

目录

1	Python语言基础
2	流程控制语句
3	序列的应用
4	函数
5	面向对象程序设计
6	模块
7	文件及目录操作



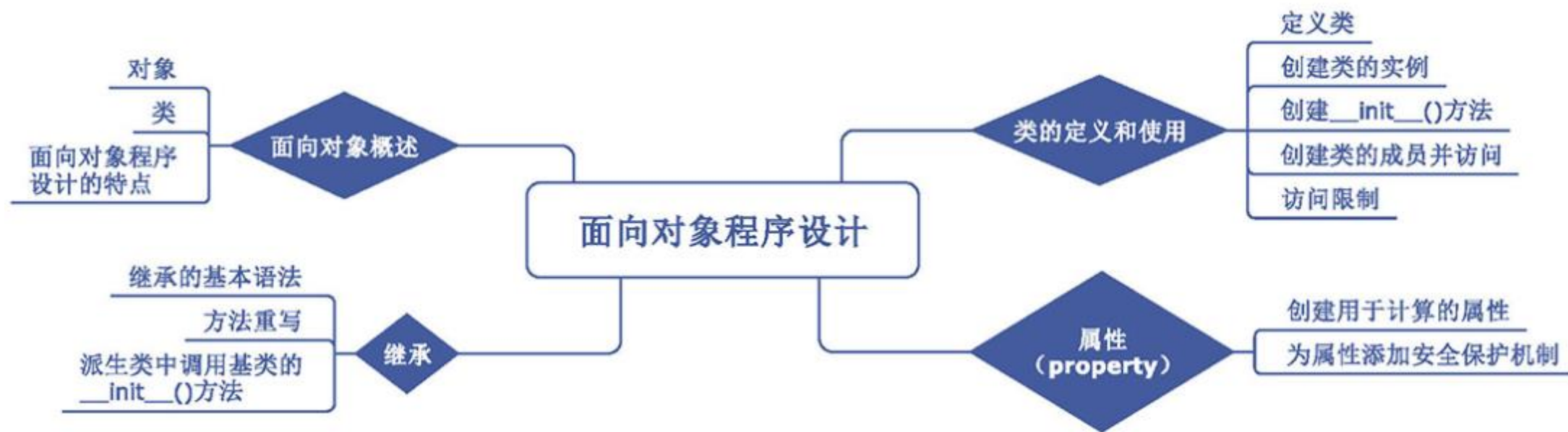


大数据，成就未来



Python语言程序设计 之 面向对象程序设计

知识框架



6.1 面向对象概述

面向对象（Object Oriented）的英文缩写是 OO，它是一种设计思想。从 20 世纪 60 年代提出面向对象的概念到现在，它已经发展成为一种比较成熟的编程思想，并且逐步成为目前软件开发领域的主流技术。如我们经常听说的面向对象编程（Object Oriented Programming，即 OOP）就是主要针对大型软件设计而提出的，它可以使软件设计更加灵活，并且能更好地进行代码复用。

面向对象中的对象（Object），通常是指客观世界中存在的对象，具有唯一性，对象之间各不相同，各有各的特点，每一个对象都有自己的运动规律和内部状态；对象与对象之间又是可以相互联系、相互作用的。另外，对象也可以是一个抽象的事物，例如，可以从圆形、正方形、三角形等图形抽象出一个简单图形，简单图形就是一个对象，它有自己的属性和行为，图形中边的个数是它的属性，图形的面积也是它的属性，输出图形的面积就是它的行为。概括地讲，面向对象技术是一种从组织结构上模拟客观世界的方法。



6.1 面向对象概述

对象

对象，是一个抽象概念，英文称作“Object”，表示任意存在的事物。世间万物皆对象！现实世界中，随处可见的一种事物就是对象，对象是事物存在的实体，如一个人，如图 7.1 所示。



图 7.1 对象“人”的示意图


通常将对象划分为两个部分，即静态部分与动态部分。静态部分被称为“属性”，任何对象都具备自身属性，这些属性不仅是客观存在的，而且是不能被忽视的，如人的性别，如图 7.2 所示；动态部分指的是对象的行为，即对象执行的动作，如人可以跑步，如图 7.3 所示。



图 7.2 静态属性“性别”的示意图



图 7.3 动态属性“跑步”的示意图

 说明：在 Python 中，一切都是对象。即不仅是具体的事物称为对象，字符串、函数等也都是对象。这说明 Python 天生就是面向对象的。

6.1 面向对象概述

类

类是封装对象的属性和行为的载体，反过来说具有相同属性和行为的一类实体被称为类。例如，把雁群比作大雁类，那么大雁类就具备了喙、翅膀和爪等属性，觅食、飞行和睡觉等行为，而一只要从北方飞往南方的大雁则被视为大雁类的一个对象。大雁类和大雁对象的关系如图 7.4 所示。

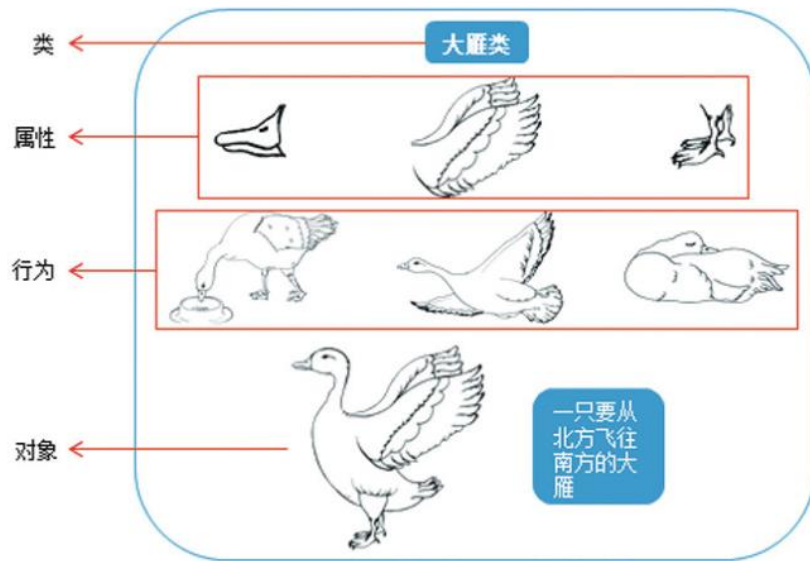


图 7.4 大雁类和大雁对象的关系图

在 Python 语言中，类是一种抽象概念，如定义一个大雁类（Geese），在该类中，可以定义每个对象共有的属性和方法；而一只要从北方飞往南方的大雁则是大雁类的一个对象（wildGeese），对象是类的实例。有关类的具体实现将在 7.2 节进行详细介绍。

6.1 面向对象概述

面向对象程序设计的特点

面向对象程序设计具有三大基本特征：封装、继承和多态。

1. 封装

封装是面向对象编程的核心思想，将对象的属性和行为封装起来，其载体就是类，类通常会对客户隐藏其实现细节，这就是封装的思想。例如，用户使用计算机，只需要使用手指敲击键盘就可以实现一些功能，而不需要知道计算机内部是如何工作的。

采用封装思想保证了类内部数据结构的完整性，使用该类的用户不能直接看到类中的数据结构，而只能执行类允许公开的数据，这样就避免了外部对内部数据的影响，提高了程序的可维护性。

使用类实现封装特性如图 7.5 所示。

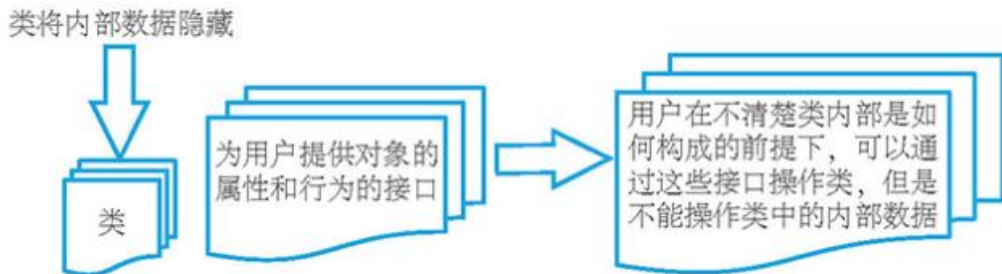


图 7.5 封装特性示意图

6.1 面向对象概述

面向对象程序设计的特点

2. 继承

矩形、菱形、平行四边形和梯形等都是四边形。因为四边形与它们具有共同的特征：拥有 4 条边。只要将四边形适当地延伸，就会得到矩形、菱形、平行四边形和梯形 4 种图形。以平行四边形为例，如果把平行四边形看作四边形的延伸，那么平行四边形就复用了四边形的属性和行为，同时添加了平行四边形特有的属性和行为，如平行四边形的对边平行且相等。在 Python 中，可以把平行四边形类看作是继承四边形类后产生的类，其中，将类似于平行四边形的类称为子类，将类似于四边形的类称为父类或超类。值得注意的是，在阐述平行四边形和四边形的关系时，可以说平行四边形是特殊的四边形，但不能说四边形是平行四边形。同理，Python 中可以说子类的实例都是父类的实例，但不能说父类的实例是子类的实例，四边形类层次结构示意图如图 7.6 所示。

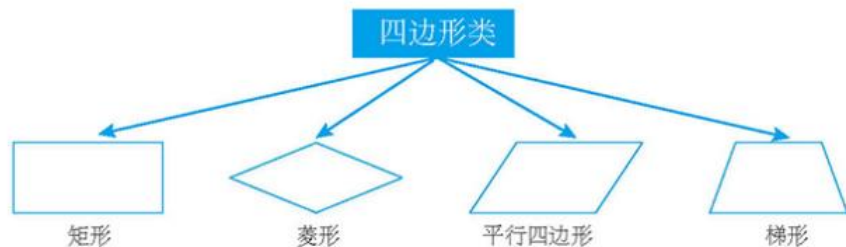


图 7.6 四边形类层次结构示意图

综上所述，继承是实现重复利用的重要手段，子类通过继承复用了父类的属性和行为的同时又添加了子类特有的属性和行为。

6.1 面向对象概述

面向对象程序设计的特点

3. 多态

将父类对象应用于子类的特征就是多态。比如创建一个螺丝类，螺丝类有两个属性：粗细和螺纹密度；然后再创建了两个类，一个是长螺丝类，一个短螺丝类，并且它们都继承了螺丝类。这样长螺丝类和短螺丝类不仅具有相同的特征（粗细相同，且螺纹密度也相同），还具有不同的特征（一个长，一个短，长的可以用来固定大型支架，短的可以固定生活中的家具）。综上所述，一个螺丝类衍生出不同的子类，子类继承父类特征的同时，也具备了自己的特征，并且能够实现不同的效果，这就是多态化的结构。螺丝类层次结构示意图如图 7.7 所示。



图 7.7 螺丝类层次结构示意图

6.2 类的定义和使用

定义类

在 Python 中，类表示具有相同属性和方法的对象的集合。在使用类时，需要先定义类，然后再创建类的实例，通过类的实例就可以访问类中的属性的方法了。

在 Python 中，类的定义使用 `class` 关键字来实现，语法如下：

```
class ClassName:
    """类的帮助信息"""      # 类文档字符串
    statement                 # 类体
```

参数说明：

☑ `ClassName`：用于指定类名，一般使用大写字母开头，如果类名中包括两个单词，第二个单词的首字母也大写，这种命名方法也称为“驼峰式命名法”，这是惯例。当然，也可根据自己的习惯命名，但是一般推荐按照惯例来命名。

☑ `"""类的帮助信息"""`：用于指定类的文档字符串，定义该字符串后，在创建类的对象时，输入类名和左侧的括号“(”后，将显示该信息。

☑ `statement`：类体，主要由类变量（或类成员）、方法和属性等定义语句组成。如果在定义类时，没想好类的具体功能，也可以在类体中直接使用 `pass` 语句代替。

例如，下面以大雁为例声明一个类，代码如下：

```
class Geese:
    """大雁类"""
    pass
```



6.2 类的定义和使用

创建类的实例

定义完类后，并不会真正创建一个实例。这有点像一个汽车的设计图。设计图可以告诉你汽车看上去怎么样，但设计图本身不是一个汽车。你不能开走它，它只能用来建造真正的汽车，而且可以使用它制造很多汽车。那么如何创建实例呢？

`class` 语句本身并不创建该类的任何实例。所以在类定义完成以后，可以创建类的实例，即实例化该类的对象。创建类的实例的语法如下：

```
ClassName(parameterlist)
```

其中，`ClassName` 是必选参数，用于指定具体的类；`parameterlist` 是可选参数，当创建一个类时，没有创建 `__init__()` 方法（该方法将在 7.2.3 小节进行详细介绍），或者 `__init__()` 方法只有一个 `self` 参数时，`parameterlist` 可以省略。

例如，创建 7.2.1 小节定义的 `Geese` 类的实例，可以使用下面的代码：


```
wildGoose = Geese()           # 创建大雁类的实例  
print(wildGoose)
```



6.2 类的定义和使用

创建__init__()方法

在创建类后，可以手动创建一个 `__init__()` 方法。该方法是一个特殊的方法，类似 Java 语言中的构造方法。每当创建一个类的新实例时，Python 都会自动执行它。`__init__()` 方法必须包含一个 `self` 参数，并且必须是第一个参数。`self` 参数是一个指向实例本身的引用，用于访问类中的属性和方法。在方法调用时会自动传递实际参数 `self`，因此当 `__init__()` 方法只有一个参数时，在创建类的实例时，就不需要指定实际参数了。

 说明：在 `__init__()` 方法的名称中，开头和结尾处是两个下划线（中间没有空格），这是一种约定，旨在区分 Python 默认方法和普通方法。

例如，下面仍然以大雁为例声明一个类，并且创建 `__init__()` 方法，代码如下：

```
class Geese:
    """大雁类"""
    def __init__(self):          # 构造方法
        print("我是大雁类!")
wildGoose = Geese()           # 创建大雁类的实例
```

运行上面的代码，将输出以下内容：

```
我是大雁类！
```

从上面的运行结果可以看出，在创建大雁类的实例时，虽然没有为 `__init__()` 方法指定参数，但是该方法会自动执行。



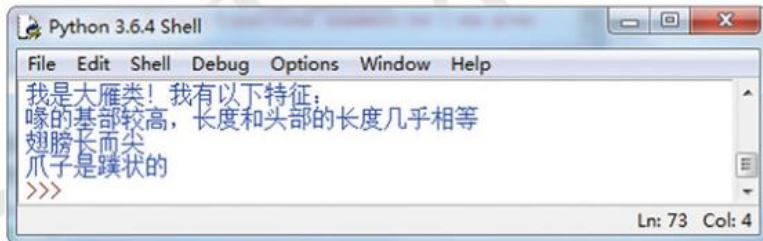
6.2 类的定义和使用

创建__init__()方法

在 `__init__()` 方法中，除了 `self` 参数外，还可以自定义一些参数，参数间使用逗号“,”进行分隔。例如，下面的代码将在创建 `__init__()` 方法时，再指定 3 个参数，分别是 `beak`、`wing` 和 `claw`。

```
class Geese:
    """大雁类"""
    def __init__(self, beak, wing, claw):          # 构造方法
        print("我是大雁类! 我有以下特征:")
        print(beak)                               # 输出喙的特征
        print(wing)                               # 输出翅膀的特征
        print(claw)                               # 输出爪子的特征
    beak_1 = "喙的基部较高, 长度和头部的长度几乎相等" # 喙的特征
    wing_1 = "翅膀长而尖"                          # 翅膀的特征
    claw_1 = "爪子是蹼状的"                         # 爪子的特征
    wildGoose = Geese(beak_1, wing_1, claw_1)        # 创建大雁类的实例
```

执行上面的代码，将显示如图 7.9 所示的运行结果。



6.3 创建类的成员并访问

类的成员主要由实例方法和数据成员组成。在类中创建了类的成员后，可以通过类的实例进行访问。

1. 创建实例方法并访问

所谓实例方法是指在类中定义的函数。该函数是一种在类的实例上操作的函数。同 `__init__()` 方法一样，实例方法的第一个参数必须是 `self`，并且必须包含一个 `self` 参数。创建实例方法的语法格式如下：

```
def functionName(self,parameterlist):  
    block
```

参数说明：

- ❑ `functionName`：用于指定方法名，一般使用小写字母开头。
- ❑ `self`：必要参数，表示类的实例，其名称可以是 `self` 以外的单词，使用 `self` 只是一个惯例而已。
- ❑ `parameterlist`：用于指定除 `self` 参数以外的参数，各参数间使用逗号“,”进行分隔。
- ❑ `block`：方法体，实现的具体功能。

📖 说明：实例方法和 Python 中的函数的主要区别就是，函数实现的是某个独立的功能，而实例方法是实现类中的一个行为，是类的一部分。



6.3 创建类的成员并访问

实例方法

实例方法创建完成后，可以通过类的实例名称和点 (.) 操作符进行访问，语法格式如下：

```
instanceName.functionName(parametervalue)
```

参数说明：

- ☑ instanceName：为类的实例名称。
- ☑ functionName：为要调用的方法名称。
- ☑ parametervalue：表示为方法指定对应的实际参数，其值的个数与创建实例方法中 parameterlist 的个数相同。



6.3 创建类的成员并访问

实例：创建大雁类并定义飞行方法

```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
我是大雁类！我有以下特征：
喙的基部较高，长度和头部的长度几乎相等
翅膀长而尖
爪子是蹼状的
我飞行时，一会儿排成个人字，一会排成一字
>>>
```

```
01 class Geese:                                     # 创建大雁类
02     """大雁类"""
03     def __init__(self, beak, wing, claw):          # 构造方法
04         print("我是大雁类！我有以下特征：")
05         print(beak)                                # 输出喙的特征
06         print(wing)                                # 输出翅膀的特征
07         print(claw)                                # 输出爪子的特征
08     def fly(self, state):                          # 定义飞行方法
09         print(state)
10     """*****调用方法*****"""
11     beak_1 = "喙的基部较高，长度和头部的长度几乎相等" # 喙的特征
12     wing_1 = "翅膀长而尖"                          # 翅膀的特征
13     claw_1 = "爪子是蹼状的"                        # 爪子的特征
14     wildGoose = Geese(beak_1, wing_1, claw_1)        # 创建大雁类的实例
15     wildGoose.fly("我飞行时，一会儿排成个人字，一会排成一字") # 调用实例方法
```


6.3 创建类的成员并访问

数据成员——类属性

2. 创建数据成员并访问

数据成员是指在类中定义的变量，即属性，根据定义位置，又可以分为类属性和实例属性。

☑ 类属性

类属性是指定义在类中，并且在函数体外的属性。类属性可以在类的所有实例之间共享值，也就是在所有实例化的对象中公用。

📖 说明：类属性可以通过类名称或者实例名访问。

例如，定义一个雁类 Geese，在该类中定义 3 个类属性，用于记录雁类的特征，代码如下：

```
class Geese:
    """雁类"""
    neck = "脖子较长"           # 定义类属性（脖子）
    wing = "振翅频率高"         # 定义类属性（翅膀）
    leg = "腿位于身体的中心支点，行走自如" # 定义类属性（腿）
    def __init__(self):         # 实例方法（相当于构造方法）
        print("我属于雁类！我有以下特征：")
        print(Geese.neck)      # 输出脖子的特征
        print(Geese.wing)      # 输出翅膀的特征
        print(Geese.leg)       # 输出腿的特征
```



6.3 创建类的成员并访问

数据成员——类属性

创建上面的类 Geese，然后创建该类的实例，代码如下：

```
geese = Geese() # 实例化一个雁类的对象
```

应用上面的代码创建 Geese 类的实例后，将显示以下内容：

我是雁类！我有以下特征：

脖子较长

振翅频率高

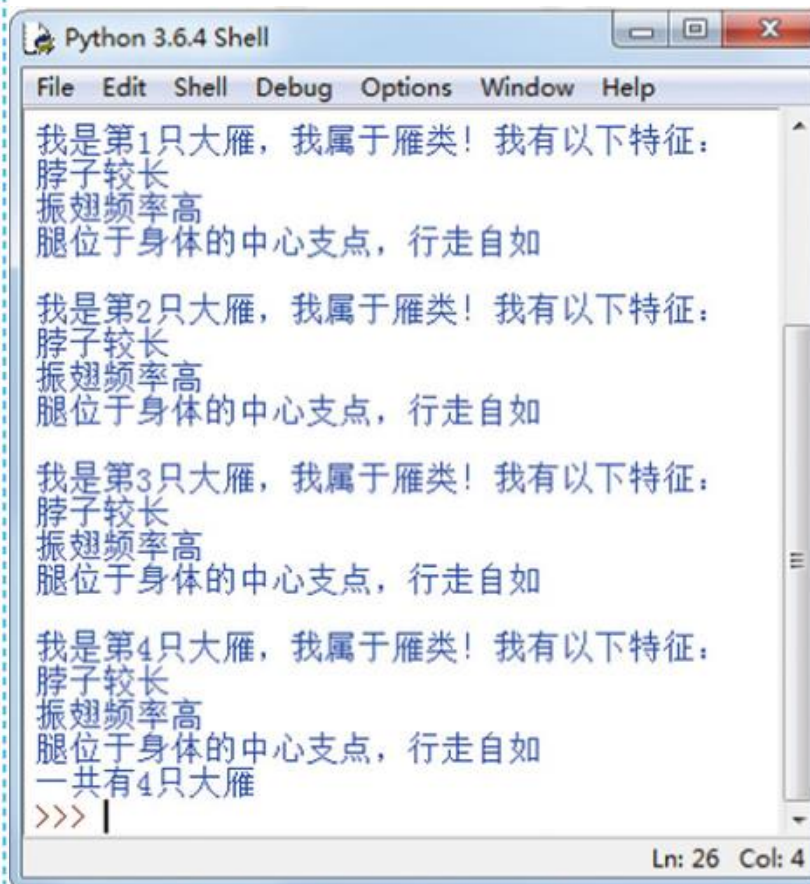
腿位于身体的中心支点，行走自如



6.3 创建类的成员并访问

实例：通过类属性统计类的实例个数

```
01 class Geese:
02     """雁类"""
03     neck = "脖子较长"           # 类属性（脖子）
04     wing = "振翅频率高"         # 类属性（翅膀）
05     leg = "腿位于身体的中心支点，行走自如" # 类属性（腿）
06     number = 0                  # 编号
07     def __init__(self):         # 构造方法
08         Geese.number += 1       # 将编号加1
09         print("\n我是第"+str(Geese.number)+"只大雁，我属于雁类！我有以下特征：")
10         print(Geese.neck)       # 输出脖子的特征
11         print(Geese.wing)       # 输出翅膀的特征
12         print(Geese.leg)        # 输出腿的特征
13 # 创建4个雁类的对象（相当于有4只大雁）
14 list1 = []
15 for i in range(4):             # 循环4次
16     list1.append(Geese())       # 创建一个雁类的实例
17 print("一共有"+str(Geese.number)+"只大雁")
```



```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help

我是第1只大雁，我属于雁类！我有以下特征：
脖子较长
振翅频率高
腿位于身体的中心支点，行走自如

我是第2只大雁，我属于雁类！我有以下特征：
脖子较长
振翅频率高
腿位于身体的中心支点，行走自如

我是第3只大雁，我属于雁类！我有以下特征：
脖子较长
振翅频率高
腿位于身体的中心支点，行走自如

我是第4只大雁，我属于雁类！我有以下特征：
脖子较长
振翅频率高
腿位于身体的中心支点，行走自如
一共有4只大雁
>>> |
```

Ln: 26 Col: 4




6.3 创建类的成员并访问

实例：创建大雁类并定义飞行方法


在 Python 中除了可以通过类名称访问类属性，还可以动态地为类和对象添加属性。例如，在实例 02 的基础上为雁类添加一个 beak 属性，并通过类的实例访问该属性，可以在上面代码的后面再添加以下代码：

```
Geese.beak = "喙的基部较高，长度和头部的长度几乎相等" # 添加类属性
print("第2只大雁的喙：", list1[1].beak)                  # 访问类属性
```

 说明：上面的代码只是以第 2 只大雁为例进行演示，读者也可以换成其他的大雁试试。

运行后，将在原来的结果后面再显示以下内容：

```
第2只大雁的喙： 喙的基部较高，长度和头部的长度几乎相等
```

 说明：除了可以动态地为类和对象添加属性，也可以修改类属性。修改结果将作用于该类的所有实例。



6.3 创建类的成员并访问

数据成员——实例属性

- 实例属性是指定义在类的方法中的属性，只作用于当前实例中。

```
class Geese:
    """雁类"""
    def __init__(self):
        self.neck = "脖子较长"
        print(self.neck)
goose1 = Geese()
goose2 = Geese()
goose1.neck = "脖子没有天鹅的长"
print("goose1的neck属性:", goose1.neck)
print("goose2的neck属性:", goose2.neck)
```

实例方法（相当于构造方法）
定义实例属性（脖子）
输出脖子的特征
创建Geese类的实例1
创建Geese类的实例2
修改实例属性

运行上面的代码，将显示以下内容：

```
脖子较长
脖子较长
goose1的neck属性: 脖子没有天鹅的长
goose2的neck属性: 脖子较长
```



6.3 创建类的成员并访问

访问限制

在类的内部可以定义属性和方法，而在类的外部则可以直接调用属性或方法来操作数据，从而隐藏了类内部的复杂逻辑。但是 Python 并没有对属性和方法的访问权限进行限制。为了保证类内部的某些属性或方法不被外部所访问，可以在属性或方法名前面添加单下划线（`_foo`）、双下划线（`__foo`）或首尾加双下划线（`__foo__`），从而限制访问权限。其中，单下划线、双下划线、首尾双下划线的作用如下：

（1）首尾双下划线表示定义特殊方法，一般是系统定义名字，如 `__init__()`。

（2）以单下划线开头的表示 `protected`（保护）类型的成员，只允许类本身和子类进行访问，但不能使用“`from module import *`”语句导入。

例如，创建一个 `Swan` 类，定义保护属性 `_neck_swan`，并使用 `__init__()` 方法访问该属性，然后创建 `Swan` 类的实例，并通过实例名输出保护属性 `_neck_swan`，代码如下：

```
class Swan:
    """天鹅类"""
    _neck_swan = '天鹅的脖子很长'           # 定义保护属性
    def __init__(self):
        print("__init__():", Swan._neck_swan)  # 在实例方法中访问保护属性
swan = Swan()                               # 创建Swan类的实例
print("直接访问:", swan._neck_swan)         # 保护属性可以通过实例名访问
```

执行下面的代码，将显示以下内容：

```
__init__(): 天鹅的脖子很长
直接访问: 天鹅的脖子很长
```

从上面的运行结果中可以看出，保护属性可以通过实例名访问。



6.3 创建类的成员并访问

访问限制

(3) 双下划线表示 private（私有）类型的成员，只允许定义该方法的类本身进行访问，而且也不能通过类的实例进行访问，但是可以通过“类的实例名._类名__xxx”方式访问。

例如，创建一个 Swan 类，定义私有属性 __neck_swan，并使用 __init__() 方法访问该属性，然后创建 Swan 类的实例，并通过实例名输出私有属性 __neck_swan，代码如下：

```
class Swan:
    """天鹅类"""
    __neck_swan = '天鹅的脖子很长'           # 定义私有属性
    def __init__(self):
        print("__init__():", Swan.__neck_swan) # 在实例方法中访问私有属性
swan = Swan()                                # 创建Swan类的实例
print("加入类名:", swan._Swan__neck_swan)    # 私有属性，可以通过“实例名._类名__xxx”方式访问
print("直接访问:", swan.__neck_swan)         # 私有属性不能通过实例名访问，出错
```

执行上面的代码后，将输出如图 7.13 所示的结果。

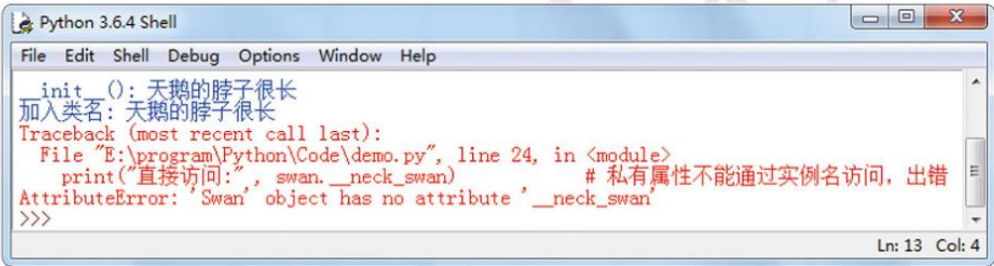


图 7.13 访问私有属性

从上面的运行结果可以看出：私有属性不能直接通过实例名 + 属性名访问，可以在类的实例方法中访问，也可以通过“实例名._类名__xxx”方式访问。

6.4 用于计算的属性

创建用于计算的属性

- 6.3节介绍类属性和实例属性将返回所存储的值，而本节要介绍的属性则是一种特殊的属性，访问它时将计算它的值。
- 另外，该属性还可以为属性添加安全保护机制。

在 Python 中，可以通过 `@property`（装饰器）将一个方法转换为属性，从而实现用于计算的属性。将方法转换为属性后，可以直接通过方法名来访问方法，而不需要再添加一对小括号“()”，这样可以让代码更加简洁。



6.4 用于计算的属性

创建用于计算的属性

通过 `@property` 创建用于计算的属性的语法格式如下：

```
@property  
def methodname(self):  
    block
```

参数说明：

- ☑ `methodname`：用于指定方法名，一般使用小写字母开头。该名称最后将作为创建的属性名。
- ☑ `self`：必要参数，表示类的实例。
- ☑ `block`：方法体，实现的具体功能。在方法体中，通常以 `return` 语句结束，用于返回计算结果。



6.4 用于计算的属性

创建用于计算的属性

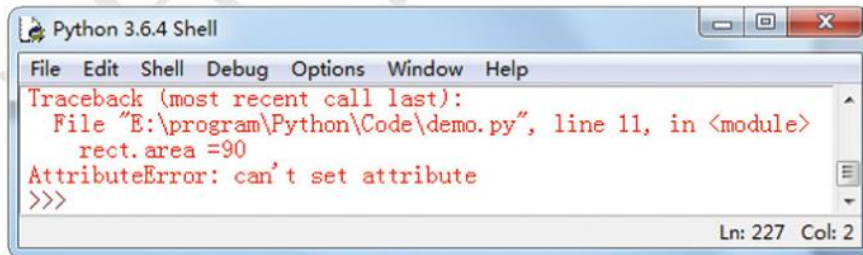
```
class Rect:
    def __init__(self,width,height):
        self.width = width
        self.height = height
    @property
    def area(self):
        return self.width*self.height
rect = Rect(800,600)
print("面积为: ",rect.area)
```

矩形的宽
矩形的高
将方法转换为属性
计算矩形的面积的方法
返回矩形的面积
创建类的实例
输出属性的值

运行上面的代码，将显示以下运行结果：

面积为: 480000

⚡ 注意：通过 `@property` 转换后的属性不能重新赋值，如果对其重新赋值，将抛出如图 7.14 所示的异常信息。



6.4 用于计算的属性

为属性添加安全保护机制

在 Python 中，默认情况下，创建的属性或者实例是可以在类体外进行修改的，如果想要限制其不能在类体外修改，可以将其设置为私有的，但设置为私有后，在类体外也不能获取它的值。如果想要创建一个可以读取但不能修改的属性，那么可以使用 `@property` 实现只读属性。

```
class TVshow:    # 定义电视节目类
    def __init__(self, show):
        self.__show = show
    @property    # 将方法转换为属性
    def show(self):    # 定义show()方法
        return self.__show    # 返回私有属性的值
tvshow = TVshow("正在播放《战狼2》")    # 创建类的实例
print("默认: ", tvshow.show)    # 获取属性值
```

执行上面的代码，将显示以下内容：

```
默认:  正在播放《战狼2》
```



6.4 用于计算的属性

为属性添加安全保护机制

通过上面的方法创建的 `show` 属性是只读的，尝试修改该属性的值，再重新获取。在上面代码中添加以下代码：

```
tvshow.show = "正在播放《红海行动》"      # 修改属性值  
print("修改后:", tvshow.show)             # 获取属性值
```

运行后，将显示如图 7.15 所示的运行结果，其中红字的异常信息就是修改属性 `show` 时抛出的异常。

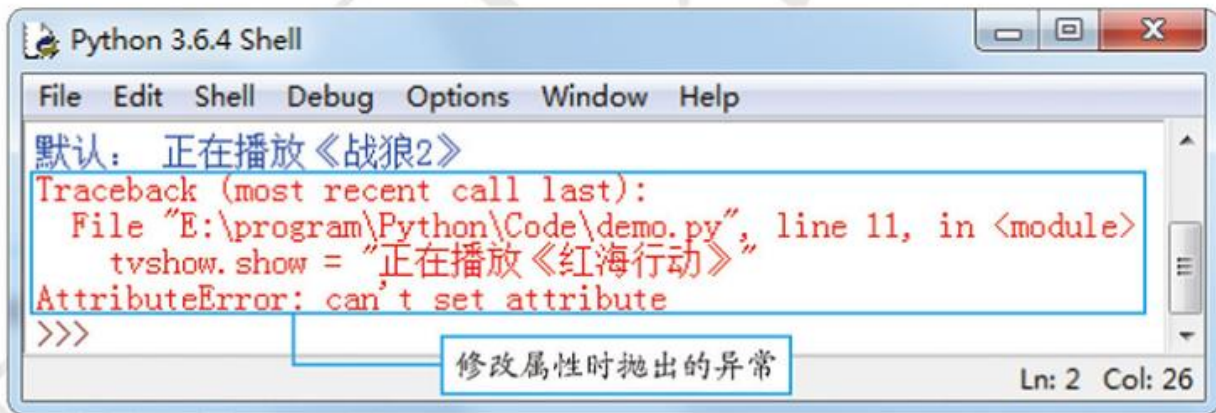


图 7.15 修改只读属性时抛出的异常

6.4 用于计算的属性

实例：在模拟电影点播功能时应用属性

情景模拟：某电视台开设了电影点播功能，但要求只能从指定的几个电影（如《战狼2》《红海行动》《西游记女儿国》《熊出没·变形记》）中选择一个。

```
01 class TVshow:                                # 定义电视节目类
02     list_film = ["战狼2","红海行动","西游记女儿国","熊出没·变形记"]
03     def __init__(self,show):
04         self.__show = show
05     @property                                  # 将方法转换为属性
06     def show(self):                            # 定义show()方法
07         return self.__show                   # 返回私有属性的值
08     @show.setter                               # 设置setter方法，让属性可修改
09     def show(self,value):
10         if value in TVshow.list_film:        # 判断值是否在列表中
11             self.__show = "您选择了《" + value + "》，稍后将播放" # 返回修改的值
12         else:
13             self.__show = "您点播的电影不存在"
14 tvshow = TVshow("战狼2")                     # 创建类的实例
15 print("正在播放：《",tvshow.show,"》")        # 获取属性值
16 print("您可以从",tvshow.list_film,"中选择要点播放的电影")
17 tvshow.show = "红海行动"                    # 修改属性值
18 print(tvshow.show)                          # 获取属性值
```



```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
正在播放：《 战狼2 》
您可以从 ['战狼2', '红海行动', '西游记女儿国', '熊出没·变形记'] 中选择要点播放的电影
您选择了《红海行动》，稍后将播放
>>>
```



```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
正在播放：《 战狼2 》
您可以从 ['战狼2', '红海行动', '西游记女儿国', '熊出没·变形记'] 中选择要点播放的电影
您点播的电影不存在
>>>
```

6.5 继承

- 在编写类时，并不是每次都要从空白开始。当要编写的类和另一个已经存在的类之间存在一定的继承关系时，就可以通过继承来达到代码重用的目的，提高开发效率。

继承是面向对象编程最重要的特性之一，它源于人们认识客观世界的过程，是自然界普遍存在的一种现象。例如，我们每一个人都从祖辈和父母那里继承了一些体貌特征，但是每个人却又不同于父母，因为每个人都存在自己的一些特性，这些特性是独有的，在父母身上并没有体现。在程序设计中实现继承，表示这个类拥有它继承的类的所有公有成员或者受保护成员。在面向对象编程中，被继承的类称为父类或基类，新的类称为子类或派生类。

FOUNDED IN 1977



6.5 继承

具体的语法格式

```
class ClassName(baseclasslist):  
    """类的帮助信息"""           # 类文档字符串  
    statement                       # 类体
```

参数说明：

- ☑ **ClassName**：用于指定类名。
- ☑ **baseclasslist**：用于指定要继承的基类，可以有多个，类名之间用逗号“,”分隔。如果不指定，将使用所有 Python 对象的根类 object。
- ☑ **""" 类的帮助信息 """**：用于指定类的文档字符串，定义该字符串后，在创建类的对象时，输入类名和左侧的括号“(”后，将显示该信息。
- ☑ **statement**：类体，主要由类变量（或类成员）、方法和属性等定义语句组成。如果在定义类时，没想好类的具体功能，也可以在类体中直接使用 `pass` 语句代替。



6.5 继承

实例：创建水果基类及其派生类

```
01 class Fruit:
02     color = "绿色"
03     def harvest(self, color):
04         print("水果是: " + color + "的!")
05         print("水果已经收获……")
06         print("水果原来是: " + Fruit.color + "的!");
07 class Apple(Fruit):
08     color = "红色"
09     def __init__(self):
10         print("我是苹果")
```

定义水果类（基类）

定义类属性

输出的是形式参数color

输出的是类属性color

定义苹果类（派生类）

```
11 class Orange(Fruit):
12     color = "橙色"
13     def __init__(self):
14         print("\n我是橘子")
15 apple = Apple()
16 apple.harvest(apple.color)
17 orange = Orange()
18 orange.harvest(orange.color)
```

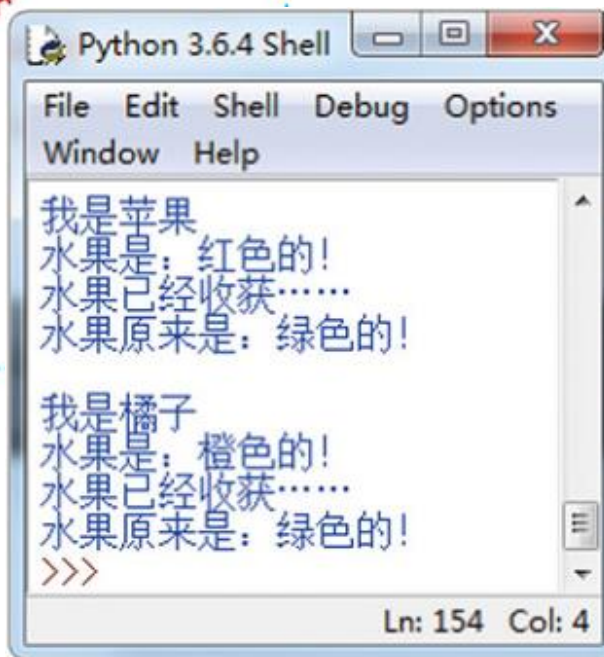
定义橘子类（派生类）

创建类的实例（苹果）

调用基类的harvest()方法

创建类的实例（橘子）

调用基类的harvest()方法



```
Python 3.6.4 Shell
File Edit Shell Debug Options
Window Help

我是苹果
水果是: 红色的!
水果已经收获……
水果原来是: 绿色的!

我是橘子
水果是: 橙色的!
水果已经收获……
水果原来是: 绿色的!
>>>
```

Ln: 154 Col: 4

6.5 继承

方法重写

基类的成员都会被派生类继承，当基类中的某个方法不完全适用于派生类时，就需要在派生类中重写父类的这个方法，这和 Java 语言中的方法重写是一样的。

```
01 class Orange(Fruit):                                # 定义橘子类（派生类）
02     color = "橙色"
03     def __init__(self):
04         print("\n我是橘子")
05     def harvest(self, color):
06         print("橘子是：" + color + "的！")           # 输出的是形式参数color
07         print("橘子已经收获……")
08         print("橘子原来是：" + Fruit.color + "的！"); # 输出的是类属性color
```



```
Python 3.6.4 Shell
File Edit Shell Debug Options
Window Help

我是苹果
水果是：红色的！
水果已经收获……
水果原来是：绿色的！

我是橘子
橘子是：橙色的！
橘子已经收获……
橘子原来是：绿色的！
>>>
```

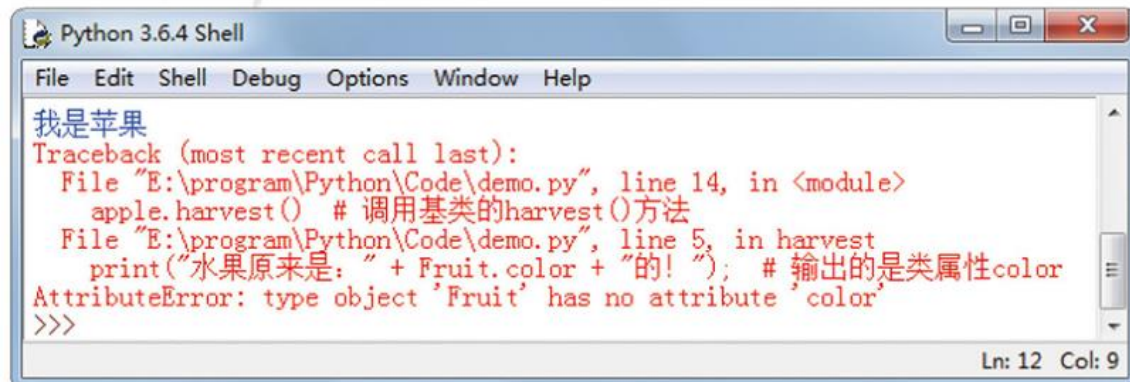
6.5 继承

派生类中调用基类的__init__()方法

- 在派生类中定义__init__()方法时，不会自动调用基类的__init__()方法。

```
01 class Fruit:                                # 定义水果类（基类）
02     def __init__(self,color = "绿色"):
03         Fruit.color = color                  # 定义类属性
04     def harvest(self):
05         print("水果原来是: " + Fruit.color + "的! ");    # 输出的是类属性color
06 class Apple(Fruit):                          # 定义苹果类（派生类）
07     def __init__(self):
08         print("我是苹果")
09 apple = Apple()                             # 创建类的实例（苹果）
10 apple.harvest()                             # 调用基类的harvest()方法
```

执行上面的代码后，将显示如图 7.20 所示的异常信息。



6.5 继承

派生类中调用基类的__init__()方法

➤ 需要在派生类的__init__()方法中使用super()函数调用基类的__init__()方法。

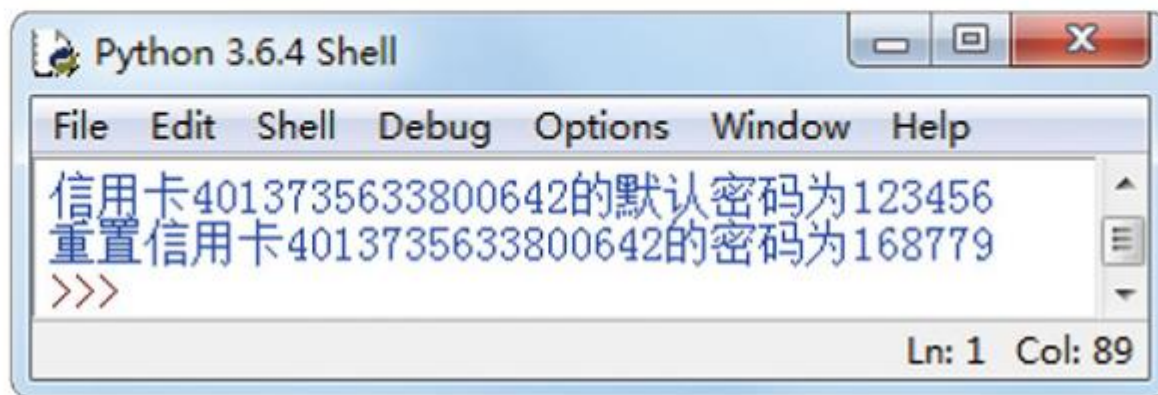
```
01 class Fruit:                                # 定义水果类（基类）
02     def __init__(self, color="绿色"):        # 定义类属性
03         Fruit.color = color
04     def harvest(self, color):                # 输出的是形式参数color
05         print("水果是: " + self.color + "的!")
06         print("水果已经收获.....")
07         print("水果原来是: " + Fruit.color + "的!");    # 输出的是类属性color
08 class Apple(Fruit):                          # 定义苹果类（派生类）
09     color = "红色"
10     def __init__(self):
11         print("我是苹果")
12         super().__init__()                  # 调用基类的__init__()方法
13 class Sapodilla(Fruit):                     # 定义人参果类（派生类）
14     def __init__(self, color):
15         print("\n我是人参果")
16         super().__init__(color)             # 调用基类的__init__()方法
17     # 重写harvest()方法的代码
18     def harvest(self, color):
19         print("人参果是: " + color + "的!")    # 输出的是形式参数color
20         print("人参果已经收获.....")
21         print("人参果原来是: " + Fruit.color + "的!"); # 输出的是类属性color
22 apple = Apple()                             # 创建类的实例（苹果）
23 apple.harvest(apple.color)                  # 调用harvest()方法
24 sapodilla = Sapodilla("白色")              # 创建类的实例（人参果）
25 sapodilla.harvest("金黄色带紫色条纹")     # 调用harvest()方法
```



课后作业

作业一：给信用卡设置密码

创建信用卡类，并且为该类创建一个构造方法，该构造方法有 3 个参数，分别是 `self`、卡号和密码。其中，密码可以设置一个默认值 123456，代表默认密码。在创建类的实例时，如果不指定密码，就采用默认密码，否则要重置密码。效果如图 7.23 所示。



```
Python 3.6.4 Shell
File Edit Shell Debug Options Window Help
信用卡4013735633800642的默认密码为123456
重置信用卡4013735633800642的密码为168779
>>>
Ln: 1 Col: 89
```

课后作业

作业二：打印每月销售明细

模拟实现输出进销存管理系统中的每月销售明细，运行程序，输入要查询的月份，如果输入的月份存在销售明细，则显示本月商品销售明细；如果输入的月份不存在或不是数字，则提示“该月没有销售数据或者输入的月份有误！”。效果如图 7.24 所示。



课后作业

作业三：模拟电影院的自动售票页面

在电影院中观看电影是一项很受欢迎的休闲娱乐，现请模拟电影院自动售票机中自动选择电影场次的页面，例如，一部电影在当日的播放时间有很多，可以自动选择合适的场次。效果如图 7.25 所示。





大数据，成就未来



Thank you!