**面向对象 Object oriented**

学习面向对象 XXOO

面向对象的学习组成部分：

面向对象的语法(简单，记忆)

面向对象的思想(稍难，理解)

**面向过程开发和面向对象的区别**

面向过程的开发主要以函数方式进行

吃饭 -> 吃饭的函数

喝水 -> 喝水的函数

洗衣服 -> 洗衣服的函数

看电影 -> 看电影的函数

看片 -> 看片的函数

....

面向对象开发使用类和对象方式进行

女朋友{

吃饭 -> 吃饭的函数

喝水 -> 喝水的函数

洗衣服 -> 洗衣服的函数

看电影 -> 看电影的函数

看片 -> 看片的函数

....

}

面向对象的特色：高内聚，低耦合！

**面向对象的几个单词**

OO 面向对象

OOA 面向对象的分析

OOD 面向对象的设计

OOI 面向对象的实现

OOP 面向对象的程序开发

OOA->OOD->OOI 面向对象的实现过程

**类和对象的概念**

**什么是类？（抽象的不具体的内容称之为类，唯心主义）**

人类 是一个类

动物类 也是一个类

鸟类 也是一个类

女朋友 就是一个类

基友 也是一个类

概念汽车 就是一个类

概念手机 也是一个类

类就是实物的集合的一个概念，特点是抽象的不具体的物体

打蛋蛋 也是一个类

泡妹子 也是一个类

跑路 也是一个类

SM 也是一个类

行为和操作的概念集合也是一个类，特点是没有实施过，也没有特指某次操作

**什么是对象（现实的具体的内容称之为对象，唯物主义）**

曹星是一个对象

牛穷也是一个对象

冯伟的小鸟也是一个对象

武强的女朋友也是一个对象

卜令涛的基友也是一个对象

樊新的汽车 是一个对象

丛浩的手机也是一个对象

实际上存在的看得见摸得着的实物就是一个对象

李云云打牛穹的一次蛋蛋

李冲冲泡了一次樊新

卜令涛带着他的小姨子在昨天跑了！

李强和杨磊昨天在7天201房间玩SM

真实发生的行为就是对象。

**类和对象的关系？**

类是由对象抽象总结而来的，

由对象总结出类的过程，称之为抽象化

对象是由类具体实施或者实现而来

有类制作出对象的过程，称之为实例化

**类中的内容**

类中具有2个内容：

用于表示事物特征相关的内容，称之为成员属性（变量）

用于表示事物功能相关的内容，称之为成员方法（函数）

**如何书写一个类文件**

类文件命名采用驼峰名法：

保证每个单词的首字母都要大写

例如： MyCar 类 Image类 DoYourSister类

**如何声明一个类**

类名：类名和文件名保持一致，还要遵守变量命名规范。

类名采用驼峰命名法：

普通变量和函数：

除了第一个单词之外，所有其他单词首字母大写

变量： myWife

函数： makeLove

类的命名

所有单词首字母大写。

类名：MyCar

其他命名规范：

my\_sister,my\_girl.....

特征：

1.声明类必须使用class关键字

2.类中内容由2部分组成：成员属性和成员方法，其他内容禁止在类中出现，比如流程控制，循环语句等操作，语法没错，但是破坏了类的结构。

3.定义成员属性使用变量赋值操作即可，如果某个成员没有值，需要使用None来赋值

4.定义成员方法使用函数声明规则即可。

**实例化类**

变量 = 类名（） 实例化了一个对象

**访问对象成员**

对象.成员属性名

对象.成员方法名()

**访问类中成员**

类.成员属性名

类.成员方法名()

**检测类和对象的所有成员**

对象所有成员检测：

对象变量.\_\_dict\_\_

类所有成员检测：

类名.\_\_dict\_\_

类和对象的成员分析

1.类和对象都可以存储成员。成员可以归类所有，也可以归对象所有

2.类存储成员时使用的是与类关联的一个类对象

3.对象存储成员时就是存储在当前对象当中

4.在对象中访问一个成员时，如果对象中没有该成员，尝试访问类中的同名成员，如果对象中有该成员呢，一定使用对象中的成员。

5.创建对象时，类中的成员不会进入对象当中，而是得到一个空的对象，没有成员。

6.通过对象对类中成员重新赋值或者通过对象添加成员时，对应成员会保存在对象中，而不会修改类成员

**关于self**

1.self的英文意思就是自己

2.self在对象的方法中表示当前对象本身，如果通过对象调用一个方法，那么该对象会自动传入到当前方法的第一个参数中。

3.self单词并不是关键字，只是一个用于接收对象的普通形参名称，所以可以使用任何变量名代替

4.方法中有self形参（并且设定为接收对象使用的参数）的方法称之为非绑定类的方法，可以通过对象访问，没有self的就是绑定类的方法，只可以通过类来访问

5.使用类访问绑定类的方法时，如果类方法中需要访问当前类的成员（无论属性还是方法），可以通过\_\_class\_\_.成员名 来访问类中其他成员（对象对应的是self.成员名）

self不是关键字！！！self不是关键字！！！self不是关键字！！！

注意：一个方法是不是绑定类的方法不取决于方法本身的结构，而是取决于如何调用该方法，将该方法的形参当作接收什么值的形参使用

class Demo:

def test(self):

pass

当作绑定类的方法

Demo.test(随意参数)

当作非绑定类的方法

demo = Demo()

demo.test()

------

**面向对象三大特性：封装，继承和多态**

**封装**

封装就是将对象的成员进行访问限制，限制访问的位置

封装的三个级别：

公开的 public (不是关键字，就是英文翻译）

受保护的 protected (不是关键字，就是英文翻译)

私有的 private （不是关键字，就是英文翻译）

检测封装的位置：

类/对象的内部

类/对象的外部

子类中

**私有化封装 private**

私有化封装是将对象成员最高级别的封装。封装之后，只能在当前类/对象中访问，无法在类的外部和，子类中访问

封装方法： 在成员名称前添加 \_\_即可（2个下划线）

例如： heart 私有化封装操作 \_ \_heart

注意：在Python中私有化封装采用的是name mangling的改名策略实现的，本质上只是把成员名改了导致原有成员名无法访问到，可以使用对象.\\_类名\\_\\_成员名的方式访问

（没事别这么干）

访问限制及方式：

类/对象中 可以访问 使用\_ \_成员名访问

类外/对象外 不可以访问 无

子类中 不可以访问 无

**受保护的封装 protected**

受保护的封装是将对象成员进行一定级别的封装，封装之后，在类中或者子类中都可以正常访问，但是无法在类外部正常访问

封装方法： 在成员名称前添加\_即可（1个下划线）

例如：money 受保护的封装 \_money

注意：在Python封装时依旧采用name mangling方式实现，正常情况下使用成员名无法在类外访问，但是通过 添加\_也可以访问

访问限制及方式：

类/对象中 可以访问 使用\_成员名访问

类外/对象外 不可以访问 无（理论上也可以使用\_成员名访问）

子类中 可以访问 使用\_成员名访问

**公开的，公共的封装**

公共的封装实际上对成员没有任何的操作，可以在类中，类外，及子类中都可以访问

封装方法： 不做任何操作

例如： sex 公开的 还是sex

访问限制及方式：

类/对象中 可以访问 使用原有名称

类外/对象外 可以访问 使用原有名称

子类中 可以访问 使用原有名称

----

**继承**

继承就是一个类可以获得另外一个类中的成员属性和成员方法（正常情况下，非私有）

作用：继承的作用主要是减少代码量，增加代码的复用功能，同时可以设置类与类之间的关系

继承与被继承的概念：

用于被继承的类，称之为父类也叫做基类（超类）

用于继承的类，称之为子类，也叫做派生类

**继承的语法**

class 父类：

pass

class 子类(父类):

pass

**继承的特征**

1.所有的类都继承自object类，所有类都是object类的子类。

2.子类继承父类即可以使用父类中所有的成员（私有成员除外）

3.子类继承父类后并没有将父类的成员完全赋值到子类当中，通过引用关系访问调用父类的方法

4.子类中可以定义子类中独有的成员属性和方法

5.如果子类中定义的成员与父类成员名称相同，则会优先使用子类的成员

6.子类如果想扩充父类方法，可以在定义新方法的同时在此访问父类成员来进行代码重用，可以使用 【父类名.父类成员】 的格式来调用父类的成员，也可以使用【super().父类成员】的格式来调用（super()需要详细介绍）

注意:

父类名.父类成员，适用于类访问成员的操作

super().父类成员，适用于对象访问成员的操作

**单继承与多继承**

单继承是指，每个类只继承一个类的方式

多继承是指，每个类可以继承多个类的方式

**单继承和多继承的优缺点**

单继承：(推荐)

优点:继承逻辑清晰，语法简单，并且隐患少

缺点：功能不能无限扩展，只能在当前唯一的继承链中扩展

多继承：

优点：类的功能扩展非常方便

缺点:随着多继承的增加，继承关系会非常混乱，程序隐患很多！

**多继承格式：**

class 类1:

pass

class 类2:

pass

...

子类继承多个父类

class 类(类1,类2..) 该语句为多继承语句

pass

**菱形继承/钻石继承**

多个子类继承同一个父类，这些子类又被同一个类继承，这种多继承关系就是菱形继承

class A：

pass

class B(A):

pass

class C(A):

pass

class D(B,C):

pass

**关于多继承的MRO**

MRO是多继承中，用于保存继承顺序的一个列表。

python本身采用了C3算法来对多继承中的菱形继承进行计算的结果。

MRO列表的计算原则：

1.子类永远在父类的前面

2.如果存在多个父类，则根据继承语法中（）内类的顺序摆放

3.如果多个类继承了同一个父类，孙子类里面只会选取继承语法()中第一个父类的父类

**super详解**

super不是一个关键字也不是一个函数，而是一个类。

super的作用是获取MRO列表中的上一个类而不是父类

super和父类之间没有任何实质性的关系，虽然使用super可能调用到父类

python3使用格式：

super().方法名()

**Mixin设计模式**

Mixin的设计模式主要采用多继承方式对类的功能进行扩展。

优点：

1.使用Mixin模式可以在不对类进行任何修改的情况下，扩充功能

2.可以方便的组织和维护不同功能组件的划分

3.可以根据需要任意调整功能类的组合。

4.可以避免创建很多新的类，导致类的继承混乱

实现方式1(使用多继承方式实现)：

**水果类**

class Fruit:

pass

**礼物相关类**

**礼物类**

class Gift:

pass

非礼物类

class NotGift:

pass

**地区相关类**

**南方类**

class South:

pass

**北方类**

**class North:**

pass

**真实水果类**

**苹果类**

class Apple(Fruit,Gift,North):

pass

**梨类**

class Pear(Fruit,NotGift,North):

pass

**香蕉类**

class Banana(Fruit,Gift,South):

pass

**榴莲**

class Liulian(Fruit,NotGift,South):

pass

**类相关函数**

issubclass() 检测一个类是否是另外一个类的子类

格式：issubclass（子类，需要检测的类的元组）

返回值：布尔值，是其中任意一个类的子类就是True

isinstance() 检测一个对象是否是一个类的实例

格式：isinstance（对象,需要检测的类的元组）

返回值：布尔值，是其中任意一个类的实例就是True

hasattr() 检测某个成员是否在对象中

格式：hasattr（对象,成员名）

返回值：布尔值

getattr() 获取某个对象的成员值

格式:getattr(对象,成员名[,默认值])

返回值：成员的值

注意：1.getattr() 相当于使用对象.成员名访问

2.getattr（）可以设置不存在的成员的默认值

setattr() 设置或者添加对象成员值

格式：setattr（对象，成员名,值）

返回值：无

注意：如果成员存在就是修改成员值，如果成员不存在就是添加成员并赋值

相当于 对象.成员名 = 值

delattr() 删除对象的成员

格式：delattr（对象,成员名）

返回值：无

注意：1.相当于 del 对象.成员名

2.删除时只能删除当前对象所具有的成员，不能删除其访问的类对象成员

dir() 获取对象的成员列表

格式:dir(对象)

返回值:成员组成的列表

property()设置对象成员描述符的操作

格式：property(获取方法，设置方法，删除方法，文档)

**成员描述符**

python描述符是为了在类中对类中的成员属性进行相关操作而创建的一种方式，主要作用就是管理成员属性或者对成员属性进行验证等操作。

描述符的相关操作：

get 获取属性操作

set 修改或者添加属性操作

delete 删除属性操作

**使用类实现描述器**

class Decriptor:

初始化方法将描述其控制的成语属性添加到类中

def \_\_init\_\_(self): self 描述符对象

self.\_name = ''

获取描述其控制成员属性的获取操作

def \_\_get\_\_(self,obj,cls): self 描述符对象，obj 管理成员属性的对象,cls 管理成员属性的类

验证或者修改等操作

return self.\_name 此处修改获取值的格式

设置描述器控制的成员属性的设置操作

def \_\_set\_\_(self,obj,value): self 描述符对象，obj 管理成员属性的对象,value设置的值

此处可以多设置的值进行验证限制等操作

self.\_name = value 赋值操作

设置描述器控制的成员属性的删除操作

def \_\_delete\_\_(self,obj)： self 描述符对象，obj 管理成员属性的对象

此处可以在删除值之前进行验证等操作

del self.\_name

描述其操作的类

class Email：

需要使用描述符控制器的成员属性

name = Decriptor()

**使用property函数实现描述符**

email类

class Email:

邮箱用户

name = ''

-----------描述符设置--------------

获取name属性的描述符方法

def fget(self):

return self.\_name[:10]

设置name属性的描述符方法

def fset(self,value):

判断是否是小写不是小写都转为小写

if value.islower():

self.\_name = value

else:

self.\_name = value.lower()

删除name属性的描述符方法

def fdel(self):

del self.\_name

**使用属性修饰符方法**

邮箱类

class Email:

init方法初始化成员

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

\_name是描述符操作真实name属性的对应属性名 \_name可以是任意名称别用属性本身

self.\_name = ''

用户名属性 进行控制 name

@property 默认获取操作设置

def name(self):

return self.\_name

@name.setter 负责设置操作

def name(self,value):

self.\_name = value

@name.deleter 负责删除操作

def name(self):

del self.\_name

在类中直接设置描述符

name = property(fget,fset,fdel)

---------------------------------

注意：无论哪种修饰符都是为了对成员属性进行访问相关控制。

类的方式 :适合多个类中的多个属性共用一个描述符的方式

propert方式：适合当前类中使用，可以控制一个类中多个属性

属性修饰符：适合当前类中使用,控制一个类中一个属性

对成员属性的操作：获取值，设置/添加值，删除值

**类的内置属性**

\_\_dict\_\_ 获取类的成员组成，以字典方式获取

\_\_doc\_\_ 获取类的文档信息 和函数文档信息一样使用三引号注释在类的开始位置定义

\_\_name\_\_ 获取类的名称，如果在模块中使用，获取模块的名称

\_\_bases\_\_ 获取某个类的所有父类组成的元组

**类的常用魔术方法**

魔术方法就是不需要认为调用,在特定时刻触发的方法.

1. \\_\\_init\\_\\_ () 初始化魔术方法 ★★★★★

触发时机：在对象初始化时触发

参数：至少一个self，可以根据调用类时传入的参数增加

返回值：不需要

作用：初始化对象使用。

2.\\_\\_new\\_\\_ 实例化对象方法 ☆

触发时机：创建对象时触发

参数:至少一个cls，可以根据调用类时传入的参数增加

返回值：必须返回一个对象，否则调用类就白干了

作用：在实例化对象时，进行对象处理，或者阻止创建对象

注意：

1.\_\_new\_\_ 运行在\_\_init\_\_方法之前

2.这个魔术方法一般别用，别动，乱了对象就没了

3.\\_\\_del\\_\\_ 析构方法 ★★★★★

触发时机：在对象没有被任何变量引用的情况下，垃圾回收机制回收对象时触发

参数：一个self 传入对象

返回值:无

作用：在使用完对象时，对对象所调用或者产生的内容进行清除

注意：

del删除对象并不一定触发 \_\_del\_\_,在对象没人引用,被系统回收时才会触发

4.\\_\\_call\\_\\_对象当函数使用触发

触发时机：当对象被当作函数使用时触发

参数：至少一个self参数接收对象，其他根据情况随便加

返回值：可以有可以没有

作用：作为快捷方式使用

注意事项：把对象当作函数调用，不是吧类当作函数调用

5.\\_\\_len\\_\_（）

触发时机: len(对象)的时候触发

参数：一个self 接收对象

返回值: 必须为整型

作用：获取对象内其他关于长度的信息，或者当作部分快捷方式使用（适合返回值为int 或者没有返回值的快捷方式）

6.\\_\\_str\\_\\_（）

触发时机：print（对象）的时候触发

参数：一个self 接收对象

返回值:必须为str类型

作用：打印时提示信息 或者跟返回字符串相关快捷操作

7.\\_\\_repr\\_\\_（）

str() 返回普通字符串，主要用于观察和使用

repr() 返回可以被eval() 执行的字符串

eval() 将一个字符串当作代码来执行

触发时机：str（对象）或者repr(对象)的时候触发

参数：一个self 接收对象

返回值:必须为str类型

作用：打印时提示信息 或者跟返回字符串相关快捷操作

注意：通常 \_\_repr\_\_和\_\_str\_\_定义一个即可，同时使用\_\_str\_\_ = \_\_repr\_\_ 设置两个魔术方法操作一致

8.\\_\\_bool\\_\\_()

触发时机：bool(对象)的时候触发

参数：一个self 接收对象

返回值：布尔值

作用：为对象的设置真假

9.\\_\\_format\\_\\_()

触发时机:字符串.format(对象)的时候触发

参数：一个self 接收对象 第二个参数格式化字符串{:标志}中的标志

返回值：必须为字符串

作用:把对象传入format可以转化为字符串输出使用

**属性相关的魔术方法**

**描述符修相关魔术方法**

1.\\_\\_set\\_\\_()

触发时机：在描述符中对属性进行设置值的时候触发

参数:self 描述符对象 obj 设置属性的对象 value 设置的值

返回值：无

作用：为具有描述符类的属性设置时进行验证或者修改等操作

2.\\_\\_get\\_\\_()

触发时机：在描述符中对属性进行获取时触发

参数：self 当前描述符对象 obj要操作的属性的对象 cls 要操作的属性的对象的类

返回值：有 内容根据情况而定

作用：为具有描述符类的属性进行获取值操作进行的修改或者验证

3.\\_\\_delete\\_\()

触发时机：在描述符中对属性进行删除时触发

参数：self 当前描述符对象 obj要操作的属性的对象

返回值：无

作用：为具有描述符类的属性进行删除时，进行验证

**属性操作相关魔术方法**

1.\\_\\_getattr\\_\\_()

触发时机:在访问一个不存在的成员属性时触发

参数:一个self 获取当前对象 一个参数获取属性名称字符串

返回值:有返回值 根据情况而定

作用:防止访问不存在的属性报错,或者为不存在的属性设置默认值

2.\\_\\_setattr\\_\\_()

触发时机:在对成员属性进行设置时触发

参数:一个self获取当前对象,一个获取设置属性名成字符串,一个是设置的值

返回值:无

作用:在进行属性设置时进验证或者修改

注意:在该方法中不可以使用简单的=赋值操作，否则会进入无限递归操作,需要借助object的设置属性的方法实现

借助基准object对象的设置功能代为设置。 自己写赋值操作会无限循环

object.\_\_setattr\_\_(self,name,value)

3.\\_\\_delattr\\_\\_()

触发时机：对成员属性进行删除操作时

参数：一个self接收当前对象 一个参数接收删除的属性名称字符串

返回值：无

作用：可以在删除前做验证等相关操作

注意:在该方法中不可以使用简单的删除操作，否则会进入无限递归操作,需要借助object的设置属性的方法实现

借助基准object对象的设置功能代为删除。 自己写删除操作会无限循环

object.\_\_delattr\_\_(self,name)

4.\\_\\_getattribute\\_\\_()

触发时机：访问任何一个成员属性时触发，无论成员是否存在

参数：一个self 接收对象 一个参数接收获取成员属性的名称字符串

返回值：最好有，否则啥也获取不到None

作用：获取属性之前作为验证或者其他操作

注意：getattribute 和getattr魔术方法之间存在先后关系

先执行getattribute，然后就没有getattr什么事了

一般情况下这两个魔术方法不会同时存在！

5.\\_\\_dir\\_\\_()

触发时机：使用dir(对象实例)时触发

参数：self 接收当前对象

返回值：列表类型

作用：为当前对象显示成员属性，即使如此用于依然可以通过\_\_dict\_\_来访问所有成员

**运算相关魔术方法(上帝模式)**

**比较运算相关魔术方法**

1.\\_ \_lt\\_ \_()

格式：

def \_\_lt\_\_(self,other):

return 数据

特征：

触发时机：进行小于判断时自动触发

参数：2个参数第一个是self，第二个判断的第二个对象

返回值：返回值可以任意类型，推荐布尔值

作用：定义小于号的行为：x < y 调用 x.lt(y)

2.\\_ \_le\\_ \_()

格式：

def \_\_le\_\_(self):

return str

特征：

触发时机：进行小于等于判断时自动触发

参数：2个参数第一个是self，第二个判断的第二个对象

返回值：返回值可以任意类型，推荐布尔值

作用：定义小于等于号的行为：x <= y 调用 x.le(y)

3.\\_ \_gt\\_ \_()

格式：

def \_\_gt\_\_(self):

return str

特征：

触发时机：进行大于判断时自动触发

参数：2个参数第一个是self，第二个判断的第二个对象

返回值：返回值可以任意类型，推荐布尔值

作用：定义大于号的行为：x > y 调用 x.gt(y)

4.\\_ \_ge\\_ \_()

格式：

def \_\_ge\_\_(self):

return str

特征：

触发时机：进行大于等于判断时自动触发

参数：2个参数第一个是self，第二个判断的第二个对象

返回值：返回值可以任意类型，推荐布尔值

作用：定义大于等于号的行为：x >= y 调用 x.ge(y)

5.\\_ \_eq\\_ \_()

格式：

def \_\_eq\_\_(self):

return str

特征：

触发时机：进行等于判断时自动触发

参数：2个参数第一个是self，第二个判断的第二个对象

返回值：返回值可以任意类型，推荐布尔值

作用：定义大于等于号的行为：x == y 调用 x.eq(y)

6.\\_ \_ne\\_ \_()

格式：

def \_\_ne\_\_(self):

return str

特征：

触发时机：进行不等于判断时自动触发

参数：2个参数第一个是self，第二个判断的第二个对象

返回值：返回值可以任意类型，推荐布尔值

作用：定义不等号的行为：x != y 调用 x.ne(y)

**算术运算相关魔术方法**

\_\_add\_\_(self, other) 定义加法的行为：+

\_\_sub\_\_(self, other) 定义减法的行为：-

\_\_mul\_\_(self, other) 定义乘法的行为：

\_\_truediv\_\_(self, other) 定义真除法的行为：/

\_\_floordiv\_\_(self, other) 定义整数除法的行为：//

\_\_mod\_\_(self, other) 定义取模算法的行为：%

\_\_divmod\_\_(self, other) 定义当被 divmod() 调用时的行为

\_\_pow\_\_(self, other[, modulo]) 定义当被 power() 调用或 \*\* 运算时的行为

\_\_lshift\_\_(self, other) 定义按位左移位的行为：<<

\_\_rshift\_\_(self, other) 定义按位右移位的行为：>>

\_\_and\_\_(self, other) 定义按位与操作的行为：&

\_\_xor\_\_(self, other) 定义按位异或操作的行为：^

\_\_or\_\_(self, other) 定义按位或操作的行为：|

**反运算相关魔术方法**

radd(self, other) （与上方相同，当左操作数不支持相应的操作时被调用）

rsub(self, other) （与上方相同，当左操作数不支持相应的操作时被调用）

rmul(self, other) （与上方相同，当左操作数不支持相应的操作时被调用）

rtruediv(self, other) （与上方相同，当左操作数不支持相应的操作时被调用）

rfloordiv(self, other) （与上方相同，当左操作数不支持相应的操作时被调用）

rmod(self, other) （与上方相同，当左操作数不支持相应的操作时被调用）

rdivmod(self, other) （与上方相同，当左操作数不支持相应的操作时被调用）

rpow(self, other) （与上方相同，当左操作数不支持相应的操作时被调用）

rlshift(self, other) （与上方相同，当左操作数不支持相应的操作时被调用）

rrshift(self, other) （与上方相同，当左操作数不支持相应的操作时被调用）

rand(self, other) （与上方相同，当左操作数不支持相应的操作时被调用）

rxor(self, other) （与上方相同，当左操作数不支持相应的操作时被调用）

ror(self, other) （与上方相同，当左操作数不支持相应的操作时被调用）

**赋值运算相关魔术方法**

iadd(self, other) 定义赋值加法的行为：+=

isub(self, other) 定义赋值减法的行为：-=

imul(self, other) 定义赋值乘法的行为：=

itruediv(self, other) 定义赋值真除法的行为：/=

ifloordiv(self, other) 定义赋值整数除法的行为：//=

imod(self, other) 定义赋值取模算法的行为：%=

ipow(self, other[, modulo]) 定义赋值幂运算的行为：\*\*=

ilshift(self, other) 定义赋值按位左移位的行为：<<=

irshift(self, other) 定义赋值按位右移位的行为：>>=

iand(self, other) 定义赋值按位与操作的行为：&=

ixor(self, other) 定义赋值按位异或操作的行为：^=

ior(self, other) 定义赋值按位或操作的行为：|=

**一元运算相关魔术方法**

pos(self) 定义正号的行为：+x

neg(self) 定义负号的行为：-x

abs(self) 定义当被 abs() 调用时的行为

invert(self) 定义按位求反的行为：~x

**类型转换相关魔术方法 JIANG**

\_\_complex\_\_(self) 定义当被 complex() 调用时的行为（需要返回恰当的值）

\_\_int\_\_(self) 定义当被 int() 调用时的行为（需要返回恰当的值）

\_\_float\_\_(self) 定义当被 float() 调用时的行为（需要返回恰当的值）

\_\_round\_\_(self[, n]) 定义当被 round() 调用时的行为（需要返回恰当的值）

\_\_index(self)\_\_

1. 当对象是被应用在切片表达式中时，实现整形强制转换

2. 如果你定义了一个可能在切片时用到的定制的数值型,你应该定义 index

3. 如果 index 被定义，则 int 也需要被定义，且返回相同的值

**上下文管理相关魔术方法**

1.\\_ \_enter\\_ \_()

2.\\_ \_exit\\_ \_()

enter(self)

1. 定义当使用 with 语句时的初始化行为

2. enter 的返回值被 with 语句的目标或者 as 后的名字绑定

exit(self, exctype, excvalue, traceback)

1. 定义当一个代码块被执行或者终止后上下文管理器应该做什么

2. 一般被用来处理异常，清除工作或者做一些代码块执行完毕之后的日常工作

**容器类型相关魔术方法 JIANG**

\_\_len\_\_(self) 定义当被 len() 调用时的行为（返回容器中元素的个数）

\_\_getitem\_\_(self, key) 定义获取容器中指定元素的行为，相当于 self[key]

\_\_setitem\_\_(self, key, value) 定义设置容器中指定元素的行为，相当于 self[key] = value

\_\_delitem\_\_(self, key) 定义删除容器中指定元素的行为，相当于 del self[key]

\_\_iter\_\_(self) 定义当迭代容器中的元素的行为

\_\_reversed\_\_(self) 定义当被 reversed() 调用时的行为

\_\_contains\_\_(self, item) 定义当使用成员测试运算符（in 或 not in）时的行为

**装饰器**

装饰器就是一种对函数或方法或者类的功能进行扩展的一种格式。

注意：

修饰符适用于对成员属性的增删改查操作进行研制和验证的另外一个东

**装饰器第四步：实现基本的装饰器**

def decor(func):

定义装饰器函数的内部函数

def \_decor():

print('烧香拜佛，祝一切顺利')

func() 调用原有函数

print('烧香拜佛，还愿')

return \_decor

要被扩展功能的函数

@decor 相当于laxi = decor(laxi)

def laxi():

print('噗哧噗哧噗哧噗哧～～')

laxi()

laxi()

'''

**装饰器第五步：带有参数和返回值的装饰器**

装饰器

def decor(func):

这就是未来的拉稀函数

def \_decor(who,weight):

扩展功能

print('烧香拜佛，祝一切顺利')

获取原函数执行之后的返回值

result = func(who,weight) 调用原有函数

print('烧香拜佛，还愿')

返回原有函数值

return result

将扩展之后的函数赋值给原有函数

return \_decor

@decor

def laxi(who,weight):

print('噗哧噗哧噗哧噗哧～～:'+who+'拉了'+weight+'斤屎')

返回值

return '一碗屎'

result = laxi('罗霖','20')

print(result)

result = laxi('罗霖霖','10')

print(result)

'''

**装饰器第六步:收集参数装饰器 可以让装饰器装饰更多类型的函数**

装饰器 可以装饰N多的函数

def decor(func):

def \_decor(\*arg,\*\*args):

扩展功能

print('烧香拜佛，祝一切顺利')

result = func(\*arg,\*\*args) 调用原有函数

print('烧香拜佛，还愿')

return result

return \_decor

@decor

def laxi(\*arg,\*\*args):

print('噗哧噗哧噗哧噗哧～～:')

print('参与拉屎的人有',arg)

print('拉屎重量分别为',args)

返回值

return '一碗屎'

result = laxi('冯伟','冯小肛','冯自信','冯巩',fw = 10,fxg = 15 ,fzx = 20,fg = 25)

print(result)

result = laxi('曹星',cx = 1000)

print(result)

@decor

def chifan(who1,who2):

print(who1+'和'+who2+'正在吃饭')

return '吃饱了'

result = chifan('朱升','杨磊')

print(result)

**装饰器第七步:带有参数的装饰器**

def decor(arg): 此处参数接收装饰器传参

def \_decor(func): 此处参数接收函数本身

根据装饰本身的参数决定返回不同的装饰之后的函数

定义扩展之后的哦未来的拉稀函数

def \_\_decorla(): 此处参数接收函数传参

print('脱裤子')

result = func()

print('提裤子')

return result

定义扩展之后的哦未来的拉稀函数

def \_\_decorchi(): 此处参数接收函数传参

print('饭前洗手')

result = func()

print('饭后洗手')

return result

将装饰装饰之后的函数返回

if arg == '拉':

return \_\_decorla

else:

return \_\_decorchi

返回装饰器的第二层函数

return \_decor

拉稀函数

@decor('拉')

def laxi():

print('扑哧扑哧～～～～')

return '热翔一碗'

调用装饰之后的laxi函数

res = laxi()

print(res)

吃饭函数

@decor('吃')

def chifan():

print('吃了这碗咖喱土豆泥！～')

return '呃～'

调用装饰之后的吃饭函数

res = chifan()

print(res)

**装饰器第八步：使用类作为装饰器参数**

用于装饰器参数中使用的类

class Wish:

def before():

print('祝一切顺利')

def after():

print('感谢真主～')

装饰器

def decor(cls): 接收装饰器参数

def \_decor(func): 接收函数本身

def \_\_decor(): 接收函数参数

调用类的某个属性方法

cls.before()

result = func()

调用类的某个属性方法

cls.after()

return result

返回加工之后的函数

return \_\_decor

返回装饰部分

return \_decor

@decor(Wish)

def laxi():

print('扑哧扑哧～～～')

return '一碗热翔'

调用函数

res = laxi()

print(res)

**装饰器第九步:把类当作装饰器使用(少用)**

设置类装饰器

class Decor:

初始化对象

def \_\_init\_\_(self,num): 接收装饰器的参数

将类装饰器的参数保存起来

self.num = num

def \_\_call\_\_(self,func): 接收函数本身

将被装饰的函数保存起来

self.func = func

调用新的扩展方法

return self.newLaxi

扩展拉稀之后的新的方法

def newLaxi(self,who):

使用装饰器的参数

i = 0

while i < self.num:

print('便前洗手')

i+=1

result = self.func(who)

print('便后洗手')

return result

@Decor(5) laxi = Decor(laxi)

def laxi(who):

print('扑哧，扑哧～～～')

return '金粒餐一枚'

res = laxi('王子明')

print(res)

'''

'''

**装饰器第十步:为类添加装饰器(单例设计模式) 本质上在装饰\_\_new\_\_**

全局变量

创建一个空字典

objdict = {}

装饰器

def decor(cls): 接收类 相当于 decor

获取类的唯一实例

def \_decor(name,sex,age): 接收类的参数 相当于\_decor

检测当前类是否已经创建过对象实例

if 'only' not in objdict: 没有创建对象实例时候

objdict['only'] = cls(name,sex,age)

return objdict['only']

return \_decor

韩梅梅的类(使用装饰器保证该类只能有一个类的实例对象)

@decor

class HanMeiMei:

name = ''

sex = ''

age = 0

def \_\_init\_\_(self,name,sex,age):

self.name = name

self.sex = sex

self.age = age

def studyEn():

print('我的英语可棒了～')

one = HanMeiMei('韩梅梅','女',18)

print(one.\_\_dict\_\_)

two = HanMeiMei('韩美美','女',28)

print(two.\_\_dict\_\_)

**装饰器第十一步:多装饰器嵌套**

装饰器1

def decor1(func):

def \_decor1():

print('便前洗手')

result = func()

print('便后洗手')

return result

return \_decor1

装饰器2

def decor2(func):

def \_decor2():

print('便前漱口')

result = func()

print('便后刷牙')

return result

return \_decor2

定义被装饰的函数

装饰器2

@decor2

装饰器1

@decor1 装饰过程 laxi = decor2(decor1(laxi))

def laxi():

print('扑哧～～～')

return '羊屎豆'

res = laxi()

print(res)

**类和对象的三种方法**

类和对象有三种方法：

**实例方法** （对象使用的方法）

需要实例化对象才可以使用的方法，在使用过程中可能借助对象的其他成员来完成

**静态方法** （绑定类的方法） 少见多怪！

不需要实例化，可以通过类直接访问的方法，在使用过程中可能借助类的相关成员来完成

**类方法** (类使用的方法)

不需要实例化，可以通过类直接访问的方法，这种方法通常独立存在，不需要借助类或者对象的其他成员来完成

以下为实例：

class Person:

实例方法

def eat(self):

print(self)

print('吃方法～')

类方法

@classmethod

def play(cls):

print(cls)

print('玩方法')

静态方法/绑定类的方法

@staticmethod

def say():

print('说话方法')

实例化对象

yuxiang = Person()

访问实例方法

yuxiang.eat()

访问类方法

Person.play()

访问静态方法

Person.say()

抽象基类 ABSTRACTCLASS

抽象类的使用需要借助于abc模块

import abc

抽象方法：没有具体方法内容的方法就是抽象方法

抽象类：包含抽象方法的类就是抽象类，通常称为ABC类

**实现真正的抽象类**

导入抽象类模块

import abc

声明一个类并且指定当前类的元类为抽象类提供的元类

class Human(metaclass = abc.ABCMeta):

定义一个普通的抽象方法(实例的抽象方法)

@abc.abstractmethod

def smoking(self):

pass

定义一个类抽象方法（绑定类的抽象方法）

@abc.abstractclassmethod

def drink():

pass

定义一个抽象静态方法(静态方法的抽象方法)

@abc.abstractstaticmethod

def play():

pass

**抽象类的特点：**

1.抽象类可以包含抽象方法也可以包含具体方法

2.抽象类中可以有方法也可以有属性

3.抽象类无法直接实例化对象，因为有抽象方法的存在

4.抽象类必须继承才可以使用，而且必须在实现了所有继承而来的抽象方法之后才可以实例化

5.抽象类的主要作用是设定类的标准，以便于开发时具有统一的规则约束

**多态**

多态就是同一个对象在不同情况下有不同的状态体现，这就是多态

多态不是语法，而是设计思想

**17.1 错误和异常处理**

在python中只有2种错误：一种是语法错误，另一种就是异常

**语法错误**

python中语法错误也叫做解析错误，语法不符合规定导致的错误，初学者常见！

**异常**

异常是指在语法正确的前提下，运行时发生的错误！

在python中，语法正确是无法运行时所报的错误都是异常。

异常是一个对象，在程序的思维中。

所以异常时可以进行处理的！

**异常的分类（标准异常分类）**

AssertError 断言语句（assert）失败

AttributeError 尝试访问未知的对象属性

EOFError 用户输入文件末尾标志EOF（Ctrl+d）

FloatingPointError 浮点计算错误

GeneratorExit generator.close()方法被调用的时候

ImportError 导入模块失败的时候

IndexError 索引超出序列的范围

KeyError 字典中查找一个不存在的关键字

KeyboardInterrupt 用户输入中断键（Ctrl+c）

MemoryError 内存溢出（可通过删除对象释放内存）

NameError 尝试访问一个不存在的变量

NotImplementedError 尚未实现的方法

OSError 操作系统产生的异常（例如打开一个不存在的文件）

OverflowError 数值运算超出最大限制

ReferenceError 弱引用（weak reference）试图访问一个已经被垃圾回收机制回收了的对象

RuntimeError 一般的运行时错误

StopIteration 迭代器没有更多的值

SyntaxError Python的语法错误

IndentationError 缩进错误

TabError Tab和空格混合使用

SystemError Python编译器系统错误

SystemExit Python编译器进程被关闭

TypeError 不同类型间的无效操作

UnboundLocalError 访问一个未初始化的本地变量（NameError的子类）

UnicodeError Unicode相关的错误（ValueError的子类）

UnicodeEncodeError Unicode编码时的错误（UnicodeError的子类）

UnicodeDecodeError Unicode解码时的错误（UnicodeError的子类）

UnicodeTranslateError Unicode转换时的错误（UnicodeError的子类）

ValueError 传入无效的参数

ZeroDivisionError 除数为零

**常见的异常案例：**

IndexError

list1 = ['冯巩','牛群','马三立','侯宝林']

print(list1[30]) 使用了不存在的索引值

KeyError

dict1 = {'niu':'穹','zhu':'升','yang':'磊','ma':'星辉'}

print(dict1['lv']) 使用了不存在的键

nameError

print(diao) 使用了为定义的变量

ZeroDivisionError

print(99/0) 被除数为0

KeyboardInterrupt

while True:

print('大爷来玩啊～')

用于按ctrl+C终止shell执行的错误

AssertionError

assert len([1,2,3,4]) > 1 断言成功，不报错

assert len([1,2,3,4]) > 10 断言失败时被报错

**错误异常处理**

语法格式：

try:

尝试实现某个操作，

如果没出现异常，任务就可以完成

如果出现异常，将异常从当前代码块扔出去尝试解决异常

except 异常类型1:

解决方案1：用于尝试在此处处理异常解决问题

except 异常类型2：

解决方案2：用于尝试在此处处理异常解决问题

except (异常类型1,异常类型2...)

解决方案：针对多个异常使用相同的处理方式

excpet:

解决方案：所有异常的解决方案

else:

如果没有出现任何异常，将会执行此处代码

finally:

管你有没有异常都要执行的代码

**try语法**

1.无任何异常情况下的处理顺序

1.执行try区域内的代码

2.如果没有任何异常发生

3.执行else区域的代码

4.执行finally区域的代码

2.发生异常的处理顺序

1.执行try区域内的代码

2.如果出现异常，将异常从当前try区域抛出

3.第一个【except 异常类型1】，会查询抛出的异常是否和我指定的类型一致

如果异常和的类型一致，则进入当前except区域进行处理

如果异常和类型不一致，则跳过当前except区域将异常送到下一个except匹配

4.第二个【except 异常类型2】，会查询抛出的异常是否和我指定的类型一致

如果异常和的类型一致，则进入当前except区域进行处理

如果异常和类型不一致，则跳过当前except区域将异常送到下一个except匹配

...

5.如果except区域执行完毕或者没有任何except区域能够接受异常对象，继续运行finally区域

**用户自定义异常**

如果要用户自己操作异常对象需要使用到raise语句，抛出系统的内置异常类型或者自定义的异常类型

raise语法

raise 错误异常对象

自定义异常类型

class 异常类名(RuntimeError):

pass

try:

raise 自定义异常类() 生成并抛出异常类对象

except 异常类 as 接收对象变量:

可以使用接收对象的变量

---

自定义异常类中成员可以根据需要添加或者不添加

常用的自定义类中的异常属性

自定义发生异常代码

自定义发生异常文字提示

自定义发生异常的行数

等

**with语法**

try:

with open('文件','模式') as 变量： 变量接收了open的返回值

使用变量

except:

错误异常处理区域

pass

使用该语法操作文件时不需要关闭文件，with会监控文件的使用情况，在适当的时机自动关闭

**17.2 模块**

一个模块就是一个包含python代码的文件，后缀名是.py就行，模块就是个python文件

**为什么要使用模块**

1. 因为现代程序非常大，在一个文件中书写和维护非常不方便，所以我们拆分成多个文件方便维护与管理

2. 模块可以增加代码重复利用的方式

3. 当作命名空间使用，避免命名冲突。

**如何自定义模块**

模块就是一个普通文件，所以任何代码都可以在模块中直接书写，不过根据模块的规范，最好在模块中书写以下内容：

函数(单一功能)

类(相似功能的组合)

测试代码(仅供测试，作为模块被导入时不执行)

\*\*注意：\*\* 如何使测试代码在单独运行时执行，而作为模块被导入时又不被执行

需要使用\_\_name\_\_的特殊变量：

在当前运行的文件/模块当中 \_\_name\_\_的值为 \_\_main\_\_

在当前文本/模块被导入时 \_\_name\_\_的值为 文件/模块名称

**如何使用模块**

import 模块

直接导入模块使用

使用方式:

模块.函数名

模块.类名.函数名

模块.类名.属性名

import 模块 as 别名

导入模块时为模块添加别名

使用方式:

别名.函数名

别名.类名.函数名

别名.类名.属性名

from 模块 import 某个函数

导入模块中的某一个函数

使用方式：

函数名

from 模块 import 函数1,函数2,类...

导入模块中的部分函数

使用方式：

函数名()

类名.方法名()

类名.属性名()

from 模块 import \*

导入模块中的所有内容

使用方式：

函数名（）

类名.方法名()

类名.属性名()

**\*\*注意\*\*：**使用from..import 的方式导入模块中的内容，可能导致出现问题：多个模块中可能存在相同的函数名，会导致冲突。

一般情况下仅在确定当前程序只使用某个模块的内容的时候才会使用该方式导入！

**模块的搜索路径与存放**

模块的搜索路径：

表示加载模块是，python程序会在哪些文件夹中查找文件

查询系统默认的模块搜索路径：

import sys

sys.path属性 可以获取路径列表

添加搜索路径

sys.path.append(新的路径)

\*\*注意：\*\*如果不想添加新的搜索路径，建议将用户的模块存放在C:\Users\xdl\AppData\Local\Programs\Python\Python36-32\Lib\site-packages

**模块的加载顺序**

1.搜索内存中已经加载的模块

2.搜索python的内置模块

3.搜索sys.path路径中的模块

**包**

包也是一种组织管理代码的一种方式。包中存放的是模块。

用于将模块包含在一起的文件夹就是包。

自定义包的结构

|---包

|---|--- \_\_init\_\_.py 包的标志文件

|---|--- 模块1

|---|--- 模块2

|---|--- 子包(子文件夹)

|---|---|--- \_\_init\_\_.py 包的标志文件

|---|---|--- 子包模块1

|---|---|--- 子包模块2

....

**包的导入操作**

**import 导入**

import 包

直接导入一个包,可以使用\_\_init\_\_中的内容

使用格式：

包.函数名

包.类名.方法名()

包.类名.属性名

注意：该方式访问的内容是\_\_init\_\_中的内容

import 包 as 别名

直接导入一个包,可以使用\_\_init\_\_中的内容

使用格式：

包别名.函数名

包别名.类名.方法名

包别名.类名.属性名

注意：该方式访问的内容是\_\_init\_\_中的内容

import 包.模块

导入包中指定的一个模块，这里包.模块相当于 路径格式

使用格式：

包.模块.函数名

包.模块.类名.方法()

包.模块.类名.属性

import 包.模块 as 别名

导入一个包中的模块，并且添加别名（模块的别名）

使用方式:

别名.函数名

别名.类名.函数名

别名.类名.属性名

**form ..import 导入**

from 包 import 模块

导入包中指定的某个模块

使用方法：

模块名.函数名

模块名.类名.方法名（)

模块名.类名.属性名

from 包 import 模块1,模块2..

导入包中指定的某个模块

使用方法：

模块名.函数名

模块名.类名.方法名（)

模块名.类名.属性名

from 包 import \*

导入当前包\_\_init\_\_模块中所有的函数和类

使用方法：

函数名

类名.方法名()

类名.属性

from 包.模块 import \*

导入包中指定模块的所有内容函数和类

使用方法：

函数名

类名.方法名()

类名.属性

**关于导入其他模块在当前模块中使用**

在开发模块时不可避免的需要使用其他模块的内容，我们可以在当前包中直接导入其他模块中的内容

import 完整的包和模块路径

**\_\_all\_\_的用法**

\_\_all\_\_的作用是设置在使用from 包 import \* 的时候 ，\*可以导入的内容

1.\_\_int\_\_.py中 如果空文件 或者没有\_\_all\_\_的时候 只可以把\_\_init\_\_文件中的类和函数导入进来

2.\_\_init\_.py中 如果设置了\_\_all\_\_的值，那么则会按照\_\_all\_\_指定的子包和模块进行加载,如此则不会导入init中的函数

格式：

\_\_all\_\_ = ['模块名'，'模块名'，'包名'...]

**命名空间**

用于区分不同位置不同功能的变量或者函数的特定前缀，这个就是命名空间，他的作用就是为了防止命名冲突。

max() 共用的获取最大值的函数

Human.max() 人类中的max()方法 使用类名作为区分命名使用

Dog.max() 狗类中的max()方法 使用类名作为区分命名使用、

**真正的命名空间**

兄弟连开发的MyUser类

namespace com/xdl/python

class MyUser(){

类代码

}

实例化类

new /com/xdl/pythonMyUser()

百度开发的MyUser类

namespace com/baidu/www

class MyUser(){

类代码

}

new /com/baidu/www/MyUser()