互不影响)
c02 = eval(repr(c01))# eval("Car("宝马",1000000,260)")
c01.price = 50
print(c02.price)#1000000

# 练习:
# 定义技能类(技能名称,攻击比例,持续时间)
# 创建技能对象,直接print.
# 克隆技能对象,体会改变其中一个,不影响另外一个.
# 15:10
class Skill:
    def __init__(self, name="", atk_ratio=0.1, duration=0.1):
        self.name = name
        self.atk_ratio = atk_ratio
        self.duration = duration
    def __str__(self):
        return "%s---%d---%d" % (self.name, self.atk_ratio, self.duration)

    def __repr__(self):
        return "Skill('%s',%d,%d)" % (self.name, self.atk_ratio, self.duration)
s01 = Skill("降龙十八掌", 3, 5)
print(s01)  # s01.__str__()
# s02 = eval("Skill('降龙十八掌',3,5)")
s02 = eval(s01.__repr__())
s01.name = "降龙18掌"
print(s02)


"""
    练习:
        定义员工管理器,记录所有员工,计算总工资.
```

```
    程序员:底薪 + 项目分红
    测试员:底薪 + bug * 5
    要求:增加新的岗位,不影响员工管理器.
    指出:三大特征,四个原则.
    封装:员工管理器,程序员,测试员
    继承:员工隔离员工管理器与具体员工的变化
    多态:员工管理器调用员工,执行程序员,测试员.
    开闭:增加新的岗位,不影响员工管理器.
    单一:员工管理器(管理员工),员工(隔离变化),程序员(计算薪资),测试员(计算薪资)
    依赖倒置:员工管理器调用员工,而不调用程序员/测试员
    组合复用:员工管理器存储具体员工的变量
"""
class EmployeeManager:
    """
        员工管理器
    """
    def __init__(self):
        self.__all_employee = []
    def add_employee(self, emp):
        # 判断 emp 是员工 则 添加....?
        # if type(emp) == Employee:
        # emp 属于 员工
        if isinstance(emp, Employee):
            self.__all_employee.append(emp)
    def get_total_salary(self):
        total_sarlay = 0
        for itme in self.__all_employee:
            # 调用的是员工
            # 执行的是程序员/测试员
            total_sarlay += itme.calculate_salary()
        return total_sarlay
class Employee:
    """
        员工
        抽象的 --> 统一概念 --> 隔离变化
    """
    def __init__(self, base_salary):
        self.base_salary = base_salary
    def calculate_salary(self):
        """
```

```
            计算薪资
        :return: 小数类型的薪资
        """
        return self.base_salary
class Programmer(Employee):
    """
        程序员
    """
    def __init__(self, base_salary=0, bonus=0):
        super().__init__(base_salary)
        self.bonus = bonus
    def calculate_salary(self):
        # return self.base_salary + self.bonus
        return super().calculate_salary() + self.bonus
class Tester(Employee):
    def __init__(self, base_salary=0, bug_count=0):
        super().__init__(base_salary)
        self.bug_count = bug_count
    def calculate_salary(self):
        return super().calculate_salary() *
self.bug_count * 5
    # def calculate_salary(self):
    #     return self.base_salary * self.bug_count * 5
manager = EmployeeManager()
p01 = Programmer(32000, 50000)
manager.add_employee(p01)
manager.add_employee(Tester(8000, 2))
print(manager.get_total_salary())
```

运算符重载

定义：让自定义的类生成的对象(实例)能够使用运算符进行操作。

算数运算符

方法名	运算符和表达式	说明
`__add__(self, rhs)`	self + rhs	加法
`__sub__(self, rhs)`	self - rhs	减法
`__mul__(self, rhs)`	self * rhs	乘法
`__truediv__(self, rhs)`	self / rhs	除法
`__floordiv__(self, rhs)`	self // rhs	地板除
`__mod__(self, rhs)`	self % rhs	取模(求余)
`__pow__(self, rhs)`	self ** rhs	幂

反向算数运算符重载

方法名	运算符和表达式	说明
`__radd__(self, lhs)`	lhs + self	加法
`__rsub__(self, lhs)`	lhs - self	减法
`__rmul__(self, lhs)`	lhs * self	乘法
`__rtruediv__(self, lhs)`	lhs / self	除法
`__rfloordiv__(self, lhs)`	lhs // self	地板除
`__rmod__(self, lhs)`	lhs % self	取模(求余)
`__rpow__(self, lhs)`	lhs ** self	幂

复合运算符重载

方法名	运算符和复合赋值语句	说明
`__iadd__(self, rhs)`	self += rhs	加法
`__isub__(self, rhs)`	self -= rhs	减法
`__imul__(self, rhs)`	self *= rhs	乘法
`__itruediv__(self, rhs)`	self /= rhs	除法
`__ifloordiv__(self, rhs)`	self //= rhs	地板除
`__imod__(self, rhs)`	self %= rhs	取模(求余)
`__ipow__(self, rhs)`	self **= rhs	幂

比较运算重载

方法名	运算符和复合赋值语句	说明
`__lt__(self, rhs)`	self < rhs	小于
`__le__(self, rhs)`	self <= rhs	小于等于
`__gt__(self, rhs)`	self > rhs	大于
`__ge__(self, rhs)`	self >= rhs	大于等于
`__eq__(self, rhs)`	self == rhs	等于
`__ne__(self, rhs)`	self != rhs	不等于

"""

```
    运算符重载(重写)
"""
# 多态:调用父,执行子.
class Vector:
    """
        向量
    """
    def __init__(self, x=0):
        self.x = x
    def __str__(self):
        return "分量是:" + str(self.x)
    # 算数运算符 --> 返回新结果
    def __add__(self, other):
        """
            当前对象与其他数据相加时被自动调用
        :param other: 其他数据
        :return: 向量
        """
        # 如果other 是 向量
        if type(other) == Vector:
            return Vector(self.x + other.x)
        # 否则
        else:
            return Vector(self.x + other)
    def __rsub__(self, other):
        return Vector(other - self.x)
    # 复合运算符 --> 在原有对象基础上修改
    def __iadd__(self, other):
        self.x += other
        return self
```

```
    def __gt__(self, other):
        return self.x  > other.x
# 1. 算数运算符
v01 = Vector(10)
# 自定义向量 + 整数  -->  向量
print(v01 + 5)  # v01.__add__(5)
# 自定义向量 + 自定义向量  -->  向量
v02 = Vector(2)
print(v01 + v02)
# 2. 反向算数运算符
# 整数 - 自定义向量 --> 向量
print(5 - v01)  # v01.__rsub__(5)
"""
list01 = [1]
# print(id(list01))
list01 += [2]# 在原有基础上增加新元素
# print(id(list01))
# print(id(list01))
list01 = list01 + [3]# 产生新的对象
# print(id(list01))
print(list01)
"""
# 3. 复合运算符
#       默认使用算数运算符
print(id(v02))
v02 += 3  # v02 = v02 + 3
print(v02)
print(id(v02))
# 4. 比较运算符
# 自定义类 > 自定义类
print(v01 > v02)# v01.__gt__(v02)
```

设计原则

开-闭原则（目标、总的指导思想）

Open **C**losed **P**rinciple
对扩展开放，对修改关闭。
增加新功能，不改变原有代码。

类的单一职责（一个类的定义）

Single **R**esponsibility **P**rinciple
一个类有且只有一个改变它的原因。

依赖倒置（依赖抽象）

Dependency **I**nversion **P**rinciple
客户端代码(调用的类)尽量依赖(使用)抽象。
抽象不应该依赖细节，细节应该依赖抽象。

组合复用原则（复用的最佳实践）

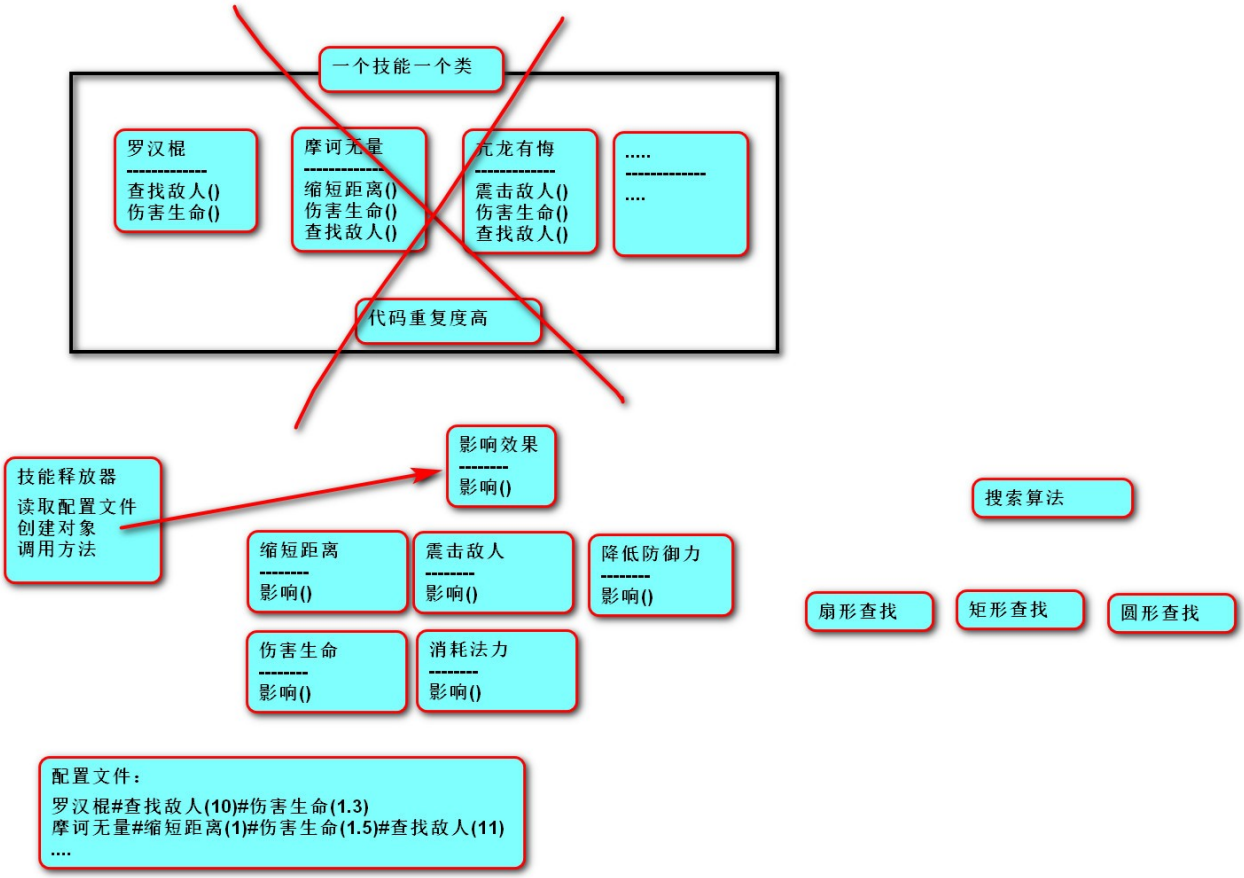
Composite Reuse Principle
如果仅仅为了代码复用优先选择组合复用，而非继承复用。
组合的耦合性相对继承低。

里氏替换（继承后的重写，指导继承的设计）

Liskov **S**ubstitution **P**rinciple
父类出现的地方可以被子类替换，在替换后依然保持原功能。
子类要拥有父类的所有功能。
子类在重写父类方法时，尽量选择扩展重写，防止改变了功能。

迪米特法则（类与类交互的原则）

Law of Demeter
不要和陌生人说话。
类与类交互时，在满足功能要求的基础上，传递的数据量越少越好。因为这样可能降低耦合度。


一个技能一个类
罗汉棍

查找敌人()
伤害生命()
摩诃无量

缩短距离()
伤害生命()
查找敌人()
亢龙有悔

震击敌人()
伤害生命()
查找敌人()
.....

....
代码重复度高
技能释放器
读取配置文件
创建对象
调用方法
影响效果

影响()
缩短距离

影响()
震击敌人

影响()
降低防御力

影响()
伤害生命

影响()
消耗法力

影响()
搜索算法
扇形查找
矩形查找
圆形查找
配置文件：
罗汉棍#查找敌人(10)#伤害生命(1.3)
摩诃无量#缩短距离(1)#伤害生命(1.5)#查找敌人(11)
....