

## 面向对象基础（下）

### 复习

#### 析构方法

当一个对象被删除或者被销毁时，python 解释器也会默认调用一个方法，这个方法为 **del()** 方法

#### 单继承

子类在继承的时候，在定义类时，小括号()中为父类的名字 父类的属性、方法，会被继承给子类。

#### 多继承

子类可以继承多个父类，在小括号()中用逗号隔开

#### 继承的传递

子类可以继承父类的父类的方法

#### 重写和调用父类方法

重写父类方法后，调用的是子类的方法。

在重写的方法里面还可以调用父类方法。

#### 多态

定义时的类型和运行时的类型不一样，此时就成为多态。

#### 类属性和实例属性

类属性：就是类对象所拥有的属性

实例属性：实例对象所拥有的属性

#### 类方法和静态方法

用 **@classmethod** 来表示类方法

用 **@staticmethod** 来表示静态方法

## 导入

前面我们已经学习了面向对象编程的类、对象、类之间的关系等，接下来我们要深入学习如何具体控制属性、方法来满足需要，完成功能。

## 目录

- 1、私有化属性
- 2、私有化方法
- 3、Property 属性
- 4、\_\_new\_\_方法
- 5、单例模式
- 6、错误与异常处理
- 7、Python 动态添加属性和方法
- 8、\_\_slots\_\_方法

## 目标

- 1、通过声明私有化属性、方法，保护和控制数据
- 2、通过 property 属性的使用，即控制好数据又方便访问
- 3、明确\_\_new\_\_方法的作用和用法
- 4、通过单例模式，控制实例个数
- 5、使用异常处理机制，处理异常，提高代码健壮性
- 6、利用动态语言特点，动态添加属性和方法
- 7、利用\_\_slots\_\_属性控制可动态的属性

# 一、私有化属性

## 1、概述

前面学习面向对象过程中，修改类属性都是直接通过类名修改的。如果有些重要属性不想让别人随便修改，或者防止意外修改，该怎么办？

为了更好的保存属性安全，即不能随意修改，将属性定义为私有属性，添加一个可调用的方法去访问。

## 2、语法

两个下划线开头，声明该属性为私有，不能在类的外部被使用或直接访问。

示例：

```
class Person(object):  
    __age = 18 # 定义一个私有化属性，属性名字前面加两个下划线
```

## 3、特性

(1) 私有化属性不能在类外面访问。

```
class Person(object):  
    __age = 18 # 定义一个私有化属性，属性名字前面加两个下划线  
  
xiaoming = Person()  
print(Person.__age) # 类对象无法访问私有类属性  
print(xiaoming.__age) # 实例对象无法访问私有属性
```

(2) 私有化属性可以在类里面访问，修改。

```
class Person(object):  
    __age = 18 # 定义一个私有化属性，属性名字前面加两个下划线  
  
    def get_age(self): # 访问私有类属性  
        return Person.__age  
  
    def set_age(self, age): # 修改私有类属性  
        Person.__age = age  
  
xiaoming = Person()  
print(xiaoming.get_age())
```

```
xiaoming.set_age(20)
print(xiaoming.get_age())
# 输出
# 18
# 20
```

(3) 子类不能继承私有化属性。

```
class Person(object):

    def __init__(self):
        self.__age = 18    # 定义一个私有实例属性
    def get_age(self):
        return self.__age    # 访问私有实例属性
    def set_age(self, age):
        self.__age = age    # 修改私有实例属性

class China(Person):
    def get_age(self):
        print(self.__age)    # 尝试访问父类的私有实例属性

xiaoming = China()
xiaoming.get_age()    # 报异常无法访问
```

## 二、私有化方法

### 1、概述

私有化方法跟私有化属性概念一样，有些重要的方法，不允许外部调用，防止子类意外重写，把普通的方法设置成私有化方法。

### 2、语法

私有化方法，即在方法名前面加两个下划线。

示例：

```
class A(object):
    ## 在方法前面加两个 __ 下划线，变成私有化方法
    def __myname(self):
```

```

        print('xiaoming')
#普通方法
    def myname(self):
        print(xiaoming)

a = A()
a.myname() # 正常调用
a.__myname() # 调用私有化方法，报错

```

### 3、特性

私有化方法一般是类内部调用，子类不能继承，外部不能调用。

#### 单下划线、双下划线、头尾双下划线说明

- `_xxx` 前面加一个下划线，以单下划线开头的表示的是 **protected** 类型的变量，即保护类型只能允许其本身与子类进行访问，不能使用 `from xxx import *` 的方式导入。
- `__xxx__` 前后两个下滑线，魔法方法，一般是 **python** 自有，开发者不要创建这类型的方法。
- `xxx_` 后面单下滑线，避免属性名与 **python** 关键字冲突。

## 三、Property 属性

### 1、概述

上节我们讲了访问私有变量的话，一般写两个方法一个访问，一个修改，由方法去控制访问。

```

class Person(object):
    def __init__(self):
        self.__age = 18 # 定义一个私有化属性，属性名字前加连个 __ 下滑线

    def get_age(self): # 访问私有类属性
        return self.__age

    def set_age(self, age): # 修改私有属性
        if age < 0:
            print('年龄不能小于 0')

```

```

        else:
            self.__age = age

xiaoming = Person()
xiaoming.set_age(18)
print(xiaoming.get_age())

```

这样给调用者的感觉就是调用了方法，并不是访问属性。我们怎么做到让调用者直接以访问属性的方式，而且我们又能控制的方式提供给调用者？

Python 中有一个被称为**属性函数(property)**的小概念，它可以做一些有用的事情。

## 2、实现方式

(1) 类属性,即在类中定义值为 **property** 对象的类属性

给 **age** 属性设置值时，会自动调用 **setage** 方法，获取 **age** 属性值时，会自动调用 **getage** 方法。

```

class Person(object):
    def __init__(self):
        self.__age = 18  # 定义一个私有化属性，属性名字前加连个 __ 下滑线

    def get_age(self):  # 访问私有实例属性
        return self.__age

    def set_age(self, age):  # 修改私有实例属性
        if age < 0:
            print('年龄不能小于 0')
        else:
            self.__age = age

    age = property(get_age, set_age)  # 定义一个属性，当对这个 age 设置值时调用
    set_age,
    # 当获取值时调用 get_age
    # 注意：必须是以 get, set 开头的方法名，才能被调用

xiaoming = Person()
xiaoming.age = 15
print(xiaoming.age)

```

(2) 装饰器，即在方法上使用装饰器

```
class Person(object):
    def __init__(self):
        self.__age = 18 # 定义一个私有化属性，属性名字前加连个 __ 下滑线

    @property # 使用装饰器对 age 进行装饰，提供一个 getter 方法
    def age(self): # 访问私有实例属性
        return self.__age

    @age.setter # 使用装饰器进行装饰，提供一个 setter 方法
    def age(self, age): # 修改私有实例属性
        if age < 0:
            print('年龄不能小于 0')
        else:
            self.__age = age

xiaoming = Person()
xiaoming.age = 15
print(xiaoming.age)
```

## 四、\_\_new\_\_方法

### 1、概述

`__new__`方法的作用是，创建并返回一个实例对象，如果`__new__`只调用了一次，就会得到一个对象。继承自 `object` 的新式类才有 `new` 这一魔法方法。

### 2、使用方式

示例 1:

```
class A(object):

    def __init__(self):
        print("__init__执行了")

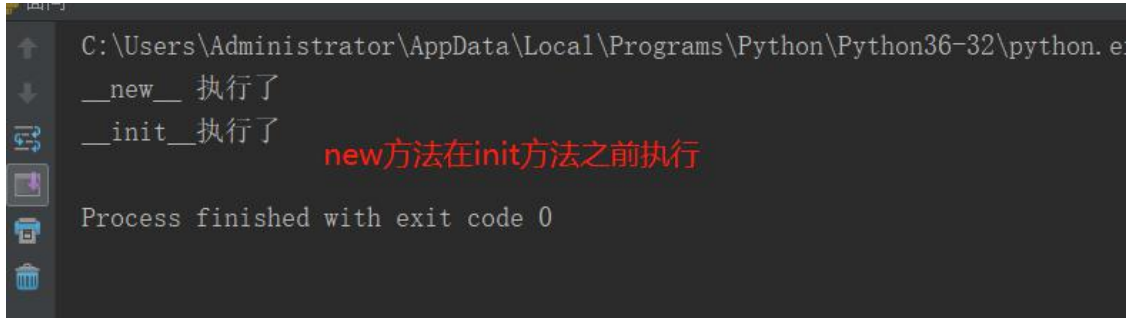
    def __new__(cls, *args, **kwargs):

        print("__new__ 执行了")

        return object.__new__(cls) # 调用父类的 new 方法
```

```
a = A()
```

输出结果:



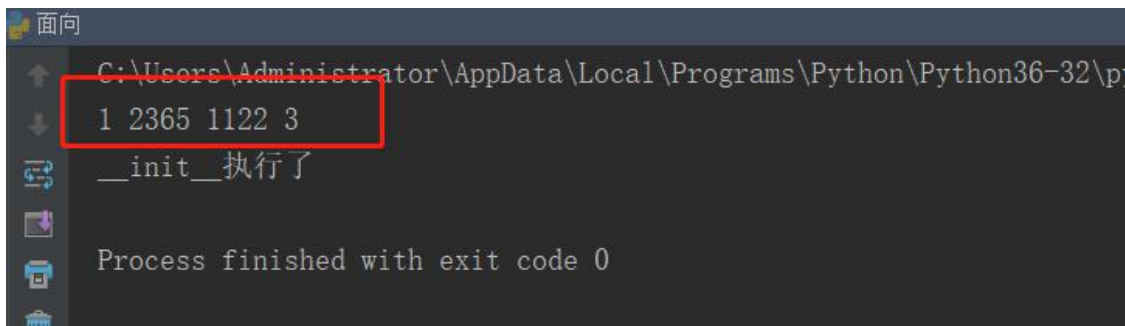
```
C:\Users\Administrator\AppData\Local\Programs\Python\Python36-32\python.exe  
__new__ 执行了  
__init__ 执行了  
new方法在init方法之前执行  
Process finished with exit code 0
```

示例 2:

```
class A(object):  
    def __init__(self, a, b, c, k):  
  
        print(a, b, c, k)  
  
        print("__init__执行了")  
  
    def __new__(cls, *args, **kwargs):  
  
        return object.__new__(cls)  # 调用父类的 new 方法  
  
a = A(1, 2365, 1122, k=3)
```

输出结果:





```
C:\Users\Administrator\AppData\Local\Programs\Python\Python36-32\python.exe
1 2365 1122 3
__init__执行了
Process finished with exit code 0
```

注意：

- `__new__` 是在一个对象实例化的时候所调用的第一个方法
- `__new__` 至少必须要有一个参数 `cls`，代表要实例化的类，此参数在实例化时由 Python 解释器自动提供，其他的参数是用来直接传递给 `__init__` 方法
- `__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实例对象来作为本类的实例，如果 `__new__` 没有返回实例对象，则 `__init__` 不会被调用
- 在 `__new__` 方法中，不能调用自己的 `__new__` 方法，即：`return cls.__new__(cls)`，否则会报错（`RecursionError: maximum recursion depth exceeded: 超过最大递归深度`）

## 五、单例模式

### 1、概述

单例模式是常用设计模式的一种，单例就比如我们打开电脑的回收站，在系统中只能打开一个回收站，也就是说这个整个系统中只有一个实例，重复打开也是使用这个实例。

简单的说就是不管创建多少次对象，类返回的对象都是最初创建的，不会再新建其他对象。

### 2、实现步骤

（1）利用类属性保存初次创建的实例对象，第二次实例化的时候判断类属性是否保存有实例对象，如果有就返回类属性保存的，如果没有就调用父类 `__new__` 方法创建新的实例对象。

```

class SingleCase(object):
    __instance = None # 保存实例对象

    def __init__(self, name, age):
        print(name, age)

    def __new__(cls, *args, **kwargs):

        # 如果类属性 __instance 的值为 None, 那么新建一个对象
        # 如果类属性值不为 None 返回 __instance 保存的对象
        if not cls.__instance:
            cls.__instance = super(SingleCase, cls).__new__(cls) # 调用父类
            new__方法生成一个实例对象
        return cls.__instance
    else:
        return cls.__instance

obj1 = SingleCase('xiaoming', 18)

obj2 = SingleCase('xiaoming', 118)

print(id(obj1)) # id 相等, 说明实例化两次对象, 实际上都是同一个对象
print(id(obj2))

```

(2) 只执行一次 **init** 方法, 通过类变量进行标记控制

```

class SingleCase(object):
    __instance = None # 保存实例对象
    __isinit = True # 首次执行 init 方法标记

    def __init__(self, name, age):
        if SingleCase.__isinit:
            self.name = name
            self.age = age
            SingleCase.__isinit = False

    def __new__(cls, *args, **kwargs):

        # 如果类属性 __instance 的值为 None, 那么新建一个对象
        # 如果类属性值不为 None 返回 __instance 保存的对象
        if not cls.__instance:
            cls.__instance = super(SingleCase, cls).__new__(cls) # 调用父类
            new__方法生成一个实例对象
        return cls.__instance
    else:

```

```
        return cls.__instance

obj1 = SingleCase('xiaoming',18)

obj2 = SingleCase('luban',118)

print(id(obj1))
print(id(obj2))
print(obj1.age)
print(obj2.age)    # 年龄都输出 18。说明 init 值初始化了一次
```

## 六、错误与异常处理

### 1、概述

有时候代码写错了，执行程序的时候，执行到错误代码的时候，程序直接终止报错，这是因为 Python 检测到一个错误时，解释器就无法继续执行了，出现了错误的提示，这就是"异常"。

示例：变量 b 没有定义，直接打印变量 b，会报异常。

```
>>> print(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>>
```

### 2、语法格式

```
try:
    可能出现错误的代码块

except:
    出错之后执行的代码块

else:
    没有出错的代码块

finally:
    不管有没有出错都执行的代码块
```

### 3、try ... except 语句

将可能出错的代码放到 **try** 里面，**except** 可以指定类型捕获异常。**except** 里面的代码是捕获到异常时执行,将错误捕获，这样程序就不会因为一段代码包异常而导致整个程序崩溃。

示例 1：捕获 ZeroDivisionError 异常

```
# 0 不能做除数，1/0 必然报错
1/0

try:
    1/0 # 将可能会报错的代码放到 try 里面
except:
    pass

try:
    1/0      # 将可能会报错的代码放到 try 里面
except ZeroDivisionError as e:    # 捕获异常，ZeroDivisionError 异常类型
    print(e)
```

示例 2：捕获所有类型的异常

```
try:
    print(a)
except Exception as e: # Exception 可以捕获任何类型的异常
    print(e)
```

### 4、try ... except ... else 语句

没有捕获到异常时才执行 **else** 语句

示例：

```
try:
    print('-----test-----')
except Exception as e: # Exception 可以捕获任何类型的异常
    print(e)
else:
    print('haha --- 没有捕获到异常') # 没有捕获到异常，将执行 else 里面代码，
    否则不执行

try:
```

```

    print('-----test-----')
    1 / 0
except Exception as e: # Exception 可以捕获任何类型的异常
    print(e)
else:
    print('haha --- 没有捕获到异常') # 捕获到异常不执行 else 的代码

```

## 5、try ... except ... finally 语句

不管是否捕获到异常都会执行 finally 语句。

一般像数据库连接，打开文件，不管中途出现了什么情况，在最后都要关闭连接，关闭文件。这样一般将关闭方法放到 finally 里面。

示例：

```

try:
    print('-----test-----')
except Exception as e: # Exception 可以捕获任何类型的异常
    print(e)
else:
    print('haha --- 没有捕获到异常') # 没有捕获到异常，将执行 else 里面代码，
    否则不执行
finally:
    print('不管有没有捕获到异常，finally 都是执行的')

try:
    print('-----test-----') # try 里面有异常
    1 / 0
except Exception as e: # Exception 可以捕获任何类型的异常
    print(e)
else:
    print('haha --- 没有捕获到异常') # 捕获到异常，执行 else 的代码
finally:
    print('不管有没有捕获到异常，finally 都是执行的')

```

## 6、try 嵌套

```

try:
    f = open('test.txt', 'r') # 打开一个文件
    try:
        while True:
            content = f.readline()

            if len(content) == 0:

```

```

        break
    time.sleep(2)
    print(content)

except Exception as e:
    print(e)      # 打印捕获的异常
    #如果在读取文件的过程中，产生了异常，那么就会捕获到
    #比如 按下了 ctrl+c
finally:
    f.close()
    print('关闭文件')
except Exception as e:
    print(e,'文件不存在')
```

## 7、异常传递

如果多个函数嵌套调用，内层函数异常，异常会往外部传递，直到异常被抛出，或被处理。

示例 1:

```

def a():
    print('执行 a 函数')
    1/0    # 制造一个异常
    print('a 函数执行完成')

def b():
    print('执行 b 函数')
    a()    # 调用 a 函数
    print('b 函数执行完成')

def c():
    print('执行 c 函数')
    b()    # 调用 b 函数
    print('c 函数执行完成')

c()
```

输出结果:

```
图同
C:\Users\Administrator\AppData\Local\Programs\Python\Python36-32\python.exe C:/Users/Administrator
Traceback (most recent call last):
  执行c函数
    File "C:/Users/Administrator/Desktop/python绝技(运用python成为顶级黑客)/python_stunts/code/tcp连
  执行b函数
    c()
  执行a函数
    File "C:/Users/Administrator/Desktop/python绝技(运用python成为顶级黑客)/python_stunts/code/tcp连
      b() # 调用b函数
    File "C:/Users/Administrator/Desktop/python绝技(运用python成为顶级黑客)/python_stunts/code/tcp连
      a() # 调用a函数
    File "C:/Users/Administrator/Desktop/python绝技(运用python成为顶级黑客)/python_stunts/code/tcp连
      1/0 # 制造一个异常
ZeroDivisionError: division by zero
Process finished with exit code 1
```

示例 2:

```
def a():
    print('执行 a 函数')
    1/0 # 制造一个异常
    print('a 函数执行完成')

def b():
    print('执行 b 函数')
    a() # 调用 a 函数
    print('b 函数执行完成')

def c():
    print('执行 c 函数')
    try:
        b() # 调用 b 函数
    except Exception as e:
        print(e)
    print('c 函数执行完成')

c()
```

输出结果:

```
F:\workspace\pycharm\python_day09\venv\Scripts\python.exe F:/workspace/pycharm/python_day09/TestException.py
执行c函数
执行b函数
执行a函数
division by zero
c函数执行完成

Process finished with exit code 0
```

## 8、自定义异常

自定义异常，都要直接或间接继承 `Error` 或 `Exception` 类。

由开发者主动抛出自定义异常，在 `python` 中使用 `raise` 关键字，

```
# 自定义一个异常类
class LeNumExcept(Exception):  # 自定义异常类需要继承 Exception

    def __str__(self):
        return '[error:]你输入的数字小于 0，请出入大于 0 的数字'

try:
    num = int(input('请输入一个数字: '))
    if num < 0:
        # raise 关键字抛出异常
        raise LeNumExcept()
except LeNumExcept as e:
    print(e)  # 捕获异常
else:
    print('没有异常')
```

## 9、抛出捕获到的异常

在开发中有，有时候会有这样一种功能，由调用者决定是否需要做异常处理，如不处理，可继续抛出。

示例：实例化对象时传入一个参数，判断捕获异常后是否做处理，不做处理用 `raise` 关键字重新抛出异常。

```
class A(object):
    def __init__(self, switch):
        self.switch = switch
```

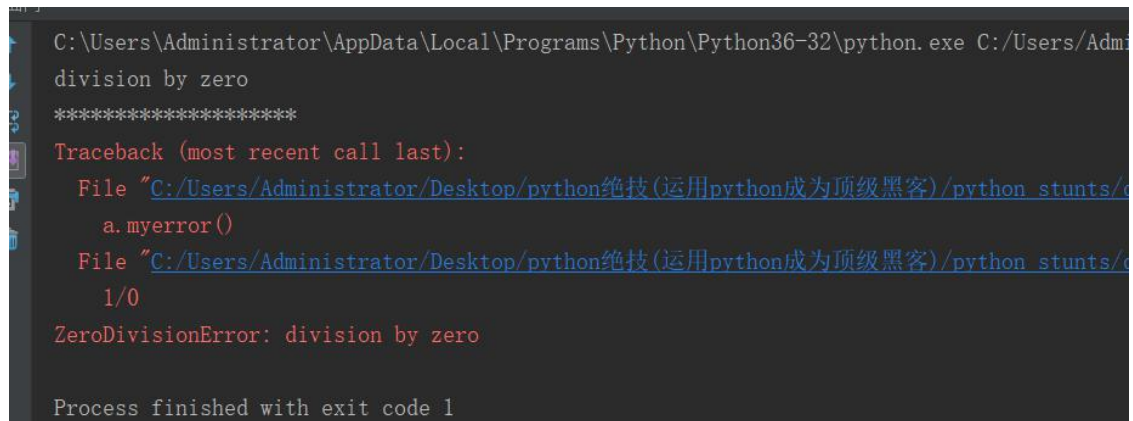


```

def myerror(self):
    try:
        1/0
    except Exception as e:
        if self.switch:
            print(e)
        else:
            raise # 重新抛出异常
a = A(True) # 做异常处理
a.myerror()
print('*'*20)
a = A(False) # 不错异常处理
a.myerror()

```

输出结果：



```

C:\Users\Administrator\AppData\Local\Programs\Python\Python36-32\python.exe C:/Users/Administrator/Desktop/python绝技(运用python成为顶级黑客)/python_stunts/c
division by zero
*****
Traceback (most recent call last):
  File "C:/Users/Administrator/Desktop/python绝技(运用python成为顶级黑客)/python_stunts/c
    a.myerror()
  File "C:/Users/Administrator/Desktop/python绝技(运用python成为顶级黑客)/python_stunts/c
    1/0
ZeroDivisionError: division by zero

Process finished with exit code 1

```

## 七、动态添加属性和方法

### 1、概述

动态语言：运行时可以改变其结构的语言，例如新的函数、对象、甚至代码可以被引进，已有的函数可以被删除或是其他结构上的变化。如 **php,JavaScript,python** 都是动态语言，**C, C#, java** 是静态语言。所以 **python** 可以在程序运行过程中添加属性和方法。

## 2、动态添加属性

### (1) 运行中给对象添加属性

```
>>> class Animal(object):
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
>>> # 定义了两个初始属性 name 和 age，但是没有颜色，我想要添加颜色又不能去修改类
...
>>> cat = Animal('小白', 5)
>>> # 动态绑定 color 属性
...
>>> cat.color = '白色'
>>> print(cat.color)
白色
>>>
```

定义类的时候并没有 `color` 属性，实例化之后还可以给实例对象绑定一个属性，这只有动态语言才可以这么干。

### (2) 运行中给类添加属性

```
>>> class Animal(object):
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
>>> # 定义了两个初始属性 name 和 age，但是没有颜色，我想要添加颜色又不能去修改类
...
>>> cat = Animal('小白', 5)
>>> # 动态绑定 colour 属性
...
>>> cat.color = '白色'
>>> print(cat.color)
白色
>>>
>>>
>>>
>>>
>>>
>>>
>>> Animal.foo = 4 # 添加类属性
>>> cat.foo
4
>>>
```

### 3、动态添加方法

#### (1) 动态添加实例方法

动态添加实例方法需要使用 `types`

```
>>> import types
>>> cat.run = types.MethodType(run, cat) # 利用 types 方法绑定实例属性
>>> cat.run()
一只 5 岁的叫小白的猫咪在跑
>>>
```

#### (2) 给类绑定类方法和静态方法

使用方式：类名.方法名 = xxxx

```
>>>
>>> # 定义一个类方法
... @classmethod
... def eat(cls):
...     print('吃东西')
...
>>> # 定义一个静态方法
... @staticmethod
... def drink():
...     print('喝水')
...
>>> # 给 Animal 类绑定类方法
... Animal.eat = eat
>>> # 调用类方法
... Animal.eat()
吃东西
>>>
>>> # 给 Animal 类绑定静态方法
... Animal.drink = drink
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'Animal' is not defined
>>> Animal.drink = drink
>>> # 调用静态方法
... Animal.drink()
喝水
>>>
```

## 八、\_\_slots\_\_属性

### 1、概述

python 是动态语言，在运行的时候可以动态添加属性。如果要限制在运行的时候给类添加属性，Python 允许在定义 class 的时候，定义一个特殊的\_\_slots\_\_变量，来限制该 class 实例能添加的属性。

只有在\_\_slots\_\_变量中的属性才能被添加，没有在\_\_slots\_\_变量中的属性添加失败。可以防止其他人在调用类的时候胡乱添加属性或方法。\_\_slots\_\_属性子类不会继承，只有在当前类中有效。

### 2、使用方式

```
>>> class A(object):
...     __slots__ = ('name', 'age')
...
>>> a = A()
>>> a.name = '旺财'
>>> a.age = 5
>>> print(a.name)
旺财
>>> print(a.age)
5
>>> a.test = 'sds' # 不再 slots 属性里，无法添加属性
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'test'
>>>
```

## 小结

### 私有化属性

两个下划线开头，声明该属性为私有，不能在类的外部被使用或直接访问。

### 私有化方法

私有化方法，即在方法名前面加两个下划线。

### Property 属性

类属性,即在类中定义值为 property 对象的类属性

装饰器，即在方法上使用装饰器

## **`__new__`方法**

`__new__`方法的作用是，创建并返回一个实例对象

## **单例模式**

不管创建多少次对象，类返回的对象都是最初创建的，不会再新建其他对象。

## **错误与异常处理**

**try:**

可能出现错误的代码块

**except:**

出错之后执行的代码块

**else:**

没有出错的代码块

**finally:**

不管有没有出错都执行的代码块

## **Python 动态添加属性和方法**

在程序运行过程中添加属性和方法

## **`__slots__`方法**

在定义 class 的时候，定义一个特殊的`__slots__`变量，来限制该 class 实例能添加的属性

## **课后作业**

### **课后问答题**

- 1、Python 中 `new` 方法作用是什么？
- 2、什么是单例模式？单例模式适用于什么场景？
- 3、私有化方法与私有化属性在子类中能否继承？
- 4、在 Python 中什么是异常？
- 5、Python 中是如何处理异常的。
- 6、Python 中异常处理语句的一般格式，可以使用伪代码的形式描述。

7、`__slots__`属性的作用

8、私有化属性的作用？

9、在类外面是否能修改私有属性。

10、如果一个类中，只有指定的属性或者方法能被外部修改。那么该如何限制外部修改。

## 课后实操题

1、编写一段代码以完成下面的要求

- 定义一个 **Person** 类,类中要有初始化方法,方法中要有人的姓名,年龄两个私有属性.
- 提供获取用户信息的函数.
- 提供获取私有属性的方法.
- 提供可以设置私有属性的方法.
- 设置年龄的范围在(0-120)的方法，如果不在这个范围，不能设置成功.

2、请写一个单例模式

3、创建一个类，并定义两个私有化属性，提供一个获取属性的方法，和设置属性的方法。利用 **property** 属性给调用者提供属性方式的调用获取和设置私有属性方法的方式。

4、创建一个 **Animal** 类，实例化一个 **cat** 对象，请给 **cat** 对象动态绑定一个 **run** 方法,给类绑定一个类属性 **colour**,给类绑定一个类方法打印字符串'ok'。

5、观察下面代码，定义了一个私有属性，并提供了两个一个 **set** 方法和一个 **get** 方法，请改下面代码利用 **propetry** 属性方式提供给外部调用 **set** 和 **get** 方法。

```
class Person(object):

    def __init__(self):
        self.__age = 18 # 定义一个私有实例属性
```

```
def get_age(self):  
    return self.__age # 访问私有实例属性  
  
def set_age(self, age):  
    self.__age = age # 修改私有实例属性
```

6、Python 是一门动态语言，那么如何在程序执行过程给实例 **dog** 添加一个实例方法，请给下面代码中添加一个实例方法。

```
class Animal:  
    def __init__(self):  
        print('init')  
  
def eat(self):  
    print('吃草')  
  
dog = Animal()
```