# 第六章 类和对象

# 目录

# 面向对象

# 对象

- **类和对象**
  - 类是一种具体事物的抽象，这些具体事物即为对象，它们是类的一个具体实例
  - 比如“大学生”，那么任何一个大学生都是大学生这个类的对象，每个对象都具有年龄、性别、毕业年份这些属性
  - 如果要使用类，就必须先将其实例化，我们通常不会说“大学生年龄很大”，但是可以说“这位大学生年龄很大”，其中的“这位大学生”就是“大学生”类的实例化对象

大学生1

对象的属性
姓名：王耀东
年龄：21
性别：男
绩点：3.51
……

都可以采用大学生共有的方法：改变姓名、改变成绩等等

大学生2

对象的属性
姓名：刘美华
年龄：22
性别：女
绩点：3.78
……

# 对象和面向对象

- Python语言是一种面向对象编程语言（Object-Oriented Programming Language），简称为OOP语言
- 无论是变量还是函数，它们都是属于某一个特定的类（class）(Python一切皆对象)。类中的对象叫做该类的实例（instance）
- 特定类的所有对象都与方法（method）相关联，这些方法类似于函数，编写一次之后，可以重复的调用
- 一旦某个对象属于某个特定类之后，该对象就可以使用该类的所有方法
- 面向对象程序设计的三要素分别为:封装、继承和多态，有时加上抽象

# 类的基本概念

- **以大学生社交网络信息数据集为例，所有的大学生即为一类**
- **类中有许多的属性**
  - 如姓名、年龄、性别、成绩
- **可对这些属性进行操作**
  - 如改变其姓名、比较两个大学生年龄的大小，提升或降低其分数等，这些操作被称为方法，这些属性和方法结合在一起，包裹起来就成为了类

**类名：**
**大学生**

**属性：姓名、年龄、性别、分数**

**方法：对属性进行查询或修改、与其他同类发生一些关系、运算等等**

# Python中创建类与对象(实例)

# 创建类

定义类的关键词　　类的名称

```
Class Student :

    school_name="Northeastern University "    #类属性
    '''
    A class of  student
    '''
    def __init__(self, name):
        self.name = name                      #实例对象的属性
        self.gender = 1
        self. amount+ =1
        self.illness = False

    def display_count (self):

        return ("Total  Student_ amount %d" % Student. amount )
```

类的文档

初始化方法

普通方法

# 创建类

- 用class类名即可以创建一个类
- 在类名的程序块中可以定义这个类的属性、方法等等
  - 如下面的创建了一个High_school_student类，并定义了这个类的一个方法——初始化方法__init()__(双下划线)

```
class High_school_student():
    def __init__(self):
        self.age = 18
        self.gender = 'M'
        self.gradyear = 2006
```

self代表类的实例，而非类

# 实例

- 创建实例
  - 在创建完High_school_student类之后，可创建相应的实例

语法

```
student_a = High_school_student()
print(student_a)
print(High_school_student)
```

```
<__main__.High_school_student object at 0x00000000051EB358>
<class '__main__.High_school_student'>
```

```
print(id(student_a))
```

```
2246314215600
```

```
print(type(student_a))
```

```
<class '__main__.High_school_student'>
```

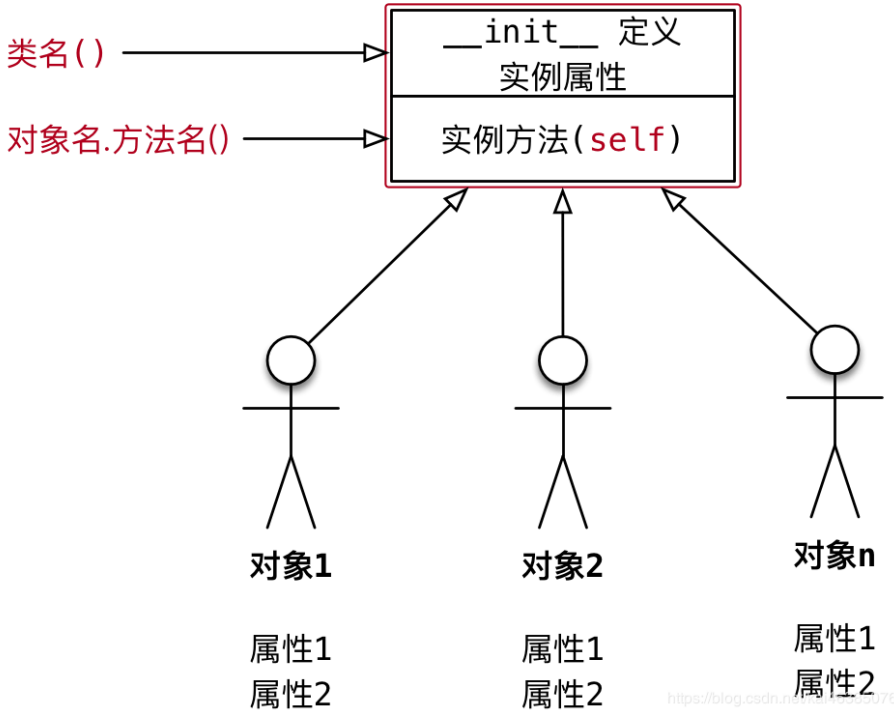
  - 可以看到，student_a是High_school_student类的一个实例，存储在“0x00000000051EB358”（内存地址）中。如果想看"student_a"的年龄"age"，则可以像模块中调用方法一样使用点符号"."

```
student_a.age
```

```
18
```

**类和对象**

- 对象是用来描述客观事物的一个实体
- 类定义了对象将会拥有的特征（属性）和行为（方法）
- 类是对象的类型，对象是类的实例
- 使用类的步骤
  1. 定义类：使用关键字class
  2. 创建类的对象：使用关键字new
  3. 使用类的属性和方法：使用“.”操作符




实例化类其他编程语言中一般用关键字 new，但是在 Python 中并没有这个关键字，类的实例化类似函数调用方式。(Java)

# 属性和方法

# 属性

- **类属性**
  - 类属性就是类对象所拥有的属性，它被**所有**类对象的实例对象所**共有**
  - 在__init__()外初始化
  - 在内部用类名.类属性名调用
  - 外部既可以用类名.类属性名又可以用实例化对象.类属性名来调用

- **实例属性**
  - 在__init__(self,...)中初始化
  - 内部调用时都需要加上self
  - 外部调用时用实例化对象.属性名

# 类属性

- 定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到

```
class People(object):
    graduatedschool  = '东北大学'  #公有的类属性
p = People() #类的实例对象

print(p. graduatedschool) #实例化对象.类属性名来调用
print(People. graduatedschool )      #类名.类属性名调用
```

```
东北大学
东北大学
```

```
dir(object)
```

```
['__class__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__']
```

python3中，类定义默认继承object，所以写不写没有区别。
目的是便于统一操作。继承object类是为了让自己定义的类拥有更多的属性。
对于不太了解python类的同学来说，这些高级特性基本上没用处，但是对于那些要着手写框架或者写大型项目的高手来说，这些特性就比较有用了，比如说tornado里面的异常捕获时就有用到__class__来定位类的名称，还有高度灵活传参数的时候用到__dict__来完成等。
我们平时写程序时，不妨养成良好的习惯，将object类继承上。

# 实例属性

```
class People(object):
    language = 'python' #类属性
    def __init__(self):
        self.name = 'xiaowang' #实例属性
        self.age = 20 #实例属性


p = People()
p.age =12 #实例属性
print(p.language) #对象调用类属性 正确
print(p.name)    #对象调用实例属性 正确
print(p.age)     #对象调用实例属性 正确

print(People.language) #类调用类属性 正确
print(People.name)    #类调用实例属性 错误
```

该属性非公共属性！应是**所有**类对象的实例对象所**共有**

**东北大学的名字**

```
python
xiaowang
12
python
------------------------------------------------------------------------
---
AttributeError                            Traceback (most recent
call last)
<ipython-input-141-c754dbe395d9> in <module>()
     11
     12 print(People. language) #类调用类属性 正确
---> 13 print(People.name)    #类调用实例属性 错误
AttributeError: type object 'People' has no attribute 'na
```

# 实例属性

- 实例属性优先级比类属性高，因此它会屏蔽掉类的language属性，但是类属性并未消失，用People.language仍然可以访问
- 删除实例属性后再次调用s.language，由于实例的language属性没有找到，类的language属性就会显示出来

```
class People(object):
    language = 'python' #类属性
print(People.language)
p = People()
print(p.language)
p.language = 'java'
print(p.language)  #实例属性会屏蔽掉同名的类属性
print(People.language) #类属性不会因实例属性变化而改变
del p.language    #删除实例属性
print(p.language)
```

```
python
python
java
python
python
```

# self的作用

- __init__是一种特殊的方法
  - 在我们创建类的实例（对象）时，Python解释器会自动调用该函数，使得实例一开始就拥有类模板所具有的属性
  - 在定义 __init__ ()方法时添加了一个 self 参数，这个参数是类中函数定义时必须使用的参数，并且永远是第一个参数，该参数表示创建的实例本身。通过__init__ ()方法，初始化的是对象的属性，而不是类的属性

```
class Hight_school_student():
    def __init__(self,age,sex):
        self.age = age
        self.sex = sex
```

# self的作用

- __init__()存在除了 self 之外的参数时，在创建实例时，需要传入相应的参数值
  - self 不需要传入数值，Python解释器会自动将实例变量传入，但会给实例变量绑定属性

```
class Hight_school_student():
    def __init__(self,age,sex):
        self.age = age
        self.sex = sex
student_a = Hight_school_student()#括号的内容必须有除self的精确匹配的初始化参数列表
```

```
---------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-145-fcce3cda30e8> in <module>()
----> 1 student_a = Hight_school_student()

TypeError: __init__() missing 2 required positional arguments:
'age' and 'sex'
```

左边的错误信息标明：

__init__()方法需要有3个参数（其中一个已经给定，即为"self"）

必须传入相应的"age"和"sex"参数

# self的作用

- 传入了两个参数——18与“M”，则可以成功创建一个对象，这个对象的年龄是18，性别是“M”
- 可以不断创建拥有不同年龄与性别的高中生对象实例

```
student_a = Hight_school_student(18,'M')
print(student_a.age)
print(student_a.sex)
student_b = Hight_school_student(19,'M')
print(student_b.age)
print(student_b.sex)
```

```
18
M
19
M
```

# 方法

- 类中的方法定义的方式与函数相同，但类中的方法与实例绑定
  - 检测实例对象中的age属性是否为缺省值，可以自定义相应的方法，如"missing_detecting"方法：
  - 类中的方法的第一个参数必须为"self"

```
class High_school_student():
    def __init__(self,age,sex):
        self.age = age
        self.sex = sex
    def missing_detecting(self):
        if self.age =='':
            print('This is a missing value!' )
        else:
            print('This is not a missing value!')

student_a = High_school_student('','F')#年龄为空
student_a.missing_detecting()
```

```
This is a missing value!
```

# 实例方法

- 自定义的方法与__init__()方法不同，它不会在创建对象的时候就自动调用，而需要在程序中主动调用

  如上面的student_a.missing_detecting()。类似missing_detecting的这种自定义方法也称作实例方法

- 在一个类里定义了一个实例方法，则这个类的每个对象都可以执行这个实例方法，但是这个类本身不能执行这个实例方法

  例如，每个大学生都可以改变自己的姓名，但是“大学生”类本身没办法改变自己的姓名(不合逻辑)

```
1  class Hight_school_student():
2      def __init__(self):
3          self.age = 12
4          self.sex = 1
5      def school(self):
6          return 'highschool'
7  student_a = Hight_school_student()
8  student_a.school()
9  Hight_school_student.school()
```

```
---------------------------------------------------------------------
-
TypeError                                 Traceback (most recent c
<ipython-input-36-f71a3142947b> in <module>
      7 student_a = Hight_school_student()
      8 student_a.school()
----> 9 Hight_school_student.school()

TypeError: school() missing 1 required positional argument: 'self'
```

# 类方法

- 类方法可以通过类直接调用，或通过实例直接调用，但无论哪种调用方式，最左侧传入的参数一定是类本身(区别于实例)
- 一般类方法使用@classmethod装饰器来声明

```
class ClassA(object):   #新式类-有object的
    @classmethod
    def func_a(cls):
        print(type(cls), cls)  #cls表示类本身(无需实例化)
ClassA.func_a()
ca = ClassA()
ca.func_a()
```

```
<class 'type'> <class '__main__.ClassA'>
<class 'type'> <class '__main__.ClassA'>
```

# 类方法

```
class CLanguage:
    #类构造方法，也属于实例方法
    def __init__(self):
        self.name = "C语言中文网"
        self.add = "http://c.biancheng.net"
    #下面定义了一个类方法
    @classmethod
    def info(cls):
        print("正在调用类方法",cls)
```

```
#使用类名直接调用类方法
CLanguage.info()
#使用类对象调用类方法
clang = CLanguage()
clang.info()
```

```
正在调用类方法 <class '__main__.CLanguage'>
正在调用类方法 <class '__main__.CLanguage'>
```

cls表示类本身，cls 参数的命名也不是规定的（可以随意命名），只是 Python 程序员约定俗称的习惯而已。Self同样

# 静态方法

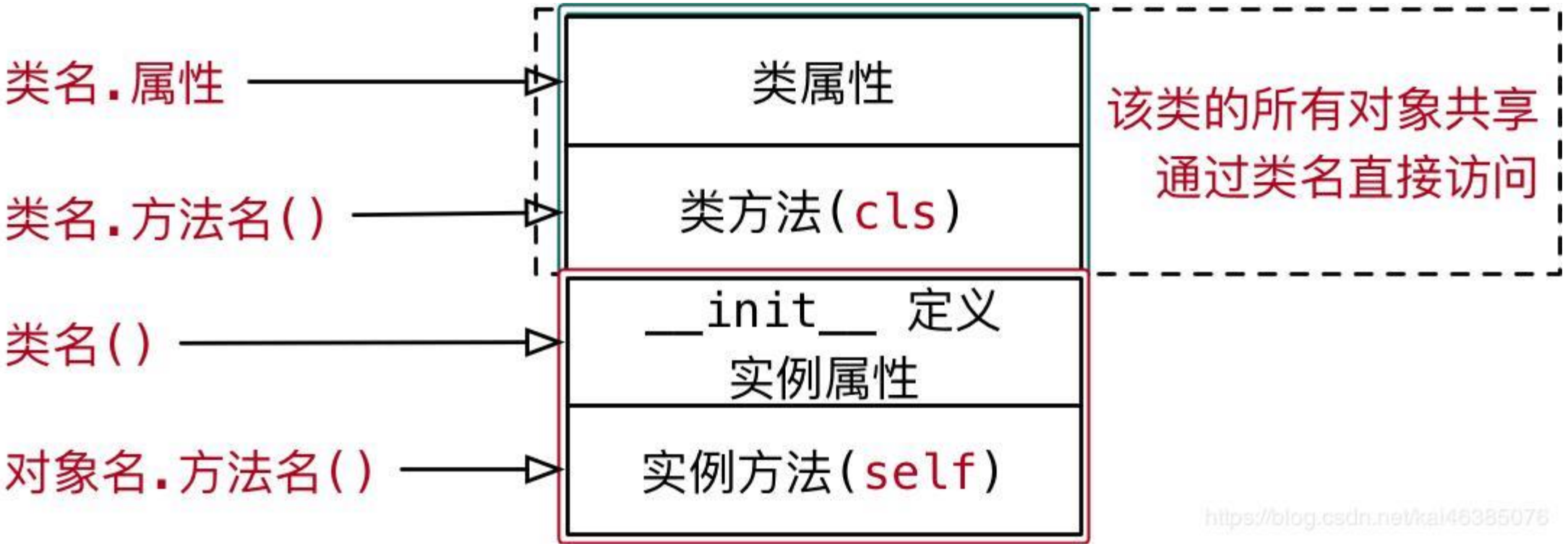
- 静态方法是指类中无需实例参与即可调用的方法(不需要self参数，也不需要cls)
  - 在调用过程中，无需将类实例化，直接在类之后使用“.”号运算符调用方法
  - 方法体中不能使用**类或实例的任何属性**和方法
- 通常情况下，静态方法使用@staticmethod装饰器来声明

```
class ClassA(object):
    @staticmethod
    def func_a():
        print('Hello Python')
ClassA.func_a()
# 也可以使用实例调用,但是不会将实例作为参数传入静态方法
ca = ClassA()
ca.func_a()
```

```
Hello Python
Hello Python
```

静态方法作用：用来存放逻辑性的代码，逻辑上属于类，但是和类本身没有关系，也就是说在静态方法中，不会涉及到类中的属性和方法的操作。可以理解为，静态方法是个独立的、单纯的函数，它仅仅托管于某个类的名称空间中，便于使用和维护----与其它语言不同。

例如：定义一个关于时间操作的类，其中有一个获取当前时间的函数。


类名.属性
类名.方法名()
类名()
对象名.方法名()
类属性
类方法(cls)
__init__ 定义
实例属性
实例方法(self)
该类的所有对象共享
通过类名直接访问

# 继承和多态

# 继承

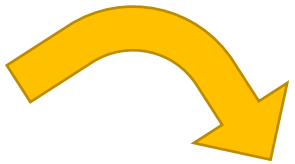
- 继承的概念

语法　　class ClassName(BaseClassName):

  - 在编程语言中，如果一个新的类继承另外一个类，那么这个新的类成为子类（Subclass），被子类继承的类称为父类
    - 可以把更加一般、范围大的类的属性在父类定义，把更加具体、范围小的特点在子类定义
  - 类的继承好比孩子与父母之间的继承关系一样，孩子拥有父母所拥有的许多特性
  - 例如定义“人类”这个类，包含姓名、身高等属性；“人类”有两个子类“学生”、“老师”，“学生”有属性“学号”、“成绩”，老师有属性“工号”、“专业”等
  - 每一个“学生”的实例，自然拥有“学生”的属性“学号”、“成绩”以及其父类“人类”的属性“学号”、“成绩”等等

# 继承

```
class High_school_student():
    def __init__(self):
        f = open('teenager_sns.csv')
        csvreader = csv.reader(f)
        teenager_sns = list(csvreader)
        #删除第一个元素，即变量名列表元素
        del teenager_sns[0]
        self.teenager_sns = teenager_sns

    def missing_detecting(self,age,sex):
        if age =='' or sex=='NA':
            missing = True
        else:
            missing = False
        return missing
```



语法

```
class Male_student(High_school_student):
    pass
class Female_student(High_school_student):
    pass
```

# 继承

- 子类Male_student的实例male_student继承了父类的初始化属性self.teenager_sns

```
male_student = Male_student()
print(male_student.teenager_sns[0])
```

```
['2006', 'M', '18.98', '7', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '1', '0', '0', '0', '0',
'0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0', '0']
```

- 子类也继承missing_detecting()实例方法

```
print(male_student.missing_detecting(male_student.teenager_sns[0][0],male_student.teenager_sns[
0][1]))
```

```
False
```

# 继承

- 面向对象的继承性
  - 如果子类也定义了同名的方法，则会覆盖父类的方法
  - 当子类无初始化__init__()方法时，在创建实例时，会自动调用父类的__init__()方法，因此子类的实例也会有父类的属性
  - 在调用实例方法时，系统会首先从子类中查找有没有该实例方法
    - 如果存在，那就运行该方法
    - 如果不存在，那么就从其上一级类，也就是其继承的父类中查找该方法
  - 在调用实例方法时，子类的优先级最高(屏蔽父类)，属性亦是如此

# 继承

- 面向对象的继承性
  - 在子类中定义missing_detecting()方法

```
class Male_student(High_school_student):
    def missing_detecting(self,age,sex):
        print('This is a subclass method!')
```

  - 创建子类实例并调用missing_detecting()方法

```
male_student = Male_student()
male_student.missing_detecting(male_student.teenager_sns[0][0],male_student.teenager_sns[0][1])
#可以看出这里我们用的是子类的missing_detecting，而不是父类的missing_detecting!
```

```
This is a subclass method!
```

# 继承

- 面向对象的继承性
  - 如果不注意继承性的这个特点，有可能导致错误

```
class Male_student(High_school_student):
    def __init__(self):
        self.male_info = 'this is subclass attribute!'
    def missing_detecting(self,age,sex):
        print('this is a subclass method!')

male_student = Male_student()#调用父类的初始化内容

male_student.teenager_sns   #
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-51-06be850871dd> in <module>()
      1 male_student = Male_student()
----> 2 male_student.teenager_sns

AttributeError: 'Male_student' object has no attribute
'teenager_sns'
```

在子类中定义__init__()方法，那么在调用父类初始化方法中的属性时，将会产生错误，

在创建实例时，调用的初始化方法是子类定义的，因而只有子类初始化方法中的属性才存在！

# 单继承

- 单继承概念
  - 单继承的子类只继承了一个父类
    - “人”作为一个类，“学生”作为子类继承“人”这个类

```
#定义父类
class people:
  #定义基本属性
  name =''
  age =0
  #定义私有属性,私有属性在类外部无法
直接进行访问
  __weight =0
  #定义初始化方法
  def __init__(self,n,a,w):
    self.name = n
    self.age = a
    self.__weight = w
  def speak(self):
    print("%s 说: 我 %d 岁。
"%(self.name,self.age))
```

```
#单继承示例
class student(people):
  grade =''
  def __init__(self,n,a,w,g):
    #调用父类的初始化函数，下面两种方式都可以
    #people.__init__(self,n,a,w)
    super().__init__(n,a,w)
    self.grade = g
    #复写父类的方法
  def speak(self):
    print("%s 说: 我 %d 岁了，我在读 %d 年级
"%(self.name,self.age,self.grade))
s = student('ken',10,60,3)
s.speak()
```

```
ken 说: 我 10 岁了，我在读 3 年级
```

# 多继承

- 多继承
  - Python同样有限的支持多继承形式。多继承的类定义形如下例:

```
class DerivedClassName(Base1, Base2, Base3):
  <statement-1>
  .
  .
  .
  <statement-N>
```

  - 需要注意圆括号中父类的顺序，若是父类中有相同的方法名，而在子类使用时未指定，python从左至右搜索 即方法在子类中未找到时，从左到右查找父类中是否包含方法(因此需要考虑顺序对实现功能的影响)。

# 多继承

```
#另一个类，多重继承之前的准备
class Programmer ():
  topic ='' #基本属性
  name =''
  def __init__(self,n,t):
    self.name = n
    self.topic = t
  def speak(self):
    print("我叫 %s，我是一个程序员，我用的语言是 %s"%(self.name,self.topic))
```

```
#多重继承
class sample(Programmer,student):
  a =''
  def __init__(self,n,a,w,g,t):
    student.__init__(self,n,a,w,g)
    Programmer.__init__(self,n,t)
test = sample("Tim",25,80,4,"Python")
test.speak()#方法名同，默认调用的是在括号中排前的父类的方法
```

```
我叫 Tim，我是一个程序员，我用的语言是 Python
```

# 多态

- 多态
  - 多态性是指不考虑实例类型的情况下使用实例---打开(电冰箱，电视，灯)
  - 意味着变量并不知道引用的对象是什么，根据不同的引用对象表现不同的行为方式
- 多态的好处
  - 当传入更多的子类，只需要继承父类型（通过父类实现方法）
  - 子类中的方法既可以**直接使用**父类的，也可以重写一个特有的
  - 增加程序的灵活性以及可扩展性
  - 著名的**“开闭”原则**

    对扩展开放（Open for extension）：允许子类重写方法函数

    对修改封闭（Closed for modification）：不重写，直接继承父类方法函数

# 多态(非继承)

- 没有继承关系的类实现多态
  - 每个类定义，同名的方法
  - 单独定义一个和该方法同名的类，类定义方法 object.方法()
  - 不同类的实例实现该方法时，方法(实例)

```python
class Duck(object):  # 鸭子类
    def fly(self):
        print("鸭子起飞")

class Swan(object):    # 天鹅类
    def fly(self):
        print("天鹅起飞")

class Plane(object):   # 飞机类
    def fly(self):
        print("飞机起飞")

def fly(object):      # 实现飞起飞
    object.fly()

duck = Duck()
fly(duck)

swan = Swan()
fly(swan)

plane = Plane()
fly(plane)

```

```
鸭子起飞
天鹅起飞
飞机起飞
```

# 多态(类的继承)

```
class Person(object):
    def __init__(self,name,sex):
        self.name = name
        self.sex = sex
    def print_title(self):
        if self.sex == "male":
            print("man")
        elif self.sex == "female":
            print("woman")

class Child(Person):  # Child 继承 Person
    def print_title(self): # 子类方法覆盖了父类方法
        if self.sex == "male":
            print("boy")
        elif self.sex == "female":
            print("girl")
tom = Child("tom","female") # 创建实例
Peter = Person("Peter","male")
```

```
print(tom.name, tom.sex,Peter.name,Peter.sex)
tom.print_title()
Peter.print_title()
```

```
tom female Peter male
girl
man
```

tom 和peter属于不同的类的实例，但是他们都有print_title方法，于是可以不用考虑两者就是什么类型而直接进行方法调用

# 多态(类的继承)

import abc
( metaclass=abc.ABCMeta) # 同一类事物:动物等

•具有不同功能的函数可以使用相同的函数名，这样就可以用一个函数名调用不同内容的函数。
向不同的对象发送同一条消息，不同的对象在接收时会产生不同的行为（即方法）。
•每个对象可以用自己的方式去响应共同的消息。所谓消息，就是调用函数，不同的行为就是指不同的实现，即执行不同的函数。
•代码子类是约定俗称的实现这个方法，加上@abc.abstractmethod装饰器后严格控制子类必须实现这个方法
•注意实现的方法

```
import abc
class Animal(metaclass=abc.ABCMeta): #同一类事物:动物
    @abc.abstractmethod
    def talk(self):
        pass

class Cat(Animal): #动物的形态之一:猫
    def talk(self):
        print('say 瞄瞄')

class Dog(Animal): #动物的形态之二:狗
    def talk(self):
        print('say 汪汪')

class Pig(Animal): #动物的形态之三:猪
    def talk(self):
        print('say 哼哼')

c = Cat()
d = Dog()
p = Pig()

def func(obj):
    obj.talk()

func(c)
func(d)
func(p)
```

```
say 瞄瞄
say 汪汪
say 哼哼
```

# 封装与私有化

# 封装与私有化

- 封装
  - 对一个对象的成员属性进行修改或访问，就必须通过对象允许的方法来进行（例如要求输入密码以确认拥有此权限等）。这样可以保护对象，使程序不易出错
  - 如在上面的 High_school_student 类中，虽然数据已经被封装在类里面，但是还是可以通过外部访问其中的变量，可以在外部对 age 进行修改

```
class High_school_student():
    def __init__(self):
        self.age = 18.0
        self.sex = 'M'

student_a = High_school_student()
print(student_a.age)
student_a.age=18.8
print(student_a.age)
```

```
18.0
18.8
```

左边的例子就说明这个High_school_student类没有进行有效的封装！

# 封装与私有化

- 私有化的方法
  - 在属性名称前加上两个下划线 __ ，表示将该属性成员私有化，该成员在类内部可以被访问，但是在类外部不能够访问

```
class High_school_student():
    def __init__(self):
        self.__age = 18.0
        self.__sex = 'M'

student_a = High_school_student()
student_a.__age
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-62-50a294a4198b> in <module>()
      5
      6 student_a = High_school_student()
----> 7 student_a.__age

AttributeError: 'High_school_student' object has no attribute '__age'
```

# 封装与私有化

- 在外部修改私有化的变量的值，系统并不会报错
  - student_a.__age = 18.8 这一句代码表示给 student_a 新增了一个 __age 变量，而内部的 __age已经被Python解释器自动命名为_High_school_student__age

```
student_a.__age = 18.8
print(student_a.__age)
```

```
18.8
```

# 封装与私有化

- 成员私有化并不是代表完全不能够从外部访问成员，而是提高了访问的门槛，防止意外或者随意改变成员，引发错误

- 通过"_类名"+"私有变量"，对变量进行访问

```
student_a.__age = 18.8
print(student_a._High_school_student__age)
print(student_a.__age)
```

```
18.0
18.8
```

# 封装与私有化

- 成员私有化不仅包括属性的私有化，也包括了方法的私有化，在方法名称前加上"__"也可以使得函数只能被内部访问，不能够被外部访问

```
#对missing_detecting方法的访问将被阻止!
class High_school_student():
    def __init__(self):
        self.age = 18.0
        self.age = 'M'
    def __missing_detecting(self):
        if age =='' or sex=='NA':
            missing = True
        else:
            missing = False
        return missing
student_a = High_school_student()
student_a.__missing_detecting
```

```
---------------------------------------------------------------------
--
AttributeError                            Traceback (most recent
call last)
<ipython-input-66-91bd2dc27119> in <module>()
     10         return missing
     11 student_a = High_school_student()
---> 12 student_a.__missing_detecting

AttributeError: 'High_school_student' object has no
attribute '__missing_detecting'
```

# 自定义对象类型

# 简单的对象类型

- 为了实现特有的属性和方法：使用特定函数，重写某些特殊方法以实现特有的程序
- 类，就是“类型”
- 特殊方法：
  - __str__()
  - __repr__()
  - 注意__repr__方法对解析器友好，__str__方法对人友好

# 简单的对象类型

- __str__()简介
  - 如果要把一个类的实例变成 str，就需要实现特殊方法__str__()，相当于对于这个对象的描写(返回用户易读)

```
#我们先定义一个Student类，打印一个实例
class Student(object):
    def __init__(self, name):
        self.name = name
print(Student('jack'))
```

```
<__main__.Student object at 0x0000000005044208>
```

```
#定义好__str__()方法，返回字符串
class Student(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Student object (name: %s)' % self.name
print(Student('jack'))
```

```
]: class Student(object):
       def __init__(self, name):
           self.name = name
       def __str__(self):
           return 'Student object (name: %s)' % self.name
   print(Student('jack'))
   a = Student('LL')
   print(type(a))
```

```
Student object (name: jack)
<class '__main__.Student'>
```

# 简单的对象类型

- __repr__()简介
  - __str__()返回用户看到的字符串，而__repr__()返回程序开发者看到的字符串，也就是说，__repr__()是为调试服务的，实际上__str__只是覆盖了__repr__以得到更友好的用户显示。（表示清楚，为开发者准备）

```
class Student():
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Student object (name=%s)' % self.name
    __repr__ = __str__
```

```
In [27]:
1 class Haha:
2     def __str__(self):
3         return 'This is str'
4     def __repr__(self):
5         return 'This is repr'
6 f=Haha()

In [28]:
1 f
Out[28]: This is repr

In [29]:
1 print(f)
This is str
```

# 简单的对象类型

```
In [27]:
1 class Haha:
2     def __str__(self):
3         return 'This is str'
4     def __repr__(self):
5         return 'This is repr'
6 f=Haha()

In [28]:
1 f

Out[28]: This is repr

In [29]:
1 print(f)

This is str
```

## __str__()

类的__str__()方法会在某些需要将对象转换为字符串的时候被调用，便于阅读信息

## __repr__()

类的__repr__()方法会在直接查看对象时调用,便于调试

# 可调用对象---类的实例

- __call__()函数：
  - 当一个类实现__call__()方法时，这个类的实例就会变成可调用对象

```
1  class CLanguage:
2      # 定义__call__方法
3      def __call__(self,name,add):
4          print("调用__call__()方法",name,add)
5  clangs = CLanguage()
6  clangs("abc","123")
```

```
调用__call__()方法 abc 123
```

如果类定义了__call__方法，那么它的实例可以作为函数进行调用。并且__call__方法可以进行自定义重写。

通过在 CLanguage 类中实现 __call__() 方法，使的 clangs 实例对象变为了可调用对象（def __init__不是必需的）

Python 中，凡是可以将 () 直接应用到自身并执行，都称为可调用对象。

# 可调用对象---其余

1、用户自定义的函数：

使用def语句或者lambda表达式创建的函数。

2、内置函数：

使用C语言实现的函数，如len、sum或者time.strftime

3、内置方法：

使用C语言实现的方法，如dict.get()

4、类方法：

在类的定义体中定义的函数

5、类：

在调用类时会运行类的__new__方法创建一个实例，然后运行__init__方法，初始化实例，最后把实例返回给调用方。Python中没有new运算符，所以调用类相当于调用函数。

6 、生成器函数：

使用yield关键字的函数或方法。调用生成器函数返回的是生成器对象。

# 构造方法

# 构造方法

- __new__()和__init__()
  - __init__()是初始化方法，是当实例对象创建完成后被调用的，然后设置对象属性的一些初始值
  - __new__()是**构造方法**，在实例创建之前被调用的，因为它的任务就是**创建实例然后返回该实例**
  - __new__()在__init__()之前被调用，__new__()的返回值（实例）将传递给__init__()方法的第一个参数，然后__init__()给这个实例设置参数

# 构造方法

**__new__类级别的方法，__init__方法：实例级别的方法**

1.有一个参数self,该self参数就是__new__()返回的实例；
2.__init__()在__new()的基础上完成初始化动作，不需要返回值；
3.若__new__()没有正确返回当前类cls的实例，那__init__()将不会被调用
4.创建的每个实例都有自己的属性，方便类中的实例方法调用

```
class B():
    def __new__(cls):
        print ("__new__方法被执行")
    def __init__(self):
        print ("__init__方法被执行")
b=B()
```

结果：
__new__方法被执行

实例调用构造函数为对象分配空间

```
class B():
    def __new__(cls):
        print (“__new__方法被执行”)
        return super(B,cls).__new__(cls)
    def __init__(self):
        print (“__init__方法被执行”)
b=B()
```

结果：
__new__方法被执行
__init__方法被执行

# 构造方法

In [12]:
```
class A:
    def  __new__(cls,name):
        result=super().__new__(cls) #实例调用构造函数
        print(result)
        result.name='Bob'
        return result
    def __init__(self,age):
        print('__init__ is called') #没有返回实例后不会进行初始化
        self.age=age
a=A(10)
print(a)
```

```
<__main__.A object at 0x00000275BA835AC0>
__init__ is called
<__main__.A object at 0x00000275BA835AC0>
```

In [139]:
```
print(a.name)
print(a.age)

```

```
Bob
10
```

In [11]:
```
class A:
    def  __new__(cls,name):
        result=super().__new__(cls) #实例调用构造函数
        print(result)
        result.name='Bob'
        #return result
    def __init__(self,age):
        print('__init__ is called') #没有返回实例后不会进行初始化
        self.age=age
a=A(10)
print(a)
```

```
<__main__.A object at 0x00000275BA8359D0>
None
```

__new__(cls)调用需要一个默认参数 cls，就是将要返回的这个实例所属的类

__new__:**根据一个类生产(构造)一个具体商品**

__init__:将这个商品(实例)进行加工(赋予属性)

Python中的**super()**方法设计目的是用来解决多重继承时父类的查找问题，所以在单重继承中用不用 super 都没关系；但是，**使用 super() 是一个好的习惯。**一般我们在子类中需要调用父类的方法时才会这么用。
**super()**的好处**就是可以避免直接使用父类的名字.主要用于多重继承**

```
class A:
    def m(self):
        print('A')
class B:
    def m(self):
        print('B')
class C(A):
    def m(self):
        print('C')
    super().m()
C().m()
```

改变子类继承的父类（由A改为B），你只需要修改一行代码（class C(A): -> class C(B)）即可，而不需要在class C的大量代码中去查找、修改基类名，另外一方面代码的可移植性和重用性也更高。

super()，它本身就是super(A, B)的简写，其中A是代码发生的类，B是代码发生的函数的第一个参数；因此在特定情况下，super().__new__(cls)扩展到super(CarModel, cls).__new__(cls)。

```
class P(object):
    def __init__(self):
        print('calling __init__ of P')

class C(P):
    def __init__(self):
        return super(C,self).__init__()
```

子类的 C 里面调用父类 P 的 __init__() 方法（绑定的），就给 super() 函数传了 C 和 C 的实例 self 两个参数表示绑定到 self 上的 P 类。这里也可以用 __class__ 、self.__class__ 或者 type(self) 来代替直接给出当前类的名字 C。
在 Python3 里可以无参数使用 super() 函数，效果和传两个参数一样

# 构造方法

在Python中new方法与init方法类似，但是如果两个都存在那么new先执行。

在基础类object中，new被定义成了一个静态方法，并且需要传递一个参数cls。
cls表示需要实例化的类，此参数在实例化时由Python解析器自动提供。

new()是在新式类中新出现的方法，它作用在构造方法init()建造实例之前，可以这么理解，在Python 中存在于类里面的构造方法init()负责将类的实例化，而在init()调用之前，new()决定是否要使用该init()方法，因为new()可以调用其他类的构造方法或者直接返回别的对象来作为本类 的实例。

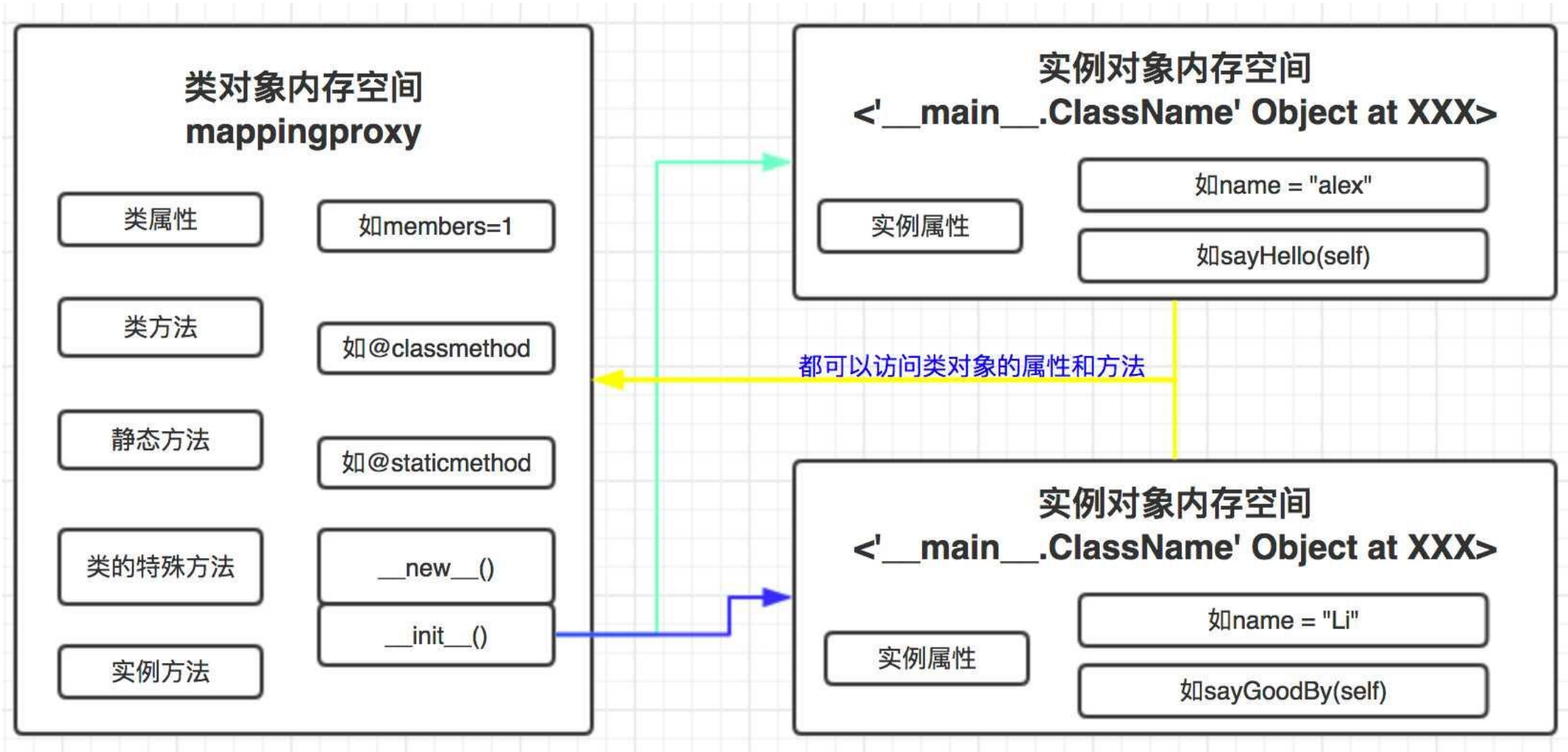
# 构造方法

```
class Person(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __new__(cls, name, age):
        if 0 < age < 150:
            #return object.__new__(cls)
            return super(Person, cls).__new__(cls)
        else:
            return None

    def __str__(self):
        return '{0}({1})'.format(self.__class__.__name__,
self.__dict__)

print(Person('Tom', 10))
print(Person('Mike', 200))
```

```
class Person(object):

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __new__(cls, name, age):
        if 0 < age < 150:
            #return object.__new__(cls)
            return super(Person, cls).__new__(cls)
        else:
            return None

    def __str__(self):
        return '{0}({1})'.format(self.__class__.__name__, self.__dict__)


print(Person('Tom', 10))
print(Person('Mike', 200))
```

```
Person({'name': 'Tom', 'age': 10})
None
```

作用：定制满足一定条件的实例！


类对象内存空间
mappingproxy
类属性
如members=1
类方法
如@classmethod
静态方法
如@staticmethod
类的特殊方法
__new__()
__init__()
实例方法
实例对象内存空间
<'__main__.ClassName' Object at XXX>
实例属性
如name = "alex"
如sayHello(self)
都可以访问类对象的属性和方法
实例对象内存空间
<'__main__.ClassName' Object at XXX>
实例属性
如name = "Li"
如sayGoodBy(self)

# 单例模式

- 单例模式：
  - 一种常用的软件设计模式，确保某一个类只有一个实例存在(打印发票)
- 单例模式的要点：
  - 某个类只能有一个实例
  - 必须自行创建这个实例
  - 必须自行向整个系统提供这个实例
- 单例模式的优点
  - 阻止其他对象实例化其自己的单例对象的副本，从而确保所有对象都访问唯一实例
  - 由于类控制了实例化过程，类可以灵活更改实例化过程
- 单例模式的缺点
  - 使用单例对象（尤其在类库中定义的对象）时，不能使用new关键字实例化对象
    - 因为可能无法访问库源代码，开发人员可能会意外发现自己无法直接实例化此类
  - 虽然数量很少，但如果每次对象请求引用时都要检查是否存在类的实例，将仍然需要开销

# 单例模式

- Python中，单例模式有以下几种实现方式
  - 使用模块
  - 使用 __new__()
  - 使用装饰器（decorator）
  - 使用元类（metaclass）

# 单例模式

- 使用__new__()实现单例模式
  - 当实例化一个对象时，先执行类的__new__()方法实例化对象
  - 然后执行类的__init__()方法，对这个对象进行初始化，从而实现单例模式

具体实现方法：实现__new__方法，然后将类的一个实例绑定到类变量_instance上；如果cls._instance为None，则说明该类还没有被实例化过，new一个该类的实例，并返回；如果cls._instance不为None，直接返回_instance

```
class Singleton(object):
    def __new__(cls, *args, **kwargs):
        if not hasattr(cls,'_instance'):
            orig=super(Singleton,cls)
            cls._instance=orig.__new__(cls,*args,**kwargs)
        return cls._instance

class MyClass(Singleton):
    a=1

one=MyClass()
two=MyClass()

# one和two完全相同，可以用id()，==，is检查
print(one.a)     # 1
print(id(one))   # 1609964673728
print(id(two))   # 1609964673728
print(one == two)    # True
print(one is two)    # True
```

```
1
1609964673728
1609964673728
True
True
```

hasattr(object, name)： object -- 对象，name -- 字符串，属性名。
如果对象有该属性返回 True，否则返回 False。

内容：保证一个类只有一个实例，并提供一个访问它的全局访问点。
适用场景：当类只能有一个实例而且客户可以从一个众所周知的访问点访问它时
优点：
对唯一实例的受控访问
单例相当于全局变量，但防止了命名空间被污染

# 元类

# 元类

- 元类概念
  - 元类（metaclass）是类的类，是类的模板
- 元类作用
  - 元类是用来控制如何创建类的，正如类是创建对象的模板一样，而元类的主要目的是为了控制类的创建行为
  - type是Python的一个内建元类，用来直接控制生成类，Python中任何class定义的类其实都是type类实例化的对象

类是实例对象的模板，元类是类的模板

# 使用元类实现单例模式

- 元类可以控制类的创建过程，它主要做三件事：
  - 拦截类的创建
  - 修改类的定义
  - 返回修改后的类

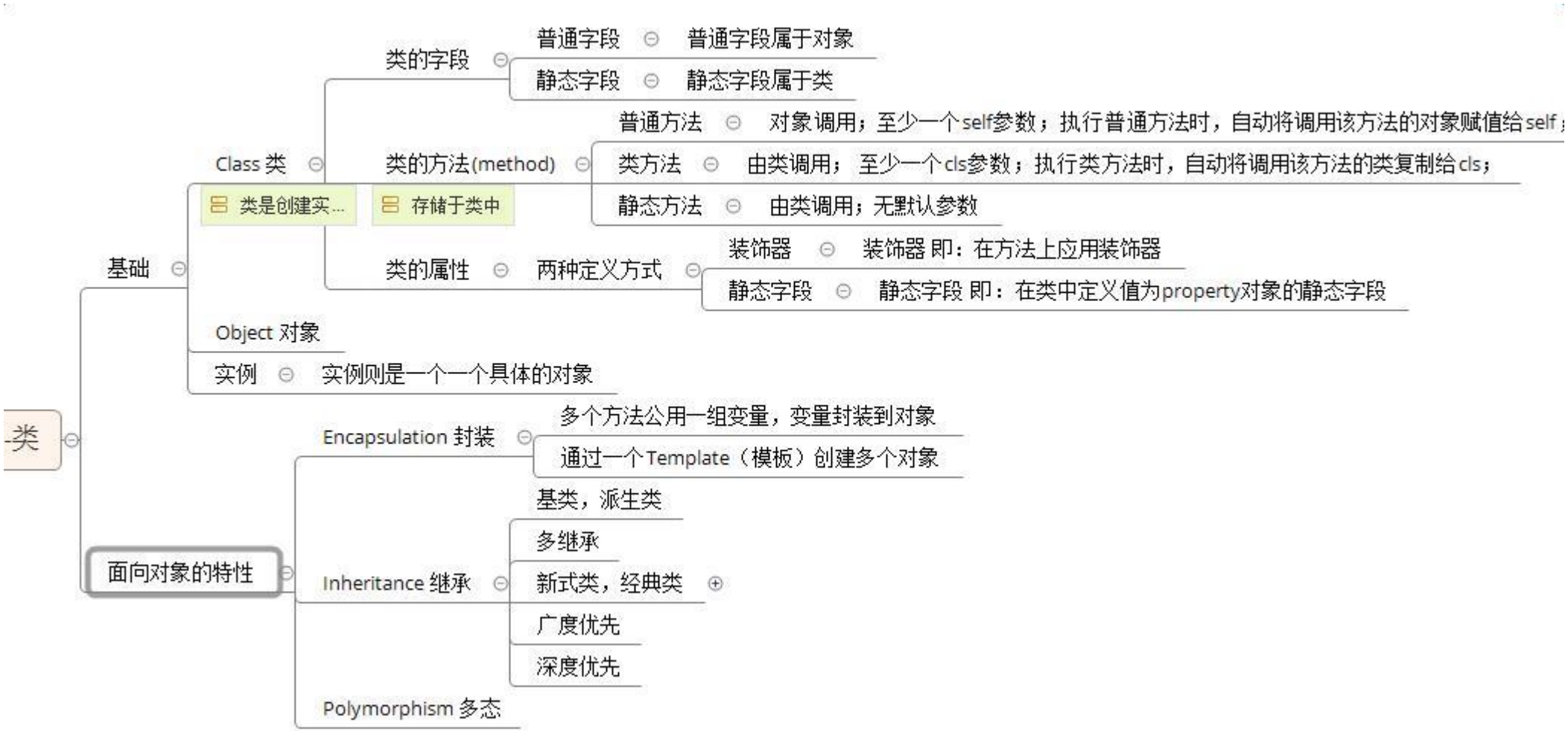
# 使用元类实现单例模式

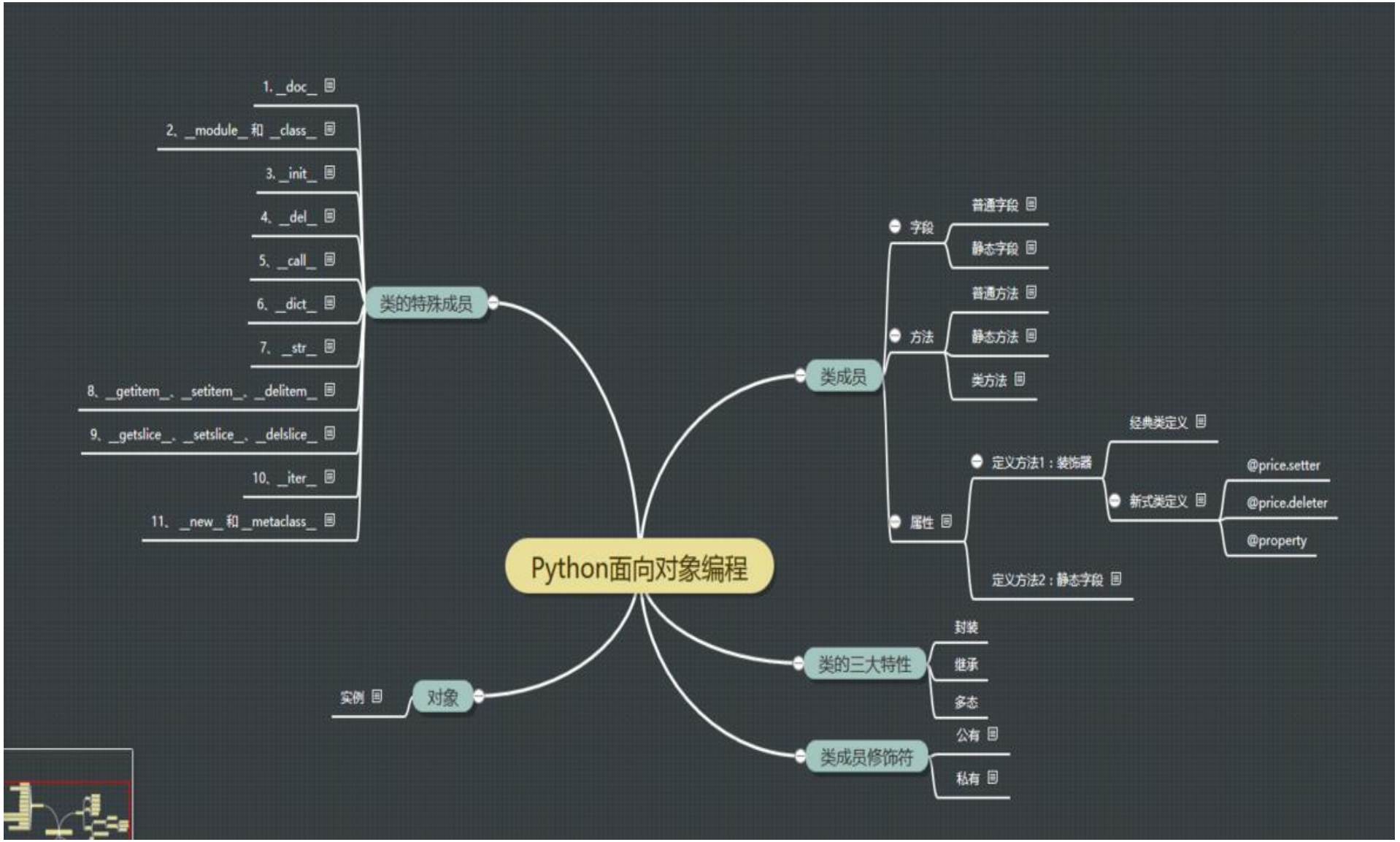
- 实现单例模式
  - 类本身就是元类的实例，当在元类中定义__call__()的函数时，会改变类的实例化行为
  - 可利用元类和__call__()，实现单例模式，元类__call__方法返回的是实例，和类的__new__方法返回的一样。__call__()中的cls代表以此元类所创建的类

In [21]:

```
class Singleton(type):#定义元类
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]
class MyClass(metaclass = Singleton):#使用元类创建类MyClass
    pass

Th1=MyClass()
Th2=MyClass()
print(id(Th1),id(Th2))
print(Th1==Th2)

```

```
2704663600000 2704663600000
True
```


类
基础
Class 类
类的字段
普通字段
普通字段属于对象
静态字段
静态字段属于类
类的方法(method)
普通方法
对象调用；至少一个self参数；执行普通方法时，自动将调用该方法的对象赋值给self；
类方法
由类调用； 至少一个cls参数；执行类方法时，自动将调用该方法的类复制给cls；
静态方法
由类调用；无默认参数
类是创建实...
存储于类中
类的属性
两种定义方式
装饰器
装饰器 即：在方法上应用装饰器
静态字段
静态字段 即：在类中定义值为property对象的静态字段
Object 对象
实例
实例则是一个一个具体的对象
面向对象的特性
Encapsulation 封装
多个方法公用一组变量，变量封装到对象
通过一个Template（模板）创建多个对象
Inheritance 继承
基类，派生类
多继承
新式类，经典类
广度优先
深度优先
Polymorphism 多态


Python面向对象编程
类的特殊成员
1、__doc__
2、__module__和__class__
3、__init__
4、__del__
5、__call__
6、__dict__
7、__str__
8、__getitem__、__setitem__、__delitem__
9、__getslice__、__setslice__、__delslice__
10、__iter__
11、__new__和__metaclass__
类成员
字段
普通字段
静态字段
方法
普通方法
静态方法
类方法
属性
定义方法1：装饰器
经典类定义
新式类定义
@price.setter
@price.deleter
@property
定义方法2：静态字段
类的三大特性
封装
继承
多态
类成员修饰符
公有
私有
对象
实例

# 迭代器与生成器---程序执行的资源优化

## 练习题

1 为二次方程式ax2+bx+c=0设计一个名为Equation的类，这个类包括：

1）代表3个系数的成员变量a、b、c；

2）一个参数为a、b、c的构造方法；

3）一个名为getDiscriminant()的方法返回判别式的值；

4）两个分别名为getRoot1()和getRoot2()的方法返回方程的两个根，如果判别式为负，这些方法返回None。

2 定义代表二维坐标系上某个点的Point类（包括x、y两个属性），为该类提供一个方法用于计算两个Point之间的距离，再提供一个方法用于判断三个Point组成的三角形是钝角、锐角还是直角三角形。

3.设计一个三维向量类，并实现向量的加法、减法以及向量与标量的乘法和除法运算。

下次课**月**日提交纸质和电子版(提交*.py文件，包含在文件夹中，文件夹名为学号+姓名)---课代表***负责汇总

# 第六章 结束