# 1. 面向对象入门

## 1.1. 面向过程和面向对象的区别

### 1.1.1. 面向过程：根据业务逻辑从上到下写代码

面向过程编程最易被初学者接受，其往往用一长段代码来实现指定功能，开发过程的思路是将数据与函数按照执行的逻辑顺序组织在一起，数据与函数分开考虑。

while True：

    if cpu利用率 > 90%:

        #发送邮件提醒

        连接邮箱服务器

        发送邮件

        关闭连接

    if 硬盘使用空间 > 90%:

        #发送邮件提醒

        连接邮箱服务器

        发送邮件

        关闭连接

    if 内存占用 > 80%:

        #发送邮件提醒

        连接邮箱服务器

        发送邮件

        关闭连接

随着时间的推移，开始使用了函数式编程，增强代码的重用性和可读性，就变成了这样.

def 发送邮件(内容)

#发送邮件提醒

连接邮箱服务器

发送邮件

关闭连接

def main():

while True：

if cpu利用率 > 90%:

发送邮件('CPU报警')

if 硬盘使用空间 > 90%:

发送邮件('硬盘报警')

if 内存占用 > 80%:

发送邮件('内存报警')

#调用函数

main()

今天我们来学习一种新的编程方式：面向对象编程（Object Oriented Programming，OOP，面向对象程序设计）

注：Java和C#来说只支持面向对象编程，而python比较灵活即支持面向对象编程也支持函数式编程。

面向过程和面向对象是解决问题的两种思路

**面向对象编程**

class MonitoringSystem(object):

"""监控系统工具"""

def 发送邮件(self,内容)

#发送邮件提醒

连接邮箱服务器

发送邮件

关闭连接

def 开始监控(self):

while True：

if cpu利用率 > 90%:

发送邮件('CPU报警')

if 硬盘使用空间 > 90%:

发送邮件('硬盘报警')

if 内存占用 > 80%:

发送邮件('内存报警')

ms = MonitoringSystem()

ms.开始监控()

### 1.1.2. 面向对象：**将数据与函数绑定到一起，进行封装，这样能够更快速的开发程序，减少了重复代码的重写过程。**

**同样去做一个很复杂的功能，用面向对象反而使其变得简单。**

注：Java和C#来说只支持面向对象编程，而Python比较灵活即支持面向对象编程也支持函数式编程

## 1.2. 菜鸟买电脑的被坑的故事

### 1.2.1. 第一种方式（面向过程）

1)在网上查找资料

2)根据自己预算和需求定电脑的型号 ThinkPad P70 顶配windows10系统 1W3

3)去市场找到联想店各种店无法甄别真假 随便找了一家

4)找到业务员,业务员推荐了另外一款 配置更高价格便宜,也是windows10系统的 1W

5)砍价30分钟 付款9088

6)成交

7)回去之后发现各种问题-开始走上维权之路

### 1.2.2. 第二种方式 （面向对象）

1)找一个靠谱的电脑高手

2)给钱交易

### 1.2.3. 第三种方式 （面向对象）

1)找一个靠谱的电商网站且靠谱的商家

2)给钱交易

3)收到电脑和说明的一样

4)给个五星好评

**面向对象和面向过程都是解决问题的一种思路而已**买电脑的第一种方式:

强调的是步骤、过程、每一步都是自己亲自去实现的

这种解决问题的思路我们就叫**做面向过程**

买电脑的第二种方式:

强调的是电脑高手, 电脑高手是处理这件事的主角,对我们而言,我们并不必亲自实现整个步骤只需要调用电脑高手就可以解决问题

这种解决问题的思路就 是**面向对象**

用面向对象的思维解决问题的重点

当遇到一个需求的时候不用自己去实现，如果自己一步步实现那就是面向过程

应该找一个专门做这个事的人来做

面向对象是基于面向过程的

## 1.3. 解决吃上真北京考鸭的问题

### 1.3.1. 第一种方式（面向过程）

1)养鸭子

2)鸭子长成

3)杀

4)作料

5)烤鸭

6)吃

### 1.3.2. 第二种方式（面向对象）:

1)找个卖北京烤鸭的店(最好正宗点的全聚德烤鸭)

2)吃

3)给钱 交易

4)吃北京烤鸭这个事情完成

需要了解的定义性文字:

面向对象(object-oriented ;简称: OO) 至今还没有统一的概念 我这里把它定义为: 按人们认识客观世界的系统思维方式,采用基于对象(实体) 的概念建立模型,模拟客观世界分析、设计、实现软件的办法。

面向对象编程(Object Oriented Programming-OOP) 是一种解决软件复用的设计和编程方法。 这种方法把软件系统中相近相似的操作逻辑和操作应用数据、状态,以类的型式描述出来,以对象实例的形式在软件系统中复用,以达到提高软件开发效率的作用。

# **2. 类和对象**

面向对象编程的2个非常重要的概念：类和对象，对象是面向对象编程的核心，在使用对象的过程中，为了将具有共同特征和行为的一组对象抽象定义，提出了另外一个新的概念——类

类就相当于制造飞机时的图纸，用它来进行创建的飞机就相当于对象

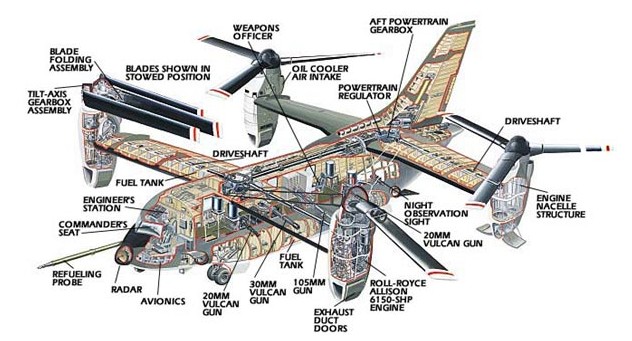
## 2.1. 类

人以类聚 物以群分。

具有相似内部状态和运动规律的实体的集合(或统称为抽象)。

具有相同属性和行为事物的统称

类是抽象的,在使用的时候通常会找到这个类的一个具体的存在,使用这个具体的存在。一个类可以找到多个对象



设计的时候就已经说明，用什么材料等等一系列问题。

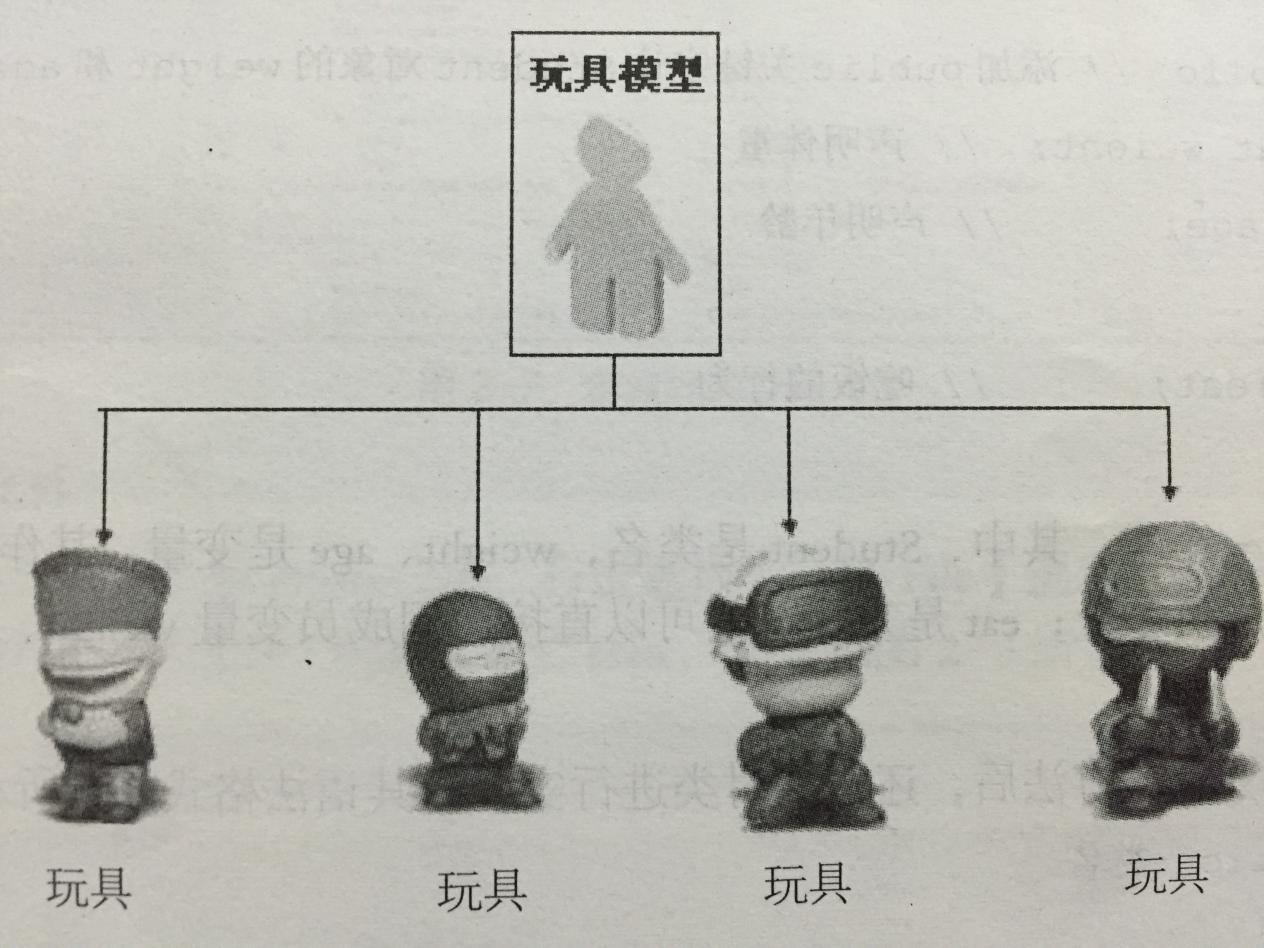
## 2.2. 对象

某一个具体事物的存在 ,在现实世界中可以是看得见摸得着的。

可以是直接使用的



## 2.3. 类和对象之间的关系



小总结：类就是创建对象的模板,对象是使用类创建出来的，而且可以创建任意多个对象。

## **2.4. 区分类和对象**

奔驰汽车 类

奔驰smart 类

张三的那辆奔驰smart 对象

狗 类

大黄狗 类

李四家那只大黄狗 对象

水果 类

苹果 类

红苹果 类 红富士苹果 类

我嘴里吃了一半的苹果 对象

我想要娶个如花似玉的媳妇

我想和我媳妇小花亲个小嘴。

## **2.5. 类的构成**

类(Class) 由3个部分构成

**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)

### 2.5.1. 举例

#### 2.5.1.1. 人类设计

只关心3样东西

事物名称(类名):人(Person)

属性:身高(height)、年龄(age)

方法(行为/功能):跑(run)、打架(fight)

#### 2.5.1.2. 狗类的设计

类名:狗(Dog)

属性:品种 、毛色、性别、名字、 腿儿的数量

方法(行为/功能):叫 、跑、咬人、吃、摇尾巴



## **2.6. 类的抽象**

如何把日常生活中的事物抽象成程序中的类?

**拥有相同(或者类似)属性和行为的对象都可以抽像出一个类**

方法:一般名词都是类(名词提炼法)

### 2.6.1. 坦克发射3颗炮弹轰掉了2架飞机

坦克-->可以抽象成 类

炮弹-->可以抽象成类

飞机->可以抽象成类

### 2.6.2. 小明在公交车上牵着一条叼着热狗的狗

小明--> 人类

公交车--> 交通工具类

热狗--> 食物类

狗--> 狗类

### 2.6.3. 有哪些类呢？



说明：

人

枪

子弹

垃圾桶

房子

购物车

### 2.6.4. 有哪些类呢？



说明:

向日葵

类名: xrk

属性:

行为: 放阳光

豌豆

类名: wd

属性: 颜色 、发型,血量

行为:发炮, 摇头

坚果:

类名:jg

属性:血量 类型

行为:阻挡;

僵尸:

类名:js

属性:颜色、血量、 类型、速度

行为:走 跑跳 吃 死

# **3. 定义类**

## 3.1. 定义一个类

class 类名:

方法列表

## 3.2. 定义一个Cat类

# 定义类

class Cat:

# 方法

def eat(self):

print('猫在吃鱼....')

def drink(self):

print("猫在喝东西...")

**说明：**

定义类时有2种：新式类和经典类，上面的Cat为经典类，如果是Cat(object)则为新式类

类名 的命名规则按照"大驼峰"

# **4. 创建对象**

**当创建一个对象时，就是用一个模子，来制造一个实物 。**

**类是创建实例对象的模板，可以创建很多个实例对象。**

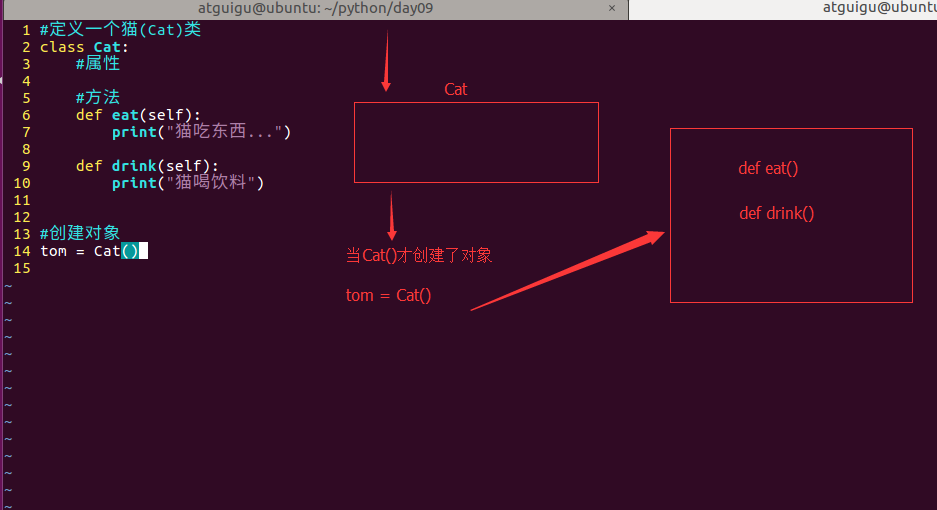


通过上一节课程，定义了一个Cat类；就好比有车一个张图纸，那么接下来就应该把图纸交给生成工人们去生成了，Python中，可以根据已经定义的类去创建出一个个对象。

## 4.1. 创建对象的格式为

对象名 = 类名()

## 4.2. 创建对象



### 4.2.1. 创建对象

# 定义类

class Cat(object):

# 吃

def eat(self):

print('猫在吃鱼....')

# 喝东西

def drink(self):

print("猫在喝东西...")

# 创建一个对象，并用变量tom来保存它的引用

tom = Cat()

### 4.2.2. 调用对象方法

# 定义类

class Cat(object):

# 吃

def eat(self):

print('猫在吃鱼....')

# 喝东西

def drink(self):

print("猫在喝东西...")

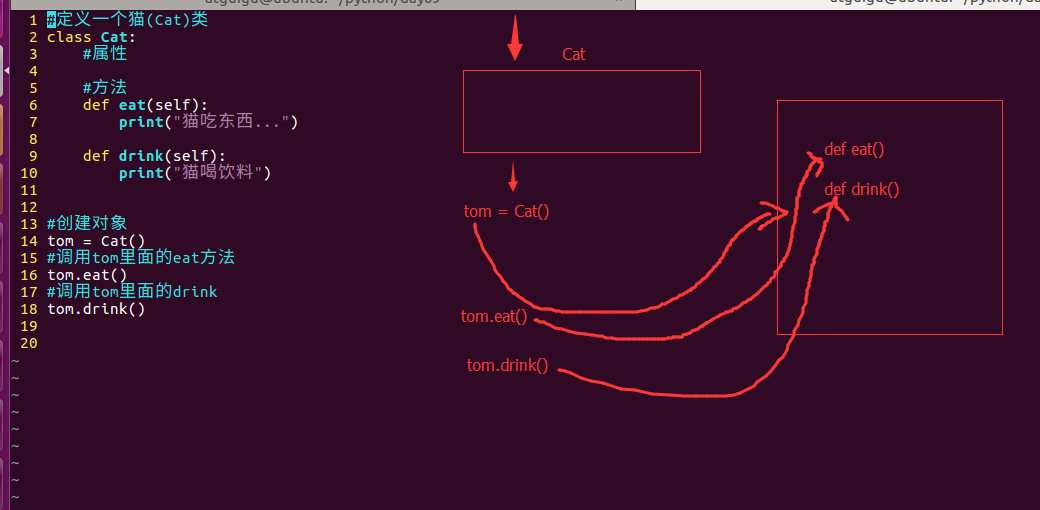
# 创建一个对象，并用变量tom来保存它的引用

tom = Cat()

tom.eat()

tom.drink()

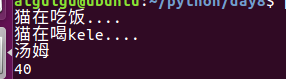
分析过程



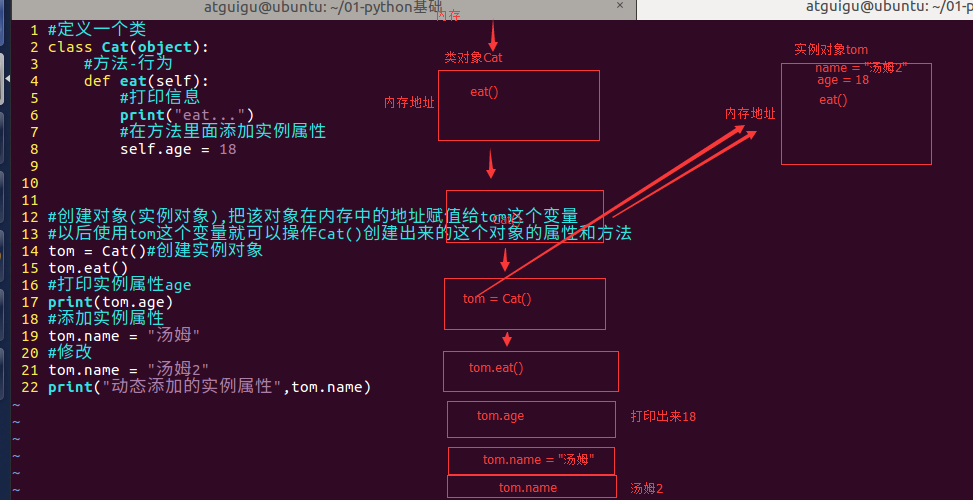
## 4.3. 添加属性

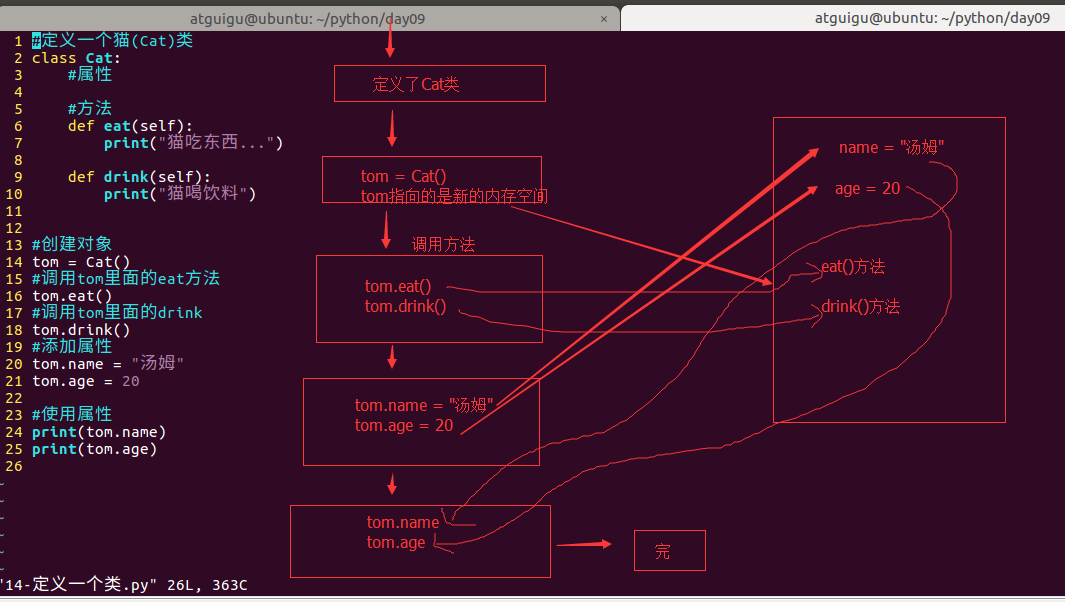


执行结果



## 4.4. 程序执行分析





## **4.5. 类和对象小结**

tom= Cat()，这样就产生了一个Cat的实例对象，此时也可以通过实例对象tom来访问属性或者方法

第一次使用tom.name= '汤姆'表示给tom这个对象添加属性，如果后面再次出现tom.name= xxx表示对属性进行修改

tom是一个对象，它拥有属性（数据）和方法（函数）

**当创建一个对象时，就是用一个模子，来制造一个实物**



# **5. 理解self**

## 5.1. 从创建多个对象引入self

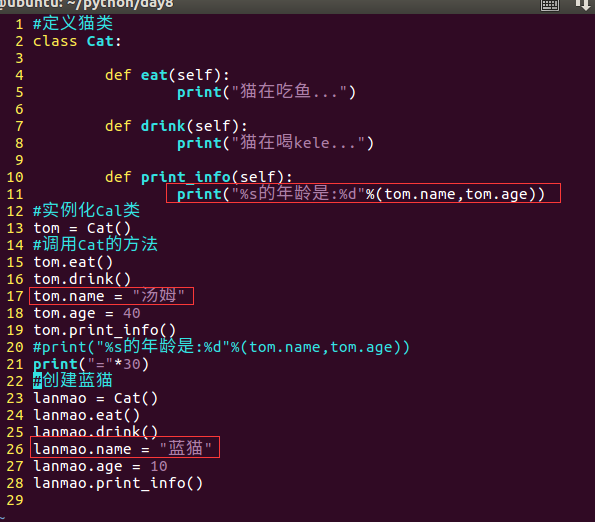
**某个对象调用其方法时，Python解释器会把这个对象作为第一个参数传递给self.**

通俗的话讲，就是谁调用这个方法就是谁。

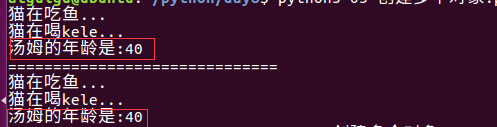
就像生活中谁拿起了一个自我介绍的卡片，自我介绍的时候就是谁。

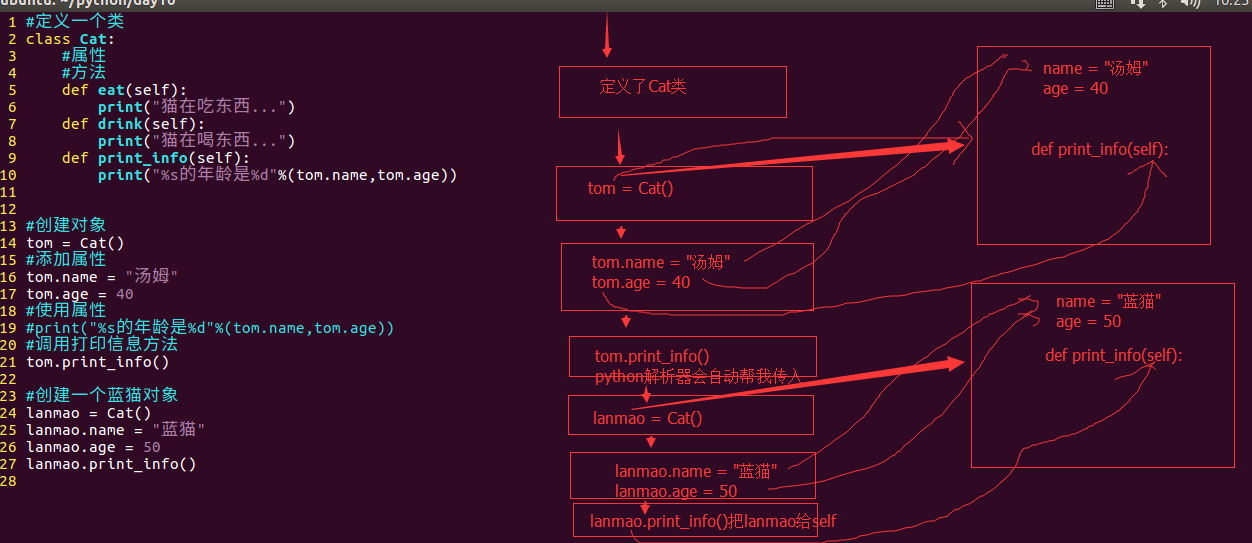
根据上两节创建一个Cat类，创建出多个猫对象，比如tom、lanmao等。

### 5.1.1. 案例1

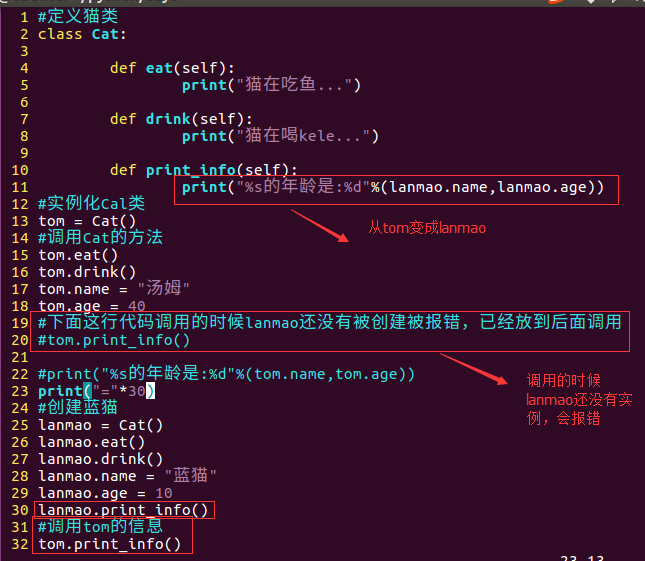


执行效果

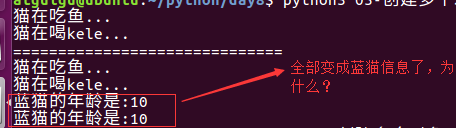




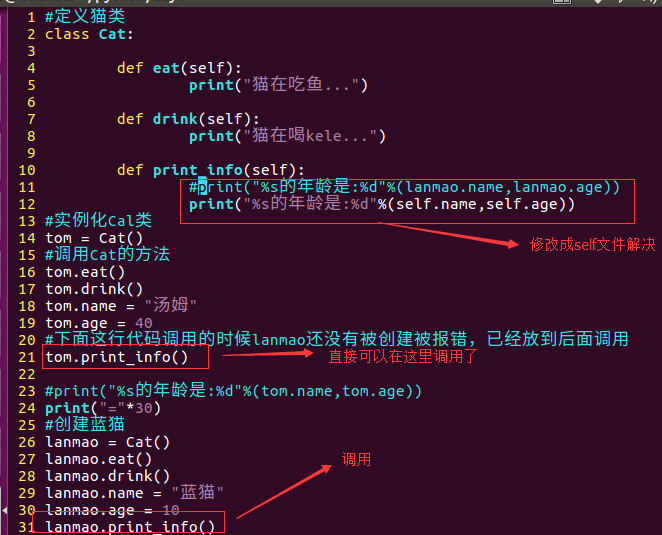
### 5.1.2. 案例2



执行效果



### 5.1.3. 使用self解决问题



那么什么是self呢

**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.**

通俗的话讲，就是谁调用这个方法就是谁。

就像生活中谁拿起了一个自我介绍的卡片，自我介绍的时候就是谁。

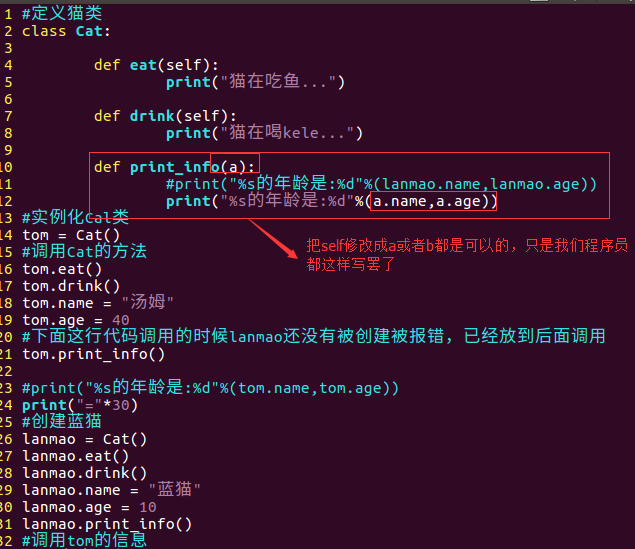
## **5.2. 理解self**

所谓的self，可以理解为自己可以把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。

**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**

**名字不一定叫self**

**调用的使用tom.eat()相当于tom.eat(tom)；但是我们实际上是不能真正把tom传入eat()方法的，Python解释器会自动帮我传递**



# 类的\_\_init\_\_()方法

1. 当创建实例对象成功后，有Python解释器来调用该方法，这个方法不用我们手动调用；
2. 误区，认为\_\_init\_\_方法用来创建对象的，初始化对象的一些数据；
3. 创建实例对象是用\_\_new\_\_方法创建的，使用\_\_new\_\_创建实例对象后要记得返回.

## 6.1. 想一想

在上一小节的demo中，我们已经给tom这个对象添加了2个属性，name（猫的名字）以及age（猫的年龄），试想如果再次创建一个对象的话，肯定也需要进行添加属性，显然这样做很费事，那么有没有办法能够在创建对象的时候，就顺便把Cat这个对象的属性给设置呢？

答:

\_\_init\_\_()方法

## **6.2. 使用方式**

### 6.2.1. 方式

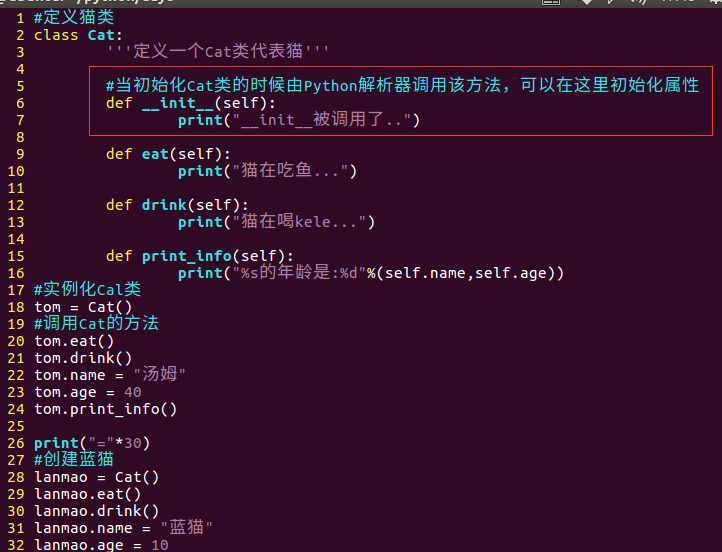
class 类名:

#初始化函数，用来完成一些默认的设定

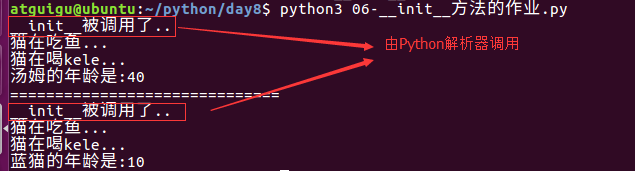
def \_\_init\_\_(self):

pass

### 6.2.2. 案例



执行结果



## 6.3. \_\_init\_\_()方法由Python解释器调用

### 6.3.1 在\_\_init\_\_()方法中给类设置默认属性

# 定义猫类

class Cat(object):

# 初始化对象

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

self.name= '汤姆'

self.age = 40

def eat(self):

print('猫在吃鱼...')

# 创建对象

tom= Cat()

print('猫的名字为:%s'%tom.name)

print('猫的年龄为:%d'%tom.age)

案例



执行效果

IMG_256

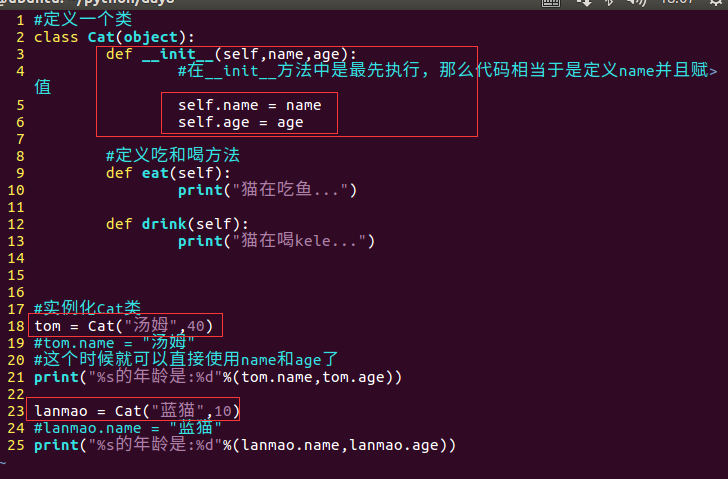
### 6.3.2. 小结

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

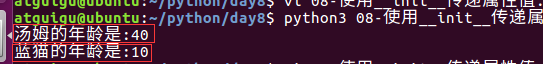
### 6.3.2. 使用\_\_init\_\_()传递参数

#### 6.3.2.1. 案例

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？



执行效果



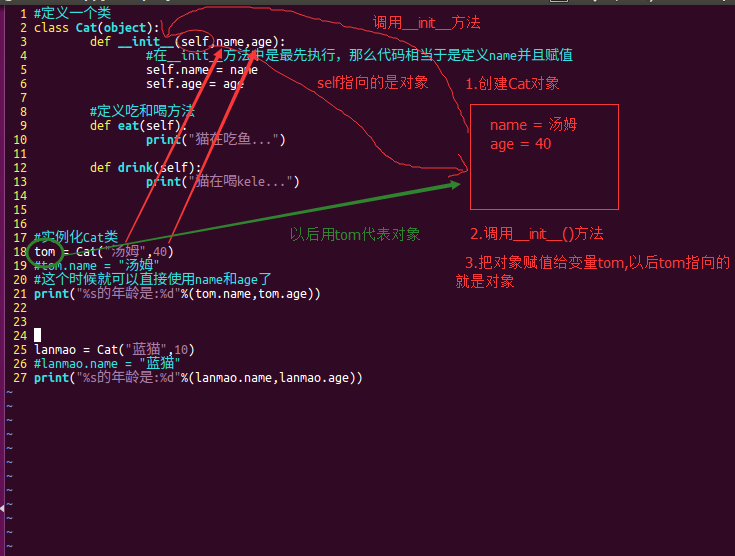
#### 6.3.2.2. 小结

\_\_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

## 6.4. \_\_init\_\_的执行流程

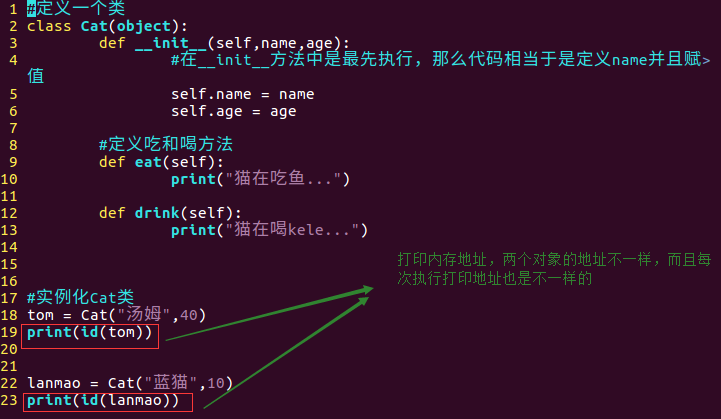




# **7. "魔法"方法**id()和\_\_str\_\_()

## 7.1. 打印id()

如果把tom使用print进行输出的话，会看到如下的信息



执行结果如下

IMG_256

即看到的是创建出来的tom对象在内存中的地址

## 7.2. 定义\_\_str\_\_()方法

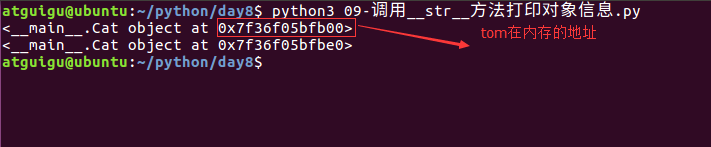
**当使用print输出对象的时候，只要自己定义了\_\_str\_\_(self)方法，那么就会打印从在这个方法中return的数据.**

该方法，在使用print打印对象的时候默认会调用该方法，通常用户输出当前对象的相关数据，在开发阶段的用得比较多。

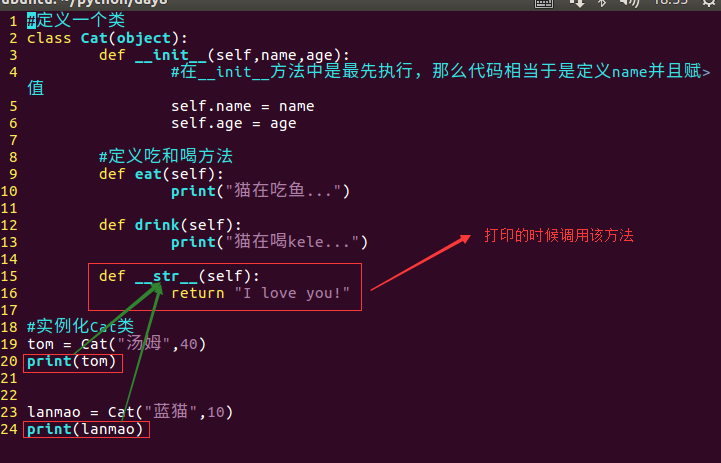
### 7.2.1. 案例



打印结果



使用\_\_str\_\_方法



执行结果

IMG_256

正确的打印



打印结果：

IMG_256

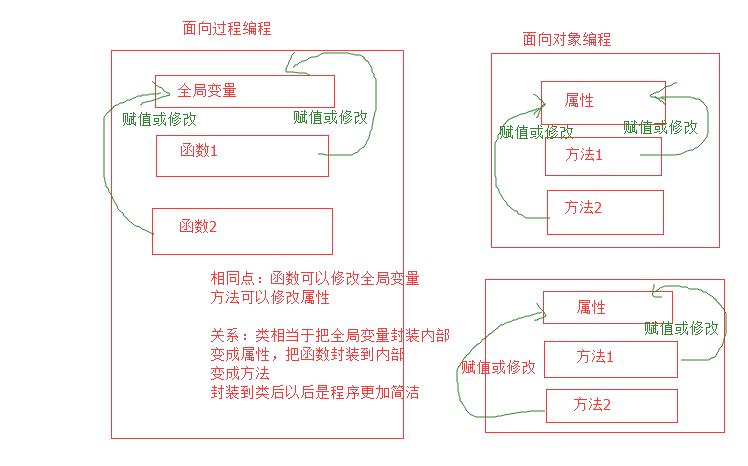
### **7.2.2. 小结**

在python中方法名如果是\_\_xxxx\_\_()的，那么就有特殊的功能，因此叫做“魔法”方法

**当使用print输出对象的时候，只要自己定义了\_\_str\_\_(self)方法，那么就会打印从在这个方法中return的数据**

# 8. 面向过程和面向对象编程联系和区别

## 8.1. 全局变量、函数和实例属性、方法的联系和区别



## 8.2. 关系和区别

全局变量，可以在各个函数中修改和使用。

属性也是可以在对象中各个方法修改和使用。

#全局变量

a = 100

def test1():

global a

a = 200

#局部变量，只能在函数内使用，用于存储函数的临时数据

b = 300

print("局部变量b==%s" %b)

def test2():

print("使用全局变量a==%d" % a)

#类

class Test(object):

#类属性

a = 100

def test1(self):

#实例属性或者说对象属性

self.b = 300

print("实例属性b==%s" % self.b)

def test2(self):

print("使用类属性a==%s" % Test.a)

print("使用类属性a==%s" % self.a)

#调用函数

test2()

test1()

test2()

#创建实例对象

t1 = Test()

#调用方法

t1.test1()

t1.test2()

#往t1对象里面添加name实例属性

t1.name = "张三"

t = Test()

#调用方法

t.test1()

t.test2()

#取出t1对象的实例属性

print(t1.name)

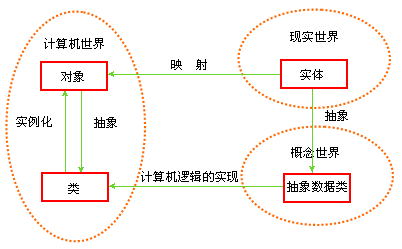
#下面的代码会保存，说明在t对象中添加的属性，在t对象是不能使用的

print(t.name)

不同之处：

**对象之间不可用相互修改数据**。

## 8.3. 类、对象、实体的相互关系和面向对象的解题思维方式



类、对象、实体的相互关系和面向对象的解题思维方式。在用面向对象的软件方法解决现实世界的问题时，首先将物理存在的实体抽象成概念世界的抽象数据类型，这个抽象数据类型里面包括了实体中与需要解决的问题相关的**数据**和**属性**；然后再用面向对象的工具，如Python和Java语言，将这个抽象数据类型用计算机逻辑表达出来，即构造计算机能够理解和处理的类；最后将类实例化就得到了现实世界实体的面向对象的映射—对象。在程序中对对象进行操作，就可以模拟现实世界中实体上的问题并解决之。  
　　实质上，面向对象技术的一个关键的设计思想就是要让计算机逻辑来模拟现实世界的物理存在，即让计算机世界向现实世界靠拢。这一点与传统的程序设计中把现实世界的问题抽象成计算机可以理解和处理的数据结构的思路，也即，使现实世界向计算机世界靠拢的思路是完全不同的。面向对象技术提出的这种新的解决问题的思路，使得我们可以用更接近于人类自然思维模式和更接近于现实问题本来面目的方法来设计解题模型。这样，无论是当时的设计实现本身，还是日后的维护、修改和扩充，都可以比较顺利、容易地在己有工作的基础之上完成，避免了用面向过程方法实现时需要面对的种种困难.

# **9. 面向对象应用:烤羊肉([Mutton](C:/Users/Administrator/AppData/Local/Yodao/DeskDict/frame/20160221235621/javascript:void(0);))**

为了更好的理解面向对象编程，下面以“烤羊肉”为案例，进行分析.

## **9.1. 分析“烤羊肉”的属性和方法**

### **9.1.1. 示例属性如下:**

cooked\_level : 这是数字；0~3表示还是**生的**，超过3表示**半生不熟**，超过5表示**已经烤好了**，超过8表示**已经烤成木炭了**！我们的羊肉开始时时生的。

cooked\_string : 这是字符串；描述羊肉的生熟程度。

condiments : 这是羊肉的配料列表，比如番茄酱、芥末酱等。

### **9.1.2. 示例方法如下:**

cook() : 把羊肉烤一段时间

add\_condiments() : 给羊肉添加配料

\_\_init\_\_() : 设置默认的属性

\_\_str\_\_() : 让print的结果看起来更好一些

## **9.2. 定义类，并且定义\_\_init\_\_()方法**

#定义`烤羊肉`类

class Mutton(object):

'这是烤羊肉的类'

#定义初始化方法

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

self.cooked\_level = 0

self.cooked\_string = "生的"

self.condiments = []

## **9.3. 添加"烤羊肉"方法cook**

#定义`烤羊肉`类

class Mutton(object):

'这是烤羊肉的类'

#定义初始化方法

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

self.cooked\_level = 0

self.cooked\_string = "生的"

self.condiments = []

#烤羊肉方法

def **cook**(self, time):

self.cooked\_level += time

if self.cooked\_level > 8:

self.cooked\_string = "烤成灰了"

elif self.cooked\_level > 5:

self.cooked\_string = "烤好了"

elif self.cooked\_level > 3:

self.cooked\_string = "半生不熟"

else:

self.cooked\_string = "生的"

## **9.4. 基本的功能已经有了一部分，赶紧测试一下**

把上面2块代码合并为一个程序后，在代码的下面添加以下代码进行测试

my\_mutton = Mutton()

print(my\_mutton.cooked\_level)

print(my\_mutton.cooked\_string)

print(my\_mutton.condiments)

完整的代码为:

#定义`烤羊肉`类

class Mutton(object):

'这是烤羊肉的类'

#定义初始化方法

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

self.cooked\_level = 0

self.cooked\_string = "生的"

self.condiments = []

#烤羊肉方法

def cook(self, time):

self.cooked\_level += time

if self.cooked\_level > 8:

self.cooked\_string = "烤成灰了"

elif self.cooked\_level > 5:

self.cooked\_string = "烤好了"

elif self.cooked\_level > 3:

self.cooked\_string = "半生不熟"

else:

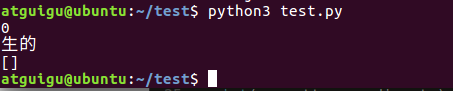
self.cooked\_string = "生的"

my\_mutton = Mutton()

print(my\_mutton.cooked\_level)

print(my\_mutton.cooked\_string)

print(my\_mutton.condiments)



## **9.5. 测试cook方法是否好用**

在上面的代码最后面添加如下代码:

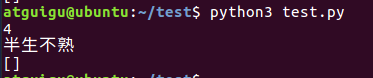
print("------接下来要进行烤羊肉了-----")

my\_mutton.cook(4) #烤4分钟

print(my\_mutton.cooked\_level)

print(my\_mutton.cooked\_string)

print(my\_mutton.condiments)



## 9.6. 定义add\_condiments()方法和\_\_str\_\_()方法

#定义`烤羊肉`类

class Mutton(object):

'这是烤羊肉的类'

#定义初始化方法

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

self.cooked\_level = 0

self.cooked\_string = "生的"

self.condiments = []

#烤羊肉方法

def cook(self, time):

self.cooked\_level += time

if self.cooked\_level > 8:

self.cooked\_string = "烤成灰了"

elif self.cooked\_level > 5:

self.cooked\_string = "烤好了"

elif self.cooked\_level > 3:

self.cooked\_string = "半生不熟"

else:

self.cooked\_string = "生的"

#打印当前类的相关数据信息

def \_\_str\_\_(self):

msg = self.cooked\_string + " 羊肉"

if len(self.condiments) > 0:

msg = msg + "("

for temp in self.condiments:

msg = msg + temp + ", "

msg = msg.strip(", ")

msg = msg + ")"

return msg

#添加佐料

def add\_condiments(self, condiment):

self.condiments.append(condiment)

my\_mutton = Mutton()

my\_mutton.cook(4) #烤4分钟

print(my\_mutton.cooked\_level)

print(my\_mutton.cooked\_string)

print(my\_mutton.condiments)

## **9.7. 再次测试**

完整的代码如下:

#定义`烤羊肉`类

class Mutton(object):

'这是烤羊肉的类'

#定义初始化方法

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

self.cooked\_level = 0

self.cooked\_string = "生的"

self.condiments = []

#烤羊肉方法

def cook(self, time):

self.cooked\_level += time

if self.cooked\_level > 8:

self.cooked\_string = "烤成灰了"

elif self.cooked\_level > 5:

self.cooked\_string = "烤好了"

elif self.cooked\_level > 3:

self.cooked\_string = "半生不熟"

else:

self.cooked\_string = "生的"

#打印当前类的相关数据信息

def \_\_str\_\_(self):

msg = self.cooked\_string + " 羊肉"

if len(self.condiments) > 0:

msg = msg + "("

for temp in self.condiments:

msg = msg + temp + ", "

msg = msg.strip(", ")

msg = msg + ")"

return msg

#添加佐料

def add\_condiments(self, condiments):

self.condiments.append(condiments)

my\_mutton = Mutton()

print("羊肉烤了4分钟")

my\_mutton.cook(4) #烤4分钟

print("------羊肉又经烤了3分钟-----")

my\_mutton.cook(3) #又烤了3分钟

print(my\_mutton)

print("------接下来要添加配料-番茄酱------")

my\_mutton.add\_condiments("番茄酱")

print(my\_mutton)

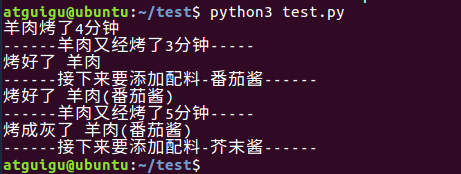
print("------羊肉又经烤了5分钟-----")

my\_mutton.cook(5) #又烤了5分钟

print(my\_mutton)

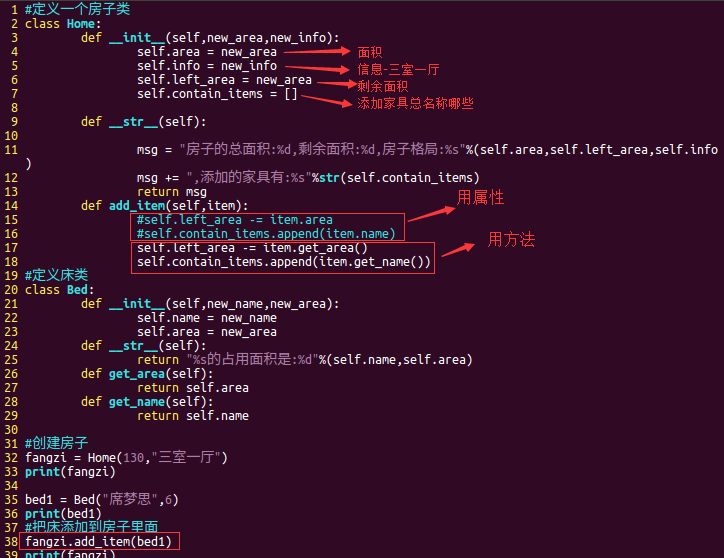
print("------接下来要添加配料-芥末酱------")

my\_mutton.add\_condiments("芥末酱")



# **10. 面向对象应用:存放家具**

## 10.1. 存放家具案例



## 10.2. 总结

如果一个对象与另外一个对象有一定的关系，那么一个对象可用是另外一个对象的属性

## 10.3. 思维升华：

添加定义一个类“书柜”，把它装到房子里面。

# **11. set和get方法隐藏数据**



可能你已经意识到，查看过着修改对象的属性（数据），有2种方法

## 11.1 直接通过对象名修改

my\_mutton.cooked\_level = 5

## 11.2. 通过方法间接修改

my\_mutton.cook(5)

## 11.3.分析

明明可以使用第1种方法直接修改，为什么还要定义方法来间接修改呢？至少有2个原因：

如果直接修改属性，烤羊肉至少需要修改2部分，即修改cooked\_level和cooked\_string。而使用方法来修改时，只需要调用一次即可完成。

如果直接访问属性，可能会出现一些数据设置错误的情况产生例如cooked\_level = -3。这会使羊肉比以前还生，当然了这也没有任何意义，通过使用方法来进行修改，就可以在方法中进行数据合法性的检查

# 总结与作业

## 12.1 总结

面向对象和面向过程的区别

类和对象的关系

定义类

创建对象

理解self

\_\_init\_\_方法

类、对象、实体、抽象数据类的关系

烤羊肉案例

存放家具案例