*"""  
第九章 类和对象  
"""*# 一 、基本概念  
# 1. 对象  
# 对象（实例）：万物皆对象。对象具有属性和行为。  
# 属性： 名词  
# 行为： 动词  
# 行为和属性都是用来描述对象的。

# 2. 类  
# 一个类别：按照需求划分，将具有相同属性（名）和行为的对象划分成一个类别。

# 3. 类和对象之间的关系  
# 类是对象的抽象，对象是类的具体表现  
# 类----模板、蓝图  
# 对象--根据模板和蓝图设计出来的具体实例

# 二、类的定义  
# 1. 语法：  
"""  
class 类名:   
 类体(类的成员)  
类名是标识符：字母数字下划线组成，不能用数字开头。习惯上说首字母大写。  
"""  
# 定义一个人的类  
class Person:  
 *'注释'* pass

# 使用类产生对象  
# 语法：对象名=类名([参数]) # [参数]取决于init方法中的参数  
p1=Person()

# 2. 动态属性和方法  
# 类是产生对象，对象是由类产生  
# 属性和行为必须存在在类中。  
# 属性：变量赋值的形式 例如：a=1 name="张三"  
# 行为：函数定义的方式 例如：def run():pass  
# 函数--方法  
# 一般把类中的函数称为方法、行为  
  
# （1）动态的属性  
# 语法：对象名.属性名=属性值  
class Person:  
 *'注释'* pass  
p1=Person()  
p1.name="张三"  
#调用属性：对象名.属性名（属性名必须存在，如果不存在会报错）

# 动态创建的属性只对当前的对象有效，对其他的对象无效。

# (2) 定义动态的行为  
# 行为是以函数的模式存在的  
# 要求：函数有self参数。  
# self:代表当前对象  
def run(self):  
 print("正在跑步")  
# 调用动态方法：对象名.方法名=函数名  
p1.run1=run  
p1.run1(p1)  
# 动态创建的方法也只对当前的对象有效，对其他的对象无效。

# 动态属性和方法的局限：  
# 每个对象都要单独赋予才可以，其他对象不赋予就没有对应属性和方法。

# 3.在类中定义属性和行为（实例属性和实例方法）  
# 位置：在类的内部  
  
# 属性：变量赋值的方式：需要通过\_\_init\_\_方法实现。  
# 行为：通过函数定义的方式：实例方法中必须要有self参数，而且位置是第一个  
  
# （1）在类中定义方法（实例方法）  
class Person:  
 def run(**self**):  
 print("正在跑步")  
 def walk(**self**):  
 pass  
#调用实例方法：  
# 第一步：先有对象  
# 第二步：对象.方法名（参数除了self以外的其他参数需要传递）  
p1=Person()  
p1.run()

#（2）在类中定义属性（实例属性）  
# 语法：  
"""  
定义：  
在init方法中  
self.属性名=属性值  
  
调用：  
对象名.属性名  
  
#说明：一般实例属性都在init方法中进行创建，目的是让代码的可读性增加  
"""

# 在实例方法中访问实例属性：  
class Person:  
 def \_\_init\_\_(**self**,name,age): # 对于每个类创建之后都自带的方法，当创建对象的时候，会被调用  
 print("执行init方法")  
 **self**.name=name  
 **self**.age=age  
  
 def run(**self**,place):  
 print("{}-{}正在{}跑步".format(**self**.name,**self**.age,place))  
 # place是参数 self.name是属性  
p1=Person("张三",20)  
p1.run("操场")

# 三、类的成员  
# 包含：实例属性、实例方法、类属性、类方法、静态方法  
# 1.类属性  
# 类属性和实例属性的区别：  
# 类属性：在类中定义的属性，类属性跟具体的实例无关，只跟当前类有关系  
# 实例属性：跟具体实例相关。每个实例的实例属性值都不同。  
  
# 类属性的定义：在类的内部定义，以变量赋值的形式定义。  
class Person:  
 # 类属性  
 desc="人的描述"  
 def \_\_init\_\_(**self**,name,age):  
 # 实例属性  
 **self**.name=name  
 **self**.age=age  
 def run(**self**,place):  
 print("{}-{}正在{}跑步".format(**self**.name,**self**.age,place))  
  
# 类属性的访问： 有两种形式：  
# 可以通过对象访问：对象名.类属性名（是只读的）  
# 可以通过类名访问：类名.类属性名(推荐使用)

# 类属性的访问： 有两种形式：  
# 可以通过对象访问：对象名.类属性名（是只读的）  
# 可以通过类名访问：类名.类属性名(推荐使用)

# 不通过【对象.类属性名】对类属性进行修改，只能给对象新创建一个动态属性

2.类方法  
# 类方法和实例方法的区别：  
# 类方法：类的方法，跟具体的实例没有关系  
# 实例方法：每个实例的方法  
# 类方法的定义: 在类的内部定义函数，在函数的上方@classmethod，使用固定的参数cls（当前类）  
class Person:  
 # 类方法  
 @classmethod  
 def copy(**cls**,p):  
 print("这是类方法")  
 return Person(p.name)  
# 类方法的调用：  
# 可以通过对象:对象名.类方法名  
# 类调用 ：类名.类方法名(推荐)  
p1=Person("张三")  
# p1.copy()  
p2=Person.copy(p1)  
print(p2.name,"通过copy方法复制的p2")

# 实例方法的调用：  
# 对象名.实例方法  
p1.run() #self不需要传入  
  
# 类名调用实例方法 # 基本不使用。显示的传入p1对象  
Person.run(p1)

# 3.静态方法  
# 静态方法跟类没关系，跟实例也没有关系  
# 定义静态方法 @staticmethod  
class Person:  
 #静态方法  
 @staticmethod  
 def sm():  
 print("这是静态方法")  
 @staticmethod  
 def makefriend(p1,p2):  
 print("{}和{}交朋友".format(p1.name,p2.name))  
# 静态方法的调用 使用类名调用静态方法  
# 对象名.静态方法名  
# 类名. 静态方法名 （推荐）  
p1=Person("张三")  
p2=Person("李四")  
p1.sm()  
Person.sm()

Person.makefriend(p1,p2)

# 类和实例的选择  
# 类属性：当属性跟具体的实例没有关系，所有的实例共享同一个属性。  
# 实例属性：当属性值跟具体的实例相关。是每个实例独有的属性。  
  
# 类方法、实例方法、静态方法：  
# 如果定义的方法是对实例进行操作的，那么设计成实例方法，参数self  
# 如果定义的方法跟具体的对象没什么关系，操作的是类属性，那么定义成类方法，参数cls  
# 既不操作类属性、也不操作实例属性，选择使用静态方法。不指定固定参数  
  
# 所有的方法都是为属性服务

1. 在方法中访问属性  
   # 1.实例方法中访问类属性  
   class Person1:  
    # 类属性  
    descc="人的描述"  
    def \_\_init\_\_(**self**,name):  
    # 实例属性  
    **self**.name=name  
    # 实例方法  
    def run(**self**):  
    print("{}正在跑步".format(**self**.name))  
    #实例方法中访问类属性：对象名.类属性名，类名.类属性名  
    print("实例方法中访问类属性:",**self**.descc) # 不推荐使用  
    # print("实例方法中访问类属性:",Person.descc) # 会硬编码的影响  
    #self.\_\_class\_\_当前对象所属的类  
    print("实例方法中访问类属性:",**self**.\_\_class\_\_.descc) # 推荐

# 2.实例方法中访问实例属性  
# 对象名.实例属性名，  
# 类名.实例属性（错误）

# 3.类方法中访问类属性  
# cls.类属性名

1. 魔法方法  
   # 在python中定义了一些列的特殊方法，格式\_\_xx\_\_ 命名——魔法方法  
   # 普通的方法都需要函数的调用 函数名(参数)，函数才能执行  
   # 魔法方法：不需要主动调用，当符合一定条件的时候，会自动调用。  
   # 1. \_\_new\_\_  
   # 在创建对象的时候，用来创建对象的方法  
   # 是一个静态方法。  
   # 必须要有return，才可能够执行init方法  
   # object

# 2. \_\_init\_\_: 实例方法  
# 是在使用\_\_new\_\_创建对象之后，调用\_\_init\_\_进行初始化的方法  
  
# 3.\_\_del\_\_(self): 当销毁对象的时候，会调用这个方法  
# class Person:  
# def \_\_new\_\_(cls, \*args, \*\*kwargs):  
# print("new方法执行")  
# # super()是用来调用父类下的方法  
# return super().\_\_new\_\_(cls) # 真正创建的方法  
#  
# def \_\_init\_\_(self, name):  
# self.name = name  
# print("执行init方法")  
# def \_\_del\_\_(self): # 在垃圾回收机制的时候，对对象进行销毁  
# print("执行了del方法")  
# p1 = Person("张三")  
# p2=p1  
# del p1 # 只是删除了p1的名字  
# print("没有被销毁")

# 4.\_\_str\_\_(self)  
# 当调用了内建函数str format print时候，会自动调用的方法  
# 返回值就是字符串

# 5. repr(self)  
# 是针对解释器：当\_\_str\_\_方法不存在的时候，会自动调用repr方法

class Person:  
 def \_\_init\_\_(**self**,name):  
 **self**.name=name  
 def \_\_str\_\_(**self**):  
 print("执行了str方法")  
 # return "当前对象来源于{}类，id={}".format(self.\_\_class\_\_,id(self))  
 return "当前对象来源于{}类，id={},name={}".format(**self**.\_\_class\_\_,id(**self**),**self**.name)  
 def \_\_repr\_\_(**self**):  
 return "success"  
p=Person("张三")  
print(p)

# 6.\_\_bytes\_\_(self):  
# 当bytes()方法会自动调用的方法，方法返回值是字节类型

class Person:  
 def \_\_bytes\_\_(**self**):  
 print("执行了bytes方法")  
 return b"success"  
p=Person("李四")  
print(bytes(p))

# 7. \_\_call\_\_(self):  
# 将对象当函数一样调用  
# 对象名()  
class Person:  
 def \_\_call\_\_(**self**, \*args, \*\*kwargs):  
 print("执行了call方法")  
p=Person()

# 六、动态属性操作

# 1.hasattr(obj,name): 判断对象obj中是否存在name指定的属性名。有True 否则False

# 2.setattr(obj,name,value)  
# 将object对象的name属性设置为value

# 3.getattr(obj,name)  
#返回obj对象中name的属性值

# 4.delattr(obj,name)  
# 删除obj对象中name的属性

# 一、元类  
# 类可以创建对象  
# 类算不算对象？ 被赋值？被传递？被返回？  
# 答案：可以  
# 类也是对象

# 对象是由类产生，类是由元类产生的  
# type产生的类，type就是元类：元类就是创建类的类。  
print(type(list)) # type  
print(type(str)) # type  
print(type(Person)) # type

print(type(type)) #type 是一切的基础，用来产生类，产生type，包括自己。输出：<class 'type'>

# 1. type  
# type有两种用法  
# (1)type(object):返回当前对象的类型  
li=[1,2,3]  
print(type(li)) #list  
# (2)充当元类，创建一个类

# 直接使用type创建类的语法：  
# type(类名（字符串），继承的父类（元组）,类关联的属性和方法（字典）)

#实例方法  
def a(self,name):  
 self.name=name  
#类方法  
@classmethod  
def cm(cls):  
 return "类方法的返回值"  
def sm():  
 return "静态方法的返回值"

Person1=type("Person2",(),{"desc":"人的描述","\_\_init\_\_":a,"cm1":cm,"sm1":sm})  
print(type(Person1)) # <class 'type'>  
p1=Person1("张三")  
print(p1) #<\_\_main\_\_.Person2 object at 0x00000000025D2358>

print(Person1.desc) # 人的描述  
print(Person1.cm1()) # 类方法的返回值  
print(p1.name) # 张三  
print(Person1.sm1()) # 静态方法的返回值

# 2.自定义元类  
# 继承：借助原来type的内容，扩充自己的内容  
# （1）继承type  
# class 元类名(继承的类type):  
# pass  
# (2) 怎么调用自定义元类来创建自己的类？  
# class 类名(metaclass=元类名)：  
# PASS  
##metaclass的默认值是type  
# class Person(继承的类,metaclass=type):  
# pass  
# 元类是用来创建类的  
# \_\_new\_\_方法是用来创建对象（类）  
# 实例：自定义元类  
# class ModelMetaClass(type):  
# def \_\_new\_\_(cls, \*args, \*\*kwargs):  
# print("我是新的元类！！！！")  
# return super().\_\_new\_\_(cls, \*args, \*\*kwargs)  
# class A(metaclass=ModelMetaClass):  
# pass  
  
# 3.三个特殊的魔法方法  
# 1). \_\_new\_\_: 创建对象，静态方法  
# 2). \_\_init\_\_：在new执行之后，用来初始化  
# 3). \_\_call\_\_：把对象当函数一样调用  
# class Person:  
# def \_\_call\_\_(self, \*args, \*\*kwargs):  
# print("调用Person的call方法")  
# p1=Person() # 调用的是谁的call方法？type的\_\_call\_\_方法  
# p1()# 调用的是Person的call方法

class ModelMetaClass(type):  
 def \_\_new\_\_(**cls**, \*args, \*\*kwargs):  
 print("元类的new方法执行")  
 return super().\_\_new\_\_(**cls**, \*args, \*\*kwargs)  
 def \_\_init\_\_(**self**,\*args,\*\*kwargs):  
 print("元类的init方法执行")  
 super().\_\_init\_\_(\*args,\*\*kwargs)  
 def \_\_call\_\_(**self**, \*args, \*\*kwargs):  
 print("元类的call方法")  
 return super().\_\_call\_\_(\*args, \*\*kwargs)  
class A(metaclass=ModelMetaClass):  
 def \_\_new\_\_(**cls**, \*args, \*\*kwargs):  
 print("A类的new方法")  
 return super().\_\_new\_\_(**cls**)  
 def \_\_init\_\_(**self**, \*args, \*\*kwargs):  
 print("A类的init方法")  
 super().\_\_init\_\_(\*args, \*\*kwargs)  
 def \_\_call\_\_(**self**, \*args, \*\*kwargs):  
 print("A类下的call方法")  
 def fun(**self**):  
 print("fun方法执行")  
  
"""  
ModelMetaClass 爷爷  
A 父亲  
a 儿子  
1. 当定义了元类ModelMetaClass之后，只要定义一个类A使用了这个元类创建，就会调用元类下  
 的new方法和init方法（无论这个A类有没有产生对象）  
2. 当使用这个A类创建对象a的时候，会先调用元类ModelMetaClass下的call来进行对象的创建A()，  
 接着调用A类下new方法和init方法  
3. 当调用a()，会调用A类下的call方法  
4. 当调用a下的实例方法时，不会调用任何new init 或者call，只调用自己的实例方法。  
  
# 一个对象（A类下的a）的需要有哪些方法执行？  
（1）元类下的call  
（2）类下的new方法  
（3）类下的init方法  
"""  
a=A()

元类的new方法执行

元类的init方法执行

元类的call方法

A类的new方法

A类的init方法  
a()

A类下的call方法  
# a.fun()

# object（继承，只是为了减少代码冗余） type(真正用来创建对象（类）)  
# type是object的子n类，但是object是type创建的。  
# issubclass(子类，父类)#

print(issubclass(type,object))  
print(issubclass(object,type))  
print(type(object))

True

False

<class 'type'>

# 二、元类的应用  
# 元编程都是设计层面的知识  
# 所有的类总是会直接或者间接的继承object  
from datetime import datetime  
class ModelMetaclass(type):  
 *"""  
 name: 使用元类ModelMetaclass创建的类类名  
 bases：使用元类ModelMetaclass创建的类需要继承哪些父类（元组）  
 attrs: 使用元类ModelMetaclass创建的类下的属性和方法（字典）  
 """* def \_\_new\_\_(**cls**, name,bases,attrs):  
 attrs["a"]=123456  
 attrs["time"]=datetime.now()  
 for k,v in attrs.items():  
 print(k,v)  
 return super().\_\_new\_\_(**cls**,name,bases,attrs)  
class A(metaclass=ModelMetaclass):  
 pass

# 1. 创建元类，使得使用这个元类创建的类不能够实例化对象：抽象类(只能用来被继承，不能用来实例化对象)。  
# class Card:  
# pass  
# class CreditCard(Card):  
# pass  
# class SaveCard(Card):  
# pass  
class NoIntance(type):  
 def \_\_call\_\_(**self**, \*args, \*\*kwargs):  
 # # assert 1>2,"不能创建对象"  
 raise ValueError("不能创建对象") # 专门用来主动抛出异常

# 2. 创建一个元类，使得使用这个元类创建的类不能够被继承 final类  
class Final(type):  
 def \_\_new\_\_(**cls**,name,bases,attrs):  
 for k in bases:  
 # isinstance(对象a，类A)：对象a是不是A产生的对象  
 if isinstance(k,Final):  
 raise TypeError("不能够继承当前类{}".format(k.\_\_name\_\_))  
 return super().\_\_new\_\_(**cls**,name,bases,attrs)  
  
# object是type产生的类  
# class Father(object,metaclass=Final):  
# pass  
# class Son(Father):  
# pass  
# son=Son()

#3. 使用元编程完成单例模式  
# 原本一个类可以创建无数个不同的对象，  
# 单例模式：一个类只能创建一个对象  
# 在类中创建一个实例属性=None  
# 在call中判断，如果实例属性是None，那么就可以创建新对象  
# 如果实例属性不是None，直接返回原来的实例对象  
class Singleton(type):  
 # def \_\_init\_\_(self,\*args,\*\*kwargs):  
 # self.instance=None  
 # super().\_\_init\_\_(\*args,\*\*kwargs)  
 \_instance=None  
 def \_\_call\_\_(**self**, \*args, \*\*kwargs):  
 if **self**.\_\_class\_\_.\_instance is None:  
 **self**.\_\_class\_\_.\_instance=super().\_\_call\_\_( \*args, \*\*kwargs)  
 return **self**.\_\_class\_\_.\_instance  
 else:  
 return **self**.\_\_class\_\_.\_instance  
# 元类下的new和init是class A的时候执行的。  
class A(metaclass=Singleton):  
 pass

# 利用new或者init：会执行A的new和init  
class B:  
 \_instance=None  
 def \_\_new\_\_(**cls**,\*args,\*\*kwargs):  
 if **cls**.\_instance is None:  
 **cls**.\_instance= super().\_\_new\_\_(**cls**,\*args,\*\*kwargs)  
 return **cls**.\_instance  
 else:  
 return **cls**.\_instance

*"""  
第十章： 面向对象的特征  
"""*# 一、 面向对象和面向过程  
"""  
面向过程：将程序按部就班的执行，为了实现代码的复用性，通过设计函数来实现，堆积大量的函数。  
 遇见问题之后，直接按部就班解决  
   
面向对象：根据需求，分析出对象，根据需求将对象划分成类，类中属性、行为，将类和类之间进行关联。  
 遇见问题之后，谁？对象下的属性和方法。  
"""

"""  
对比面向过程的缺陷：  
1. 员工不能审批，领导也不用请假，主谓宾分的不清晰  
2. 复杂的数据不能存储，当需求发生改动的时候，改动量极大，方法太多，无法分清方法功能

# 面向过程适合解决：一成不变的需求。操作系统，纯函数

# 面向对象适合解决：需求经常变动场景。

# 二、面向对象的三个特征  
# 封装、继承、多态  
# 1. 封装： 信息的隐藏  
# 作用：  
# （1）数据的安全性  
# （2）方便调用者调用: 内部功能如何修改，不影响调用者的调用者调用方式。

# 1）.成员私有化  
# 类属性、类方法、实例属性、实例方法、静态方法  
# 将类中定义的变量进行私有化，使得变量只能在当前类中访问，不能在外部访问，如果访问需要指定接口。  
# 如果需要将一个属性名或者方法名私有化  
# 命名格式： \_\_开头（不能以两个\_\_结尾）

#如果希望在类的外部访问私有成员，需要提供接口（方法get\set）

# 在python中，私有变量是个假的私有变量  
# 通过将私有变量名字: \_\_变量名  
# 内部其实将名字封装成： \_类名+私有成员的名字

# print(c.\_Computer\_\_memory)  
# c.\_Computer\_\_memory=111111

# 2）. propery  
# 为了调用者方便调用私有成员  
# 两种方式：  
# （1）property函数  
# （2）property装饰器  
  
  
# （1）property函数  
class Computer:  
 def \_\_init\_\_(**self**):  
 **self**.\_\_memory="128g"  
 **self**.cpu="intel5"  
  
 def setmemory(**self**,memory):  
 **self**.\_\_memory=memory  
 def getmemory(**self**):  
 return **self**.\_\_memory  
 def delmemory(**self**):  
 del **self**.\_\_memory  
 # 加入property函数，将私有成员使用property改名字  
 # 语法： 对外名字 = property(获得私有成员的方法，设置私有成员的方法，删除私有成员的方法，注释)  
 # memory是 被property过的：memory是私有属性，但是已经使用propery提供了方便的调用方式  
 memory= property(getmemory,setmemory,delmemory,"内存大小")  
 memory= property(getmemory,setmemory,None,"注释")  
  
c=Computer()  
# print(c.getmemory())  
# c.setmemory("256g")  
  
# 使用property的变量来访问  
c.memory="256g"  
print(c.memory)  
# del c.memory  
  
c.cpu="intel7"  
print(c.cpu)

# （2）property装饰器  
# @property  
# 注意：当使用 @property就要定义好要对外提供的名字是什么。  
class Computer:  
 def \_\_init\_\_(**self**):  
 **self**.\_\_memory="128g"  
  
 # 对外提供的属性名 就是get、set、del方法的名字  
 # 一般先写get方法:例如 对外提供的属性名字memory，就将get获取方法写成memory  
 # 在对应get和set、del方法上加装饰器  
 # 在get方法上加的装饰器： @property  
 # 在set方法上加的装饰器： @对外属性名字.setter  
 # 在del方法上加的装饰器： @对外属性名字.deleter  
 @property  
 def memory(**self**):  
 return **self**.\_\_memory  
  
 @memory.setter  
 def memory(**self**,memory):  
 **self**.\_\_memory=memory  
  
 @memory.deleter  
 def memory(**self**):  
 del **self**.\_\_memory  
  
 def \_\_a(**self**):  
 print("执行\_\_a方法")  
 def exec\_a(**self**):  
 **self**.\_\_a()  
  
c=Computer()  
print(c.memory)  
c.memory="256g"  
print(c.memory)  
# c.\_\_a() # 不能调用  
c.exec\_a()

# 2.继承

# （1）背景  
# 元类是用来产生类的。  
# 继承是用来继承代码（成员）的。

# 代码冗余问题：  
# 1. 写的时候冗余，读的时候不易懂  
# 2. 代码的扩展性不强  
  
# 解决方式：使用继承解决类和类之间的相似问题。

# （2）继承实现  
"""  
继承描述关系：  
继承实现是一种【一般】与【特殊】的关系。  
水果类---一般性-----父类----父类  
苹果类---特殊性-----子类----扩展类、衍生类  
  
子类应该继承父类  
  
子类继承父类之后，父类中有的成员（属性、方法等等），子类中也有。（子承父业）  
就像直接定义在子类中一样。  
"""  
  
#继承方式分为两种：显式继承、隐式继承  
# 第一种范式：显式继承：  
"""  
语法：  
class 类名(父类):  
 类体  
"""  
class Fruit:  
 def show(**self**):  
 print("我是水果")  
class Apple(Fruit):  
 pass  
  
# 第二种：隐式继承：专门指如果不写明继承父类，则默认继承object类  
# A偷偷的继承object  
class A:  
 pass  
# 新式类 经典类  
# object  
  
# (3) 继承意义  
# 通过继承，可以实现代码的重用性，把公共功能或者属性提取出来方法父类中，每一个子类继承父类，子类中就会具有父类的所有方法和属性，子类还可以在此基础上进行扩展。  
  
# (4) 两个内建函数  
# isintance(对象，类)：第一个参数是否属于第二个参数(包含父类)产生的  
# issubclass（子类，父类）：第一个参数是否是第二个参数的子类(自己也是自己的子类)

# (5)成员的继承  
# 类属性、类方法、实例属性、实例方法、静态方法 五个成员  
# ① 类属性、类方法、实例方法、静态方法 的继承

# 重写时，在子类中调用父类的方法，语法：super().父类的方法  
# super().父类方法，进行继承的时候，会将方法全部引用过来，不能只继承一部分。就像使用复制，粘贴将父类的代码粘贴过来一样。 将父类中定义的变量继承过来之后，可以进行修改。

# ③实例属性的继承  
# 实例属性是在\_\_init\_\_方法中定义  
# 实际上，实例属性的继承，就相当于重写父类中的\_\_init\_\_方法

# class Bike:  
# def \_\_init\_\_(self):  
# self.color="白色"  
# self.\_\_size=28  
# class CvBike(Bike):  
# def \_\_init\_\_(self):  
# super().\_\_init\_\_() # 属性继承的时候，建议将super放在第一行  
# self.color="蓝色" # 修改父类的属性  
# self.cv="变速器" # 扩展父类的属性

# ④ 私有成员的继承  
# 子类是允许访问父类的私有成员。print(cvBike.\_Bike\_\_size)# 如果访问真实名字，也可以访问到，但是不建议这样写。

# （6） 多重继承  
# 一个子类可以继承多个父类  
# 多个父类使用,分隔  
# 正方形是特殊的矩形，特殊菱形

# 当前方法找不到的时候，也会按照mro原则找到方法

# 继承原则 MRO原则：继承的顺序  
# 如果多个父类中的方法和属性都不同名，一般不需要查看MRO原则  
# 如果多个父类中出现重名的方法或者属性，才需要按MRO搜索具体继承的是那个属性或者方法。  
  
# 对于树形结构的两种MRO遍历方法：  
# 深度优先  
# 广度优先

"""  
python2.2 之前的版本，经典类方式实现类。  
class A(object):  
MRO：深度优先  
  
python2.3--python2.7 : 经典类和新式类并存  
2.3  
新式类： MRO广度优先  
经典类： MRO深度优先  
  
2.3-2.7，C3算法  
C3分为两种情况：一种有统一父类的情况，另一种无统一父类的情况  
  
python3 ,类只有新式类：不写(object)也默认继承object  
使用C3算法：  
分为：  
有统一父类（环状结构）：一直深度优先到查看到有统一父类，变为广度优先。  
没有统一父类（非环状结构）：深度优先  
"""

# 练习一  
class A:  
 pass  
class B:  
 pass  
class C(A,B):  
 pass  
class D(A,B):  
 pass  
class E(C,D):  
 pass  
print(E.mro())

[<class '\_\_main\_\_.E'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.D'>, <class '\_\_main\_\_.A'>, <class '\_\_main\_\_.B'>, <class 'object'>]

# 练习二  
class F:  
 pass  
class E:  
 pass  
class D:  
 pass  
class C(D,F):  
 pass  
class B(E,D):  
 pass  
class A(B,C):  
 pass  
print(A.mro())

[<class '\_\_main\_\_.A'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.E'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.D'>, <class '\_\_main\_\_.F'>, <class 'object'>]

# 如果有多个父类，多个父类又有相同的方法，比如area方法，如果非要想继承菱形的area，这么办？  
# 1. 菱形写在继承的第一个位置 (推荐)  
# 2. 可以使用 父类.实例方法 （可以做到，但是属于 类.实例方法 ----并不好）

class Father:  
 pass  
class Son(Father):  
 pass  
  
# 多态对象  
# Father f=Father()  
# Son s=Son()  
# Son s\_new=Father()  
  
# 多态参数  
# class A:  
# def talk(self,Father father):  
# pass  
# a=A()  
# a.talk(s)

# 多态子类

# Cat Dog  
class Animal:  
 def run(**self**):  
 pass  
class Cat(Animal):  
 def run(**self**):  
 print("猫在跑步")  
class Dog(Animal):  
 def run(**self**):  
 print("狗在跑步")  
def run(a):  
 a.run()  
cat=Cat()  
dog=Dog()  
run(cat)  
run(dog)

*"""  
第十一章 模块和包  
"""*# 一、模块  
# 1.模块的基本定义  
# 模块的定义：一个py文件就是一个模块。————物理的角度  
# 模块中包含：定义的语句（类、函数、变量）  
# 模块在内存中存储的名字就是py文件的文件名。  
# 为什么划分模块？  
# 代码太多，不能都存在在一个文件中；会出现名称冲突  
# 划分模块之后的好处：  
# 1. 定义模块有利于将项目按照功能进行划分，方便协作开发  
# 2. 模块提供了独立的命名空间（全局命名空间），解决了命名冲突问题  
# 3. 模块可以供多个人使用，提高了程序的复用性。  
  
# 2.模块的使用  
# 模块和模块之间的访问：通过导入  
# （1）使用import导入  
# 导入模块的格式：import 模块名（如果有多个模块，可以是用,分隔）  
# import random,math  
# 当我们在代码上import 模块，计算机执行了什么内容？  
# 当导入了一个模块之后，被导入模块中的语句就会被执行。只执行一次。  
# 按照惯例，模块的导入模块的最上方  
  
# 调用导入模块的格式：模块名.名称  
# 使用import导入的模式，名字不会跟被导入模块中的名字冲突

# （2） from ...import...  
# 通过指定要导入的内容来导入  
# 被导入的模块都会在导入之后，执行一次，而且只执行一次，跟import是一样的。  
# 导入语法格式： from 模块名 import 名称1，名称2.....  
# 调用导入模块的语法： 直接使用名称1，名称2

# 注意：from import的方式因为使用的是名字，所以容易跟当前模块中的名字冲突  
# 后者覆盖前者

# 3. 模块的别名  
# 当导入模块时，可以使用as来为模块中的名称指定别名  
# 语法  
"""  
import 模块名 as 模块别名, 模块名 as 模块别名  
from 模块名 import 名称1 as 名称1别名，名称2 as 名称2别名,  
"""  
# 当名称有别名了之后，原名字失效。

# 4.隐藏模块的数据  
# 注意：只对from 模块 import \*这种情况有效，对于import无效  
# 下面两种方式中，如果一个带有\_的变量放在\_\_all\_\_，程序会认为可以导入。  
# from another1808 import \*  
# 可以在使用from import的时候，选择性导入名称。  
# 方式有两种：  
#（1）指定不能被导入的名字： 名字前面使用\_，这样的名字不会被from import \* 导入。  
# print(\_w)  
#（2）指定能被导入的名字： 在模块内定义 \_\_all\_\_，关联列表，列表中的元素是能够被导入的名字

# 5.\_\_name\_\_  
# 当前模块的名称  
# 对于一个py文件来说，执行方式有两种：  
# （1）直接run :\_\_main\_\_  
# （2）被导入 :another1808

# 当导入模块中有不希望被导入时执行的代码，  
# 可以加入if \_\_name\_\_=="\_\_main\_\_"判断。

# 6.模块的搜索路径  
#当import 名称，按照如下的路径搜索名称  
"""  
 (1) 内置解释器中搜索  
（2）作为脚本执行的模块所在路径  
（3）python的path环境变量指定路径下  
（4）python的安装路径（lib）  
"""

# 注意：类名、方法名、变量名、模块名 都不要跟内建命名空间的名字和lib包下的名字一样。

# 7. 模块缓存  
# 导入模块的时候，只会执行一次被导入的模块  
#是因为python在导入模块时，会缓存加载模块，产生pyc文件，是字节码文件，也叫预编译  
# 什么时候创建？创建在哪？  
# （1）自定义模块被导入的时候，会产生  
# （2）会创建到，同路径下的\_\_pycache\_\_的包下  
# 作用：加快模块的加载速度，因为第二次加载直接读取缓存文件。不会加快模块的执行速度。

# 1. 包的概念  
# 包的概念：包提供了独立的命名空间有，类似于window系统中的文件夹。  
# 注意：跟文件夹不同的是，python开发包必须要有\_\_init\_\_.py  
# 普通的文件夹不能被导入，但是包可以被导入  
  
# 2.包的使用  
# 两种导入方式  
# import 包名  
# import 包名.模块名  
  
# from 包名 import 模块名  
# from 包名.模块命名 import 变量名，函数名。。。

# 3.\_\_init\_\_.py  
# python的每一个开发包下，都必须要init文件：用途是用来初始化的。  
# 当import [包] 的时候，会执行当前包中\_\_init\_\_.py中的所有代码。  
# \_\_init\_\_中一般都使用来初始化包中的变量。  
# \_\_init\_\_.py文件中定义的变量或者函数或者类等等，一些列的名字，在整个包的所有py文件中都可以访问到

# 在py文件中访问当前包中的init文件，语法：  
# 【包名】.属性名  
# import day12

# print(day12.x) # 使用import的形式访问

# 4. \_\_all\_\_变量  
# all是一个列表，可以在列表下加入可以导入的名字

*三、数学模块math*

# math  
import math  
# 1.圆周率  
print(math.pi)  
  
# 2. 数学常数  
print(math.e)  
  
# 3.向上取整  
print(math.ceil(5.3))  
print(math.ceil(-5.3))  
  
# 4.向下取整  
print(math.floor(5.3))  
print(math.floor(-5.3))  
  
# 5.返回e的x次幂  
print(math.exp(1))  
  
# 6.返回x的y次幂  
print(math.pow(2,7))  
  
# 7.返回以base为底，x的对数,base默认以e为底  
# print(math.log(x,base))  
print(math.log(100,10))  
  
# 8.fabs返回浮点类型的绝对值  
print(abs(-1.0))  
print(math.fabs(-3.4))  
print(math.fabs(-3))  
  
# 9.factorial 阶乘  
print(math.factorial(3))  
  
# 10. 取余，商是正数：向下取整；商是负数，向上取整  
print(math.fmod(5,3))  
print(math.fmod(5,-3))  
  
# 11.求累加和，结果是浮点类型  
print(math.fsum([1,2,3,4]))  
  
# 12. 返回平方根  
print(math.sqrt(100))  
  
# 13.返回最大公约数  
print(math.gcd(12,15))

# random产生随机数模块  
import random  
# 1.返回一个0到1之间的浮点数，包括0不包括1 [0,1)  
print(random.random())  
# while random.random()!=0:  
# pass  
# print("已经获得了")  
  
# 2. randint(a,b)产生 a<=x<=b 包含起止点  
print(random.randint(1,3))  
  
# 3.包含start，不包含end  
# random.randrange(start,stop,step)  
print(random.randrange(1,10,2))  
  
# 4.random.uniform(a,b) 可以产生 a<=x<=b 浮点数，闭区间  
# print(random.uniform(2.3,2.5))  
  
# 5. choice(seq),从迭代对象中随机选择一个元素  
print(random.choice([1,3,5,6,7,8]))  
  
# 6. choices(seq,权重,k)k是取几个值  
print(random.choices([1,2,3,4,5],weights=[1,1,1,1,10000],k=3))  
  
# 7.sample(seq,k) k是抽取的个数  
# 跟choices不同在于取完元素之后，不放回  
print(random.sample([1,2,3,4,5],k=3))  
  
# 8.shuffle(seq) 可以对列表进行打乱顺序后重新输出列表  
# 就地洗牌  
li=[1,2,3,4,5]  
random.shuffle(li)  
print(li)

*"""时间模块"""*# 一、time包：都是关于时间的操作  
import time  
# UTC：世界标准时间，本初子午线上时间  
# 1. timezone: 返回与UTC时间相差的秒数  
# 2.time.time()返回从新纪元到当前时间走过的秒数，小数是微秒  
# 新纪元：unix产生时间1970-1-1  
print(time.time())  
# 3.time.localtime([s])返回从新纪元走过的s秒之后的时间，返回的是时间元组  
# 如果不写s，默认返回的是本地当前时间  
# 4.gmtime([s]) 返回从新纪元走过的s秒之后的时间, 返回值是utc时间  
print(time.gmtime(1))  
print(time.gmtime())  
  
# 5. time.mktime(tuple) 将时间元组转换成秒数，秒数---从新纪元到tuple时间元组的时间  
t=time.localtime()  
print(time.localtime(1536983039))  
print(time.localtime(1536983039))  
print(time.mktime(time.localtime()))  
  
# 6.asctime（tuple） 将时间元组转换成字符串  
t=time.localtime()  
print(time.asctime(t))  
  
# 7.time.sleep(s) 使得当前程序暂停s秒，注意，包含cpu执行的时间，所以不到1s  
# time.sleep(1)  
# print("dddd")  
  
# 8.clock() :  
"""  
linux、unix返回cpu的计算时间  
windows 函数第一次调用，返回的是cpu的计算时间，从第二次调用开始，返回的是距离第一次调用  
 所经历的时间  
"""

# 9.返回精准的性能计数器，时间包含调用sleep函数暂停的时间  
# start=time.perf\_counter()  
# time.sleep(1)  
# end=time.perf\_counter()  
# print(end-start)  
  
# 10.time.process\_time() 返回精准的计数器，不包含sleep函数调用的时间  
# start=time.process\_time()  
# time.sleep(1)  
# end=time.process\_time()  
# print(end-start)  
  
#11. time.strftime(format,元组): 将时间元组转换成字符串  
t=time.localtime()  
print(time.strftime("%Y-%m-%d %H:%M:%S",t))  
  
# 12 time.strptime(str，format)：将字符串转换成时间元组，要求str必须符合format格式的标准，否则无法转换成元组，报错  
print(time.strptime("2018-09-15 12:00:02","%Y-%m-%d %H:%M:%S"))  
print(time.strptime("2018-09/15 12:00:02","%Y-%m/%d %H:%M:%S"))

#二、datetime  
# 分为三个类：time date datetime  
from datetime import date  
# (一)date类  
#1. 构造器: 就是类名.\_\_init\_\_方法。（因为init方法是初始化实例属性的方法。）  
# date(year,month,day) :用来创建参数指定日期类型对象  
print(date(2018,10,12)) #2018-10-12 是因为在date类下重写str方法

#2. 实例属性  
# year  
print(d1.year)  
# month  
print(d1.month)  
# day  
print(d1.day)  
# d1.day=55 #propery之后，只提供了get方法，是只读的，相当于date下是这样设计的  
# class date:  
# def \_\_init(self,year,month,day):  
# self.\_\_year=year  
# self.\_\_month=month  
# self.\_\_day=day  
# def getyear(self):  
# return self.\_\_year  
# year=property(getyear)

#3. 类属性  
# max :最大的对象 9999-12-31  
print(date.max)  
# min：最小 0000-01-01  
print(date.min)  
# resolution: 两个对象之间最小的间隔,date为天  
print(date.resolution)

# 4.实例方法  
d1=date(2018,10,12)  
# （1）d1.ctime() 返回特定格式的字符串来表示对象的日期  
print(d1.ctime())

# (2)d1.replace(year=年，month=月，day=日)，不是原地修改，新创建日期进行修改  
d1.replace(month=9)  
print(d1.replace(month=9))  
print(d1.replace(day=9))

# (3) 可以返回对象的时间元组  
# 类似于time.localtime() 不写参数返回当前时间  
print(d1.timetuple() ) # 返回当前对象的时间 元组  
  
# (4)d1.weekday()返回当前的日期对象是星期几 （0-6）0是星期一 6是星期日  
print(d1.weekday())  
  
# (5)返回当前日期的序数：0001年1月1日 序数1  
print(d1.toordinal())  
  
# (6)d1.strftime(format) 将日期类型的对象格式化成字符串  
print(d1.strftime("%Y-%m-%d"))  
print(d1,type(d1))  
a=d1.strftime("%Y-%m-%d")  
print(a,type(a))

# 5. 类方法： 返回date类型的对象  
# （1）date.today()# 返回当前日期的date对象。  
print(date.today())  
#（2）date.fromtimestamp(时间戳) 时间戳：从新纪元时间走过的秒数  
print(date.fromtimestamp(0)) # 1970-01-01  
# (3)date.fromordinal(ordinal) 根据指定的ordinal参数返回date对象  
print(date.fromordinal(1)) # 0001-01-01

# （二）time类  
# 针对时间的操作  
from datetime import time  
# 1.构造器  
# time(hour,minute,second,microscond) microscond是有默认值0  
t=time(3,4,5)  
  
#2.实例属性  
print(t.hour)  
print(t.minute)  
print(t.second)  
print(t.microsecond)

#3. 类属性  
# max min resolution  
print(time.max) # 23:59:59 999999  
print(time.min)  
print(time.resolution) # 1微秒

# 4.实例方法  
#（1）t.replace(hour=,minute=,second=,micorsecond=)

#(2) t.strftime()将time类型的数据转换成字符串  
s=t.strftime("%H:%M:%S")

# （三）.datetime类  
# 针对日期和时间的操作，date和time两个功能并集  
# 1.构造器  
from datetime import datetime  
# datetime(年，月，日，时，分，秒，微妙=0)  
print(datetime(2018,10,1,1,1,1))

# 2.实例属性

# 3.类属性

#4. 实例方法  
dt=datetime(2018,10,1,1,1,1)  
# （1）dt.date() 将datetime类型的对象，返回一个date类型的对象。  
d=dt.date()  
print(d,type(d))  
# (2) dt.time() 将datetime类型的对象，返回一个time类型的对象。  
t=dt.time()  
print(t,type(t))  
# (3)ctime（）返回特定格式的字符串  
print(dt.ctime())  
  
#(4) dt.replace(year=,month=,day=,hour=,minute=,second=,microsecond=)  
print(dt.replace(second=24))  
  
#(5) dt.timetuple() 将当前对象转换成时间元组  
print(dt.timetuple())  
  
# (6) dt.weekday()，返回当前日期对象的星期几  
print(dt.weekday())  
  
# (7)dt.toordinal() 返回当前datetime对象的时间序数  
print(dt.toordinal())  
  
#（8）dt.strftime(fmt) 将datetime类型的对象转换成字符串  
print(dt.strftime("%Y-%m-%d %H:%M:%S"))  
  
#(9)dt.strptime(fmt) 将字符串转换成datetime类型的对象  
print(dt.strptime("2018-10-01 01:01:01","%Y-%m-%d %H:%M:%S"))

# 5.类方法  
# （1）datetime.today() 返回当前日期的datetime  
print(datetime.today())  
# (2) datetime.fromtimestamp(时间戳) 根据指定时间戳返回datetime类型的对象  
# print(datetime.fromtimestamp(3600\*24))# 只支持1天以上的  
# print(datetime.fromtimestamp(3600\*26))  
#（3）datetime.fromordinal(ord) 根据参数返回datetime类型对象  
# print(datetime.fromordinal(1))  
# (4)print(datetime.now()返回当前日期的datetime，同today  
print(datetime.now())  
# （5）datetime.utcfromtimestamp()  
# 从1970-1-1，经历了参数指定的时间戳，创建datetime（utc的对象）  
print(datetime.utcfromtimestamp(1))  
  
# (6)datetime.strptime(str,fmt) # 将字符串转换成datetime类型的对象  
dt1=datetime.strptime("2018-10-01 01:01:01","%Y-%m-%d %H:%M:%S")  
print(dt1,type(dt1))

# 三、calendar日历包  
import calendar  
print(calendar.month(2016,1))  
print(calendar.calendar(2018))