第九章 面向对象

本节所讲内容：

9.1 面向对象定义

9.2 类的编写

9.3 实例方法和self

9.4 类属性和实例属性

9.1 面向对象的定义

9.1.1 面向对象介绍：

在了解面向对象之前，我们首先来了解一下什么是编程，所谓编程就是程序员用和计算机约定好的语言向计算机描述我们的需求。所以想要学好编程要从两个方向着手：

了解我们要编写的事物的规律，在工作当中，我们称之为业务，比如，一个建筑公司的程序员就需要了解 建筑方向的一下基本的业务逻辑，比如概算流程、预算流程、项目审核、质检安检等，在这里也希望各位同学在工作当中，了解代码的同时，也要了解公司的业务，这样才可以有更好的一个发展。

就是一个好的描述方式，就好像我们在小学的时候写作文，有记叙文有议论文，对于程序员来说 ，所谓的记叙文或者议论文就是我们编程的思路，通常有：面向对象，面向过程。我们今天学习的就是面向对象。

那我们来向思考下，什么是面向对象？面向对象无非就是一种编程思路的表达方式而已。

那么接下来，我们就来具体的认识一下面向对象编程。

在了解面向对象编程的时候，我们必须先了解一下概念

对象：万物皆对象，就是事实存在的事物

类： 是对事物的划分，当我们在描述的时候用到类，就是在描述一系列的事物的共性特征， 比如：鸟类，我们想要说的是所有鸟的共性比如：卵生，羽毛

实例： 从类当中具体映射出来的个体，比如鸟类当中的鸵鸟

在这里我们要注意，类和实例都有独立的内存空间，相互独立，互不影响，实例来源于类但是要比类更加具有个性，比如：鸵鸟属于鸟类，但是它的体重就是鸟类当中的一个个性

域： 属于类或者实例的变量，一些名词，例如name，age……

方法：属于类或者实例的功能(函数)，一般来说是函数。一些动词，吃，喝……

属性: 域和方法的统称

类方法：属于类的功能

类属性：属于类的属性

实例方法：属于实例的功能

实例属性：属于实例的属性

那么到这里，我们来理解一下，什么是面向对象：在编程过程当中，把数据及对数据的操作方法放在一起，作为一个相互依存的整体，用这样的思路编程就是面向对象。

通俗的来说就是我们在编程的时候将一类事物的参数和功能描述在一个叫做class的东西里面，进行编程的思路。

例如：去跪键盘和我去跪键盘……

那么我们看一个例子：

请用程序描述如下事情：

同学报道登记信息

A同学报到

B 学校记录

面向过程的方式：

school = {

'Linux':[],

'Python':[]

}

def register(major,student):

global school

if major in school:

school[major].append(student)

print('报名成功')

else:

print('抱歉啊，我们学校没有你的%s专业'%major)

if \_\_name\_\_ == '\_\_main\_\_':

student = {

'name':'for',

'age':18,

'gender':'男',

'major':'Python'

}

register(student.get('major'),student)

print(school)

运行结果如下：

报名成功

{'Linux': [], 'Python': [{'age': 18, 'name': for, 'major': 'Python', 'gender': '男'}]}

面向对象的方式：

class Registration:

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

self.school={

'Linux':[],

'Python':[]

}

def register(self,major,student):

if major in self.school:

self.school[major].append(student)

print('报名成功')

else:

print('抱歉我们学校没有你要的%s专业'%major)

if \_\_name\_\_ == '\_\_main\_\_':

student = {

'name':'for',

'age':18,

'gender':'男',

'major':'Python'

}

reg = Registration()

reg.register(student.get('major'),student)

print(reg.school)

运行结果如下

报名成功

{'Linux': [], 'Python': [{'age': 18, 'name': for, 'major': 'Python', 'gender': '男'}]}

两种方法得到的结果都是一样的，但是，考虑现实生活中，报到是到一个叫做报到处的地方办理报到，而不是调用报到的功能。

就好比解决吃啤酒鸭的问题

第一种方式（面向过程）:

1)养鸭子

2)鸭子长成

3)杀

4)作料

5)烹饪

6)吃

第二种方式（面向对象）:

1)找个卖啤酒鸭的人

2)给钱 交易

3)吃

4)胖2斤

所以，面向对象更加贴近我们的思维惯性，毕竟在生活当中我们周围是一个一个的事物对象。



9.2 类的编写

Python编程中类的概念可以比作是某种类型集合的描述，如“人类”可以被看作一个类，然后用人类这个类定义出每个具体的人——你、我、他等作为其对象。

类还拥有属性和功能，属性即类本身的一些特性，如人类有名字、身高和体重等属性，而具体值则会根据每个人的不同而不同；功能则是类所能实现的行为，如人类拥有吃饭、走路和睡觉等功能。

9.2.1 类的定义

定义一个类，格式如下：

class 类名（继承列表）:

'类的文档字符串'

实例属性

实例方法的定义

类变量的定义

类方法的定义(@classmethod)

静态方法的定义(@staticmethod)

类(class) 由以下几个部分构成

关键字 :class

类的名称: 类名

类的属性: 一组数据

类的方法: 允许对进行操作的方法 (行为)

实例方法：允许对象操作的方法（行为）

作用：

类是为了描述对象的行为和属性

类也可以用于创建一个或者多个对象

案例：

定义一个Cat类

#coding = utf-8

class Cat:

print（'我是猫咪'）

强调：

类名的命名规范要符合“大驼峰”规则。

9.2.2 构造对象

通过上面的操作我们创建了一个Cat类，然后我们就好比有了一张猫类的图纸，我们可以通过图纸创建多个对象

1、创建对象的格式：

对象名 = 类名（[传参列表]）

class Cat():

print('我是喵咪')

*#实例化一个tom对象*

tom = Cat()

运行结果如下：

我是喵咪

实例对象预先说明:

1. 实例有自己的作用域和名字空间,可以为该类的对象添加实例变量(也叫属性)

2. 实例可以调用类方法和实例方法

3. 实例可以访问类变量和实例变量

9.3 实例方法和self

9.3.1 实例方法

语法:

class 类名(继承列表):

def 实例方法名(self, 参数1, 参数2, ...):

'文档字符串'

语句

作用:

用于描述一个对象的行为,让此类型的全部对象都拥有相同的行为。

说明:

1. 实例方法实质是函数,是定义在类内的函数

2. 实例方法至少有一个形参,第一个形参代表调用这个方法的实例,一般命名为 'self'

实例方法的调用语法:

实例.实例方法名(调用传参)

或

类名.实例方法名(实例,调用传参)

class Cat():

*'我是喵咪'*

def eat(self):

*'猫'*

print('猫在吃……')

def drink(self):

print('猫在喝水……')

*#创建一个tom实例化对象*

tom = Cat()

*#调用实例方法*

tom.eat()

tom.drink()

*#通过类名调用实例方法*

Cat.eat(tom)

运行结果如下：

猫在吃……

猫在喝水……

猫在吃……

为什么Cat.eat(tom)调用的时候把tom这个实例化对象作为一个参数传递进去？

为什么实例方法中也能够正常的打印？

让我们走进self的世界，探索下self的内心底层的秘密。

9.3.2 self

我们在前面的讲解中知道的类的构成和创建对象，那么有些同学肯定对每个实例中的self很感兴趣，它的背后到底有什么前尘往事，就让我为大家一一道来，首先我们看下这则案例。

案例：

#定义一个cat类

class Cat():

*'我是喵咪'*

def eat(self):

*'猫'*

print('猫在吃……')

def drink(self):

print('猫仔喝水……')

def prinr\_info(self):

print('%s 的年龄是%s'%(tom.name,tom.age))

*#创建一个tom实例化对象*

tom = Cat()

*#调用实例方法*

tom.eat()

tom.drink()

*#通过类名调用实例方法*

Cat.eat(tom)

tom.name = '汤姆'

tom.age = 18

tom.print\_info()

print('^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^')

lucky = Cat()

lucky.eat()

lucky.drink()

lucky.name = '幸运'

lucky.age = 18

lucky.print\_info()

运行结果如下：

猫在吃……

猫仔喝水……

猫在吃……

汤姆 的年龄是18

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

猫在吃……

猫仔喝水……

汤姆 的年龄是18

发现两次print的执行结果是一样的？

分析：

首先定义了一个Cat类，然后我们实例化了tom这个对象，并设置tom.name = '汤姆', tom.age=40,而咱们在Cat类中的print\_info的方法调用的是tom这个对象的name和age属性，所以打印name=汤姆，age=40

接着实例化了lucky这个对象，虽然我们也是设置了name和age属性值，但是我们在print\_info方法中调用的还是tom对象的属性，所以结果没有发生变化。

这个问题很让我们头大，那我们可以想下深层次的知识，是不是利用一个中间变量过渡下呢？

class Cat():

*'我是喵咪'*

def eat(self):

*'猫'*

print('猫在吃……')

def drink(self):

print('猫仔喝水……')

def prinr\_info(self):

print('%s 的年龄是%s'%( self.name, self.age))

*#创建一个tom实例化对象*

tom = Cat()

*#调用实例方法*

tom.eat()

tom.drink()

*#通过类名调用实例方法*

Cat.eat(tom)

tom.name = '汤姆'

tom.age = 18

tom.print\_info()

print('^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^')

lucky = Cat()

lucky.eat()

lucky.drink()

lucky.name = '幸运'

lucky.age = 18

lucky.print\_info()

运行结果如下：

猫在吃……

猫仔喝水……

猫在吃……

汤姆 的年龄是18

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

猫在吃……

猫仔喝水……

幸运的年龄是18

分析：

print\_info实例方法中我们改为self，所有问题都解决。

到这里我们可以知道self 竟然惊人的是每个实例化对象本身，那到底self代表的到底是不是每个实例化对象本身呢？让我们证明下。

class Bird:

def hello(self):

print(self)

#创建一个实例对象

b = Bird()

print(b)

b.hello()

#官方证明

Bird.hello(b)

#创建的另一个实例

c = Bird()

print(c)

c.hello()

运行结果如下：

<\_\_main\_\_.Bird object at 0x0000024BCF432630>

<\_\_main\_\_.Bird object at 0x0000024BCF432630>

<\_\_main\_\_.Bird object at 0x0000024BCF432630>

<\_\_main\_\_.Bird object at 0x0000024BCF4326A0>

<\_\_main\_\_.Bird object at 0x0000024BCF4326A0>

由以上结果得出，我们可以明显得出self 代表的就是实例化对象本身。也就是当某个对象调用其方法时，python解释器会把这个对象作为第一个参数传递给self，所以开发者只需要传递后面的参数即可。

比如：调用的使用b.hello()相当于b.hello(b)；但是我们实际上是不能真正把b传入hello()方法的，Python解析器会自动帮我传递。

通俗的讲，就是哪个对象调用这个实例方法，这个实例方法中的self就是这个对象。

我们也可以把self，理解为对象自身，当然可以把self当做C++和java中类里面的this指针一样理解，就是对象自身的意思。

注意：

我们在编写程序的时候self 不一定是固定的写法，也可以用其他的标识符进行替代。

class Cat():

*'我是喵咪'*

def eat(self):

*'猫'*

print('猫在吃……')

def drink(self):

print('猫仔喝水……')

def print\_info(a):

print('%s 的年龄是%s'%(a.name,a.age))

*#创建一个tom实例化对象*

tom = Cat()

*#调用实例方法*

tom.eat()

tom.drink()

*#通过雷鸣调用实例方法*

Cat.eat(tom)

tom.name = '汤姆'

tom.age = 18

tom.print\_info()

print('^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^')

lucky = Cat()

lucky.eat()

lucky.drink()

lucky.name = '幸运'

lucky.age = 18

lucky.print\_info()

运行结果如下：

猫在吃……

猫仔喝水……

猫在吃……

汤姆 的年龄是18

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

猫在吃……

猫仔喝水……

幸运的年龄是18

分析：

self也可以用其他标识符替代，但在这里self已经是约定俗成一个，它更加符合咱们的世界观。

9.4 类属性和实例属性

9.4.1 类属性

Python 中类的属性就是定义在类的内部并且在实例方法外部的属性。

顾名思义，类属性就是类对象所拥有的属性，但是它可以被所有类的实例对象所共有，在内存中只存在一个副本。这个和C++，java中类的静态成员变量有点类似。接下来让我们通过例子直观的感受到类属性。

class Cat():

age = 18

def eat(self):

*'猫'*

print('猫在吃……')

def drink(self):

print('猫仔喝水……')

tom = Cat()

print(tom.age)

print(Cat.age)

运行结果如下：

18

18

上面的例子来说我们可以直观的看到，我们可以通过类名来调用类中的属性。那如果在类的内部用对象进行访问是否可行呢？我该如何调用呢？

class Cat():

age = 18

def eat(self):

*'猫'*

print('猫在吃……')

def drink(self):

print('猫仔喝水……')

def print\_info(self):

print(self.age)

tom = Cat()

print(tom.age)

print(Cat.age)

tom.print\_info()

运行结果如下：

18

18

18

分析：我们从上面的例子可以得出类属性也可以用实例化对象进行访问，并且还可以在实例方法的内部调用

9.4.2 实例属性

在学习实例属性之前，我们先学习下\_\_init\_\_方法

初始化方法

作用:

对新创建的对象添加属性

语法:

class 类名(继承列表):

def \_\_init\_\_(self [, 形参列表]):

语句块

注明：[形参列表] 代表中的内容可省略

说明:

1、初始化方法名必须为 '\_\_init\_\_' 不可改变

2、初始化方法会在构造函数创建实例后自动调用.且将实例自身通过第一个参数self 传入 \_\_init\_\_ 方法

3、构造函数的实参将通过\_\_init\_\_方法的参数列表传入到 '\_\_init\_\_' 方法中

4、初始化方法内如果需要return 语句返回,则只能返回None

2. 使用方式

方式

class 类名:

#初始化函数，用来完成一些默认的设定

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

pass

案例：

class Cat():

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

print('\_\_init\_\_被调用了')

def eat(self):

print('猫在吃……')

def drink(self):

print('猫仔喝水……')

运行结果如下：

\_\_init\_\_被调用了

3、在\_\_init\_\_()方法中给类设置默认属性

class Cat():

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

self.name = '汤姆'

self.age = 18

def eat(self):

print('猫在吃……')

def drink(self):

print('猫仔喝水……')

def print\_info(self):

print('%s 的年龄是%s'%(self.name,self.age))

*#实例化对对象*

tom = Cat()

tom.print\_info()

运行结果如下：

汤姆 的年龄是18

小结：

当创建Cat对象后，在没有调用\_\_init\_\_()方法的前提下，tom就默认拥有了2个属性name和age，原因是\_\_init\_\_()方法是在创建对象后，就立刻被Python解析器默认调用了

4、使用\_\_init\_\_()传递参数

既然在创建完对象后\_\_init\_\_()方法已经被默认的执行了，那么能否让对象在调用\_\_init\_\_()方法的时候传递一些参数呢？如果可以，那怎样传递呢？

案例：

class Cat():

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

self.name = name

self.age = age

def eat(self):

print('猫在吃……')

def drink(self):

print('猫仔喝水……')

def print\_info(self):

print('%s 的年龄是%s'%(self.name,self.age))

*#实例化对对象*

tom = Cat('汤姆',18)

*# tom.name = '汤姆'*

*# tom.age = 18*

tom.print\_info()

运行结果如下：

汤姆 的年龄是18

小结：

\_\_init\_\_()方法，在创建一个对象时默认被调用，不需要手动调用;

\_\_init\_\_(self)中，默认有1个参数名字为self，如果在创建对象时传递了2个实参，那么\_\_init\_\_(self)中出了self作为第一个形参外还需要2个形参，例如\_\_init\_\_(self,x,y);

\_\_init\_\_(self)中的self参数，不需要开发者传递，python解释器会自动把当前的对象引用传递进去。

9.4.3 扩展知识点

定义一个类分为2种：新式类和经典类，上面class Cat(object)是一个新式类，显式继承object。那么我们想一想经典类的写法是什么呢？它们之间的区别在哪里呢？

*#经典类*

class Cat:

class Cat():

*#新式类*

class Cat(object):

2、 新式类和经典类的区别

在Python 3.x中取消经典类，默认都是新式类，并且不必显式的继承object，也就是说：

*#新式类*

class Cat:

class Cat():

class Cat(object):

三种写法并无区别，推荐第一种。

但是在Python2.x中，默认都是经典类，只有显式继承了object才是新式类，即：

class Cat(object): 新式类写法

class Cat():经典类写法

class Cat: 经典类写法

新式类都从object继承，经典类不需要。

注意：新式类继承父类的时候只会执行一次构造函数（\_\_init\_\_），经典类重复执行多次。

总结：

9.1 面向对象定义

9.2 类的编写

9.3 实例方法和self

9.4 类属性和实例属性