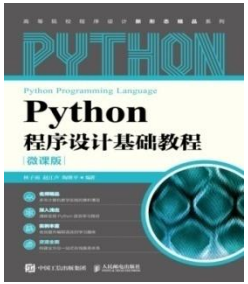
《Python程序设计基础教程（微课版）》

http://dblab.xmu.edu.cn/post/python

第7章 面向对象程序设计

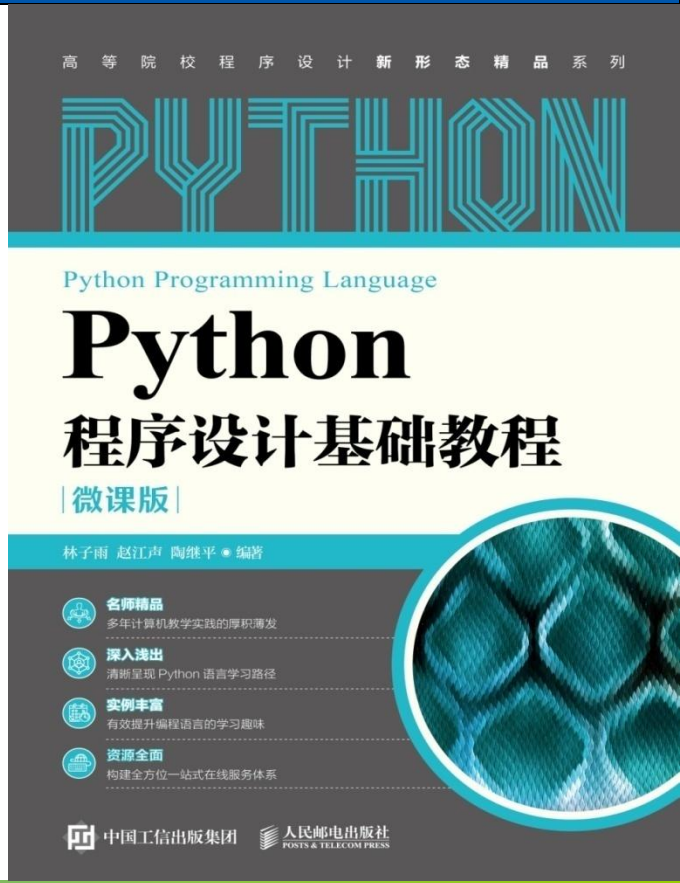


第7章 面向对象程序设计

本PPT是如下教材的配套讲义：
《Python程序设计基础教程（微课版）》

厦门大学 林子雨,赵江声,陶继平 编著，人民邮电出版社

《 Python程序设计基础教程（微课版） 》教材官方网站：
http://dblab.xmu.edu.cn/post/python



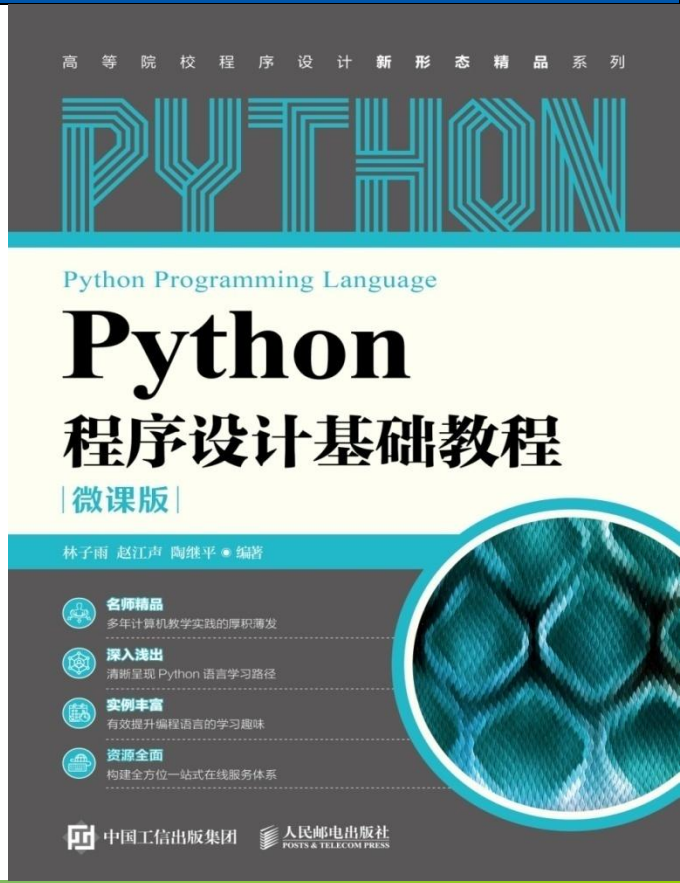
7.1 面向对象编程概述

7.1.1 对象与类
7.1.2 继承与多态

本PPT是如下教材的配套讲义：
《Python程序设计基础教程（微课版）》

厦门大学 林子雨,赵江声,陶继平 编著，人民邮电出版社

《 Python程序设计基础教程（微课版） 》教材官方网站：
http://dblab.xmu.edu.cn/post/python



7.1.1 对象与类

- 面向对象编程的核心概念是对象。
- 在我们的自然语言里，对象是对客观存在事物的一个统称。
- 不同的对象之间相互依存但边界清晰。
- 在计算机程序世界里，对象是建立在数据和基于数据的操作之上的实体。
- 对象之间逻辑边界清晰，在对象之外，整个程序就是各种对象的生成、调用、交互与销毁的过程。
- 通过对象的概念，将数据和业务逻辑细节进行合适的隐藏，只对外提供一些访问接口，用户无需知道对象内部实现的细节，但可以通过该对象对外提供的接口与对象进行安全有效的交互。
- 这个过程称为“封装”，它使得程序的组织更加清晰，可读性更强，同时可扩展性更好。

7.1.1 对象与类

- 与对象紧密相关的一个概念是“类”。
- 类是对不同对象的共同属性和共同行为特征的抽象。
- 作为一种抽象的数据类型，类定义了对象所具有的静态特征和动态行为。
- 在面向对象的术语里，一般将静态特征称为“属性”，将动态行为称为“方法”。
- 例如，在《模拟人生》这个游戏里，可以定义一个“建筑工”类来建模现实世界的建筑工人，它可能包括姓名、年龄及身高等属性，同时还具有“走路”、“跑步”及“搬砖”等方法。

7.1.1 对象与类

•通俗地说，类是用来创建对象的蓝图或模板。

•通过类这个模板，可以制造出一个个具体的对象，这些对象称为“类的实例”（instance），这个过程称为“实例化”（instantiation）。

•从同一个类实例化出的对象具有相同的属性和方法，但属性的取值可能不同，方法的执行结果也可能不同。

•例如，通过前面定义的“建筑工”类可以实例化出张三、李四等具体的建筑工实例，但他们的年龄身高等属性值可能不一样，走路的快慢和搬砖的能力也可能不一样。

7.1.2 继承与多态

•一个系统的多个类通常是存在功能上的纵向层级关系的，也就是说，一种类型可能是另一种类型的一种扩展或者特殊化。

•因此，为了提高代码的复用性，可以将两个类型所共有的功能抽象为一个上层类型，而将扩展的功能抽象为一个下层类型。

•在面向对象的编程术语里，用继承来表示这种类型之间的层级关系。

•继承是通过已存在的类来建立新类的技术，已存在的类称为“父类”或“基类”，新类称为“子类”或“派生类”。

•新类不仅继承了父类的属性和方法，还可以增加新的属性和方法。

7.1.2 继承与多态

•通俗地说，继承所描述的是一种从属关系，如果有两个类型A和B可以描述为“B是A”，则可以表示为B继承A。

•例如，如果有一个“动物”类和“狗”类，因为“狗是动物”，所以可以表示为“狗”类继承“动物”类。

•继承所表示的从属关系具有传递性，如果B继承A，C继承B，那么也可以说C继承A。

7.1.2 继承与多态

•子类在继承父类时，除了可以原封不动地继承父类的方法，还可以修改父类的方法，这个过程称为“重写”（override）。

•基于这一原则，经常将一些具有类似功能但实现细节不是完全一致的方法抽象到父类中，父类只描述这种方法的对外输入输出功能，而不给出具体的实现，这种方法称为“抽象方法”，包含这种方法的类称为“抽象类”。

•当从抽象类派生子类时，子类需要根据具体的场景实现父类中定义的抽象方法，不同的子类给出的具体实现可能不一样。

•例如，假设“动物”类具有一个“发出叫声”的方法，但不同的动物发出的叫声可能不一样，“狗”这个子类是旺旺叫，“猫”子类是喵喵叫。

•这种同一个方法具有不同实现的特征，称为“多态”（Polymorphism）。

7.1.2 继承与多态

•多态使得程序的灵活性或可扩展性更好。

•例如，假设有一个“动物”类型的数组，可以遍历数组统一调用“发出叫声”这个方法，而不需要考虑实际运行时这个数组包含的是“狗”类型的对象还是“猫”类型的对象，多态行为保证了“发出叫声”会自动动态绑定到“狗”或者“猫”的“发出叫声”方法，即使后续增加了一个新的“鸭子”子类对象，它也会自动发出呱呱叫，而无需修改原来的调用代码。

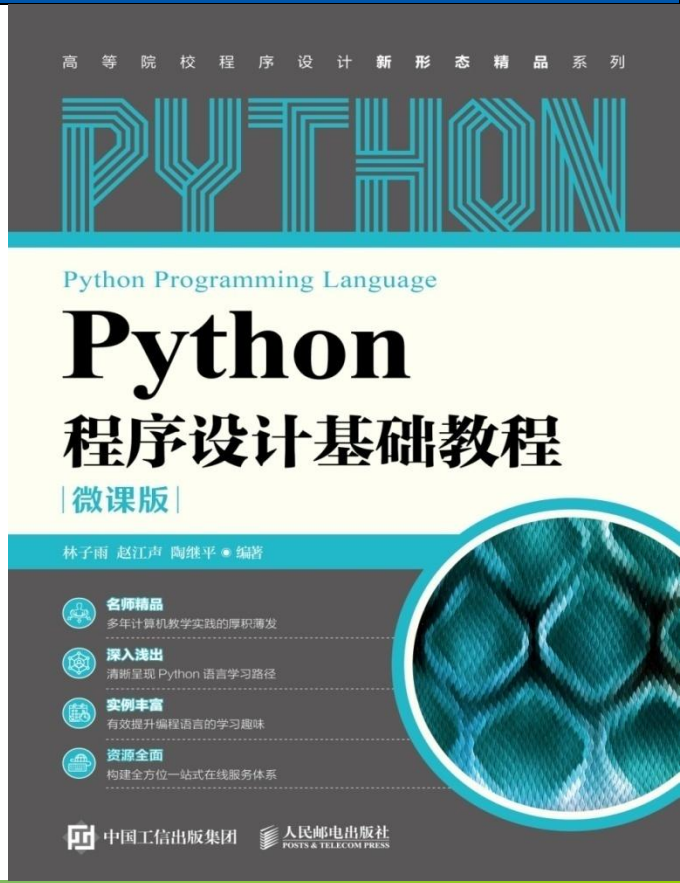
7.2 Python中的面向对象

7.2.1 对象
7.2.2 类

本PPT是如下教材的配套讲义：
《Python程序设计基础教程（微课版）》

厦门大学 林子雨,赵江声,陶继平 编著，人民邮电出版社

《 Python程序设计基础教程（微课版） 》教材官方网站：
http://dblab.xmu.edu.cn/post/python



7.2.1 对象

•每个对象可以有多个变量名，但只有一个唯一的整数值身份号（identity），可以使用内置函数id()来查看对象的身份号。
•一个对象一旦被创建，身份号就保持不变。
•Python采用引用计数技术自动管理对象所占用内存的回收，一般来说，用户不需要关注该过程。
•可以用is操作符检查两个对象是否是同一个对象，也就是身份号是否相等。如果对象所包含的值不能改变，则称为“不可变对象”，反之，如果所包含的值可以改变，则称为“可变对象”。
•前面章节已经学过的int、float等数值类型的值都是不可变对象，而列表、集合等值都是可变对象。
•值得一提的是，当可变对象有多个引用名时，通过一个引用名对对象的修改，对另一个引用名而言也是可见的，因为它们操作的是同一个对象。

7.2.1 对象

举例如下：

```
>>> a = 34
>>> b = a  # a和b是同一个对象34的不同名字，二者都指向对象34
>>> b = 45  #将名字b重新指派给了对象45（并不是改变b指向对象的值，因为b指向的是不可变对象）
>>> a    # a指向的对象保持不变
34
>>> a is b  # a和b已经指向不同的对象了
False
>>> c = [1,2,3]
>>> d = c  #c,d是同一个列表对象[1,2,3]的不同名字，并没有创建了一个新对象
>>> d[0] = 100  #通过名字d修改了列表对象
>>> c   # c和d指向的还是同一个对象
[100, 2, 3]
>>> c is d
True
```

7.2.1 对象

Python用关键字None表示空对象。没有显式返回值的函数都默认返回None对象。
对于一个给定的对象，可以用“.”运算符调用其属性或方法，使用内置函数dir()可以查看对象所支持的所有属性与方法，该函数将对象的所有属性名和方法名以一个列表的形式返回。实例如下：

```
>>> a = 12
>>> dir(a)  #查看整数对象的属性与方法，下面省略了部分返回内容
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
'__dir__', '__divmod__', '__doc__', … … 'bit_length', 'conjugate', 'denominator',
'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
>>> a.bit_length()  #调用对象的bit_length()方法，该方法返回整数的二进制位数
4
>>> dir('hello')  #查看字符串对象的属性与方法，下面省略了部分返回内容
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', … … 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

7.2.2 类

- Python将所有对象分成了不同的类别，这个类别就称为对象所属的“类”，或称为“类型”。
- Python有很多内置的类型，每一个类型都有一个名称，例如在前面章节已经学过的int、float等数值类型以及列表、集合等序列类型。
- 有了类型的概念，就可以用类型来实例化出一个该类型的具体对象，相应的语法为“类型名(参数列表)”。
- 可以使用内置函数type()来查看一个对象所属的类型。

7.2.2 类

举例如下：

```
>>> i = int(123)  #通过类型名int实例化对象，等效于i = 123
>>> type(i)  #type()函数返回对象所属的类型对象
<class 'int'>   # i属于int类，表示整数类型
>>> type("Hello World")
<class 'str'>  # "Hello World"属于str类，表示字符串类型
>>> type([7,8,9])
<class 'list'>  # [7,8,9]属于list类，表示列表类型
>>> def fun():
        print("Hello World")
>>> type(fun)
<class 'function'> # fun属于函数类型
```

7.2.2 类

既然Python里一切都是对象，那么对象所属的类型本身也是对象，例如上面提到的int、str、list及function等，都是对象。既然是对象，那也就有相应的类型。在Python里，所有的类型对象都属于名为type的类型。举例如下：

```
>>> type(int)
<class 'type'>
>>> type(str)
<class 'type'>
>>> type(list)
<class 'type'>
>>> type(type(fun))
<class 'type'>
```

7.2.2 类

•需要指出的是，Python的官方文档中关于“类型”有两个对应的词，分别是“class”和“type”。

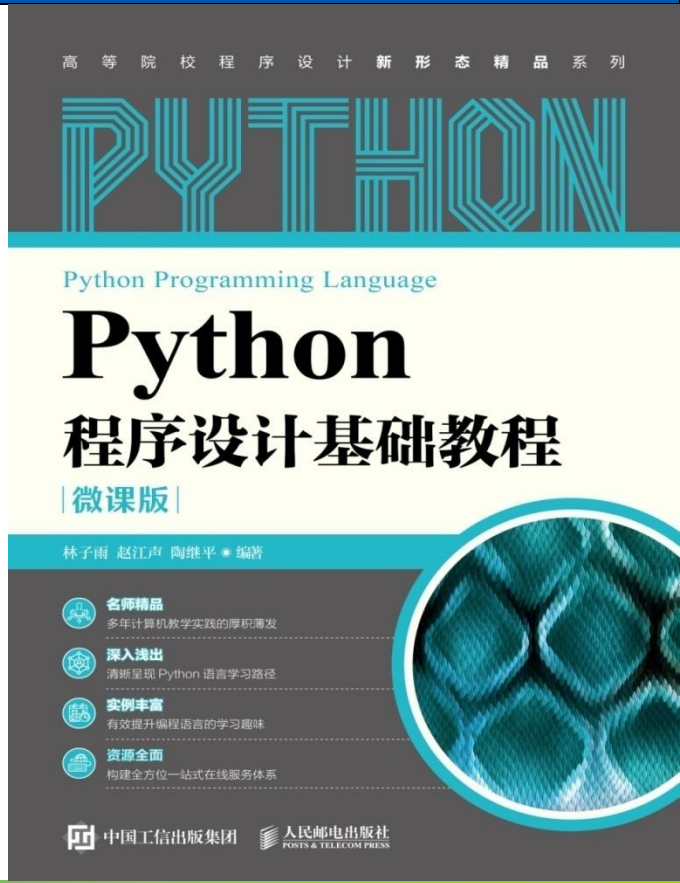
•在Python 3中，当表示类型这个概念时，这两个词之间可以互换使用；当分别作为自定义类的关键字和类型对象所属的类名时，不能互换。

7.2.2 类

•Python具有丰富的内建类型系统。
•简要概括起来，可以分为三大类别。
•第一类是用于表示数据的内建类型，例如int、str等基本数据类型以及诸如list及set等序列类型；
•第二类是用于表示程序结构的内建类型，例如函数对象的类型、type类型，还有后文将提到的object类型（表示所有类型的默认父类型）；
•第三类是用于表示Python解释器内部相关操作的类型，例如，types.TracebackType表示的是异常出现时的回溯对象的类型，它记录了异常发生时的堆栈调用等信息。
•对于大部分使用者而言，我们使用最多的就是表示数据的第一类内建类型。

7.3 自定义类

7.3.1 类的定义与实例化
7.3.2 构造器
7.3.3 类属性与实例属性



本PPT是如下教材的配套讲义：
《Python程序设计基础教程（微课版）》

厦门大学 林子雨,赵江声,陶继平 编著，人民邮电出版社

《 Python程序设计基础教程（微课版） 》教材官方网站：
http://dblab.xmu.edu.cn/post/python

7.3.1 类的定义与实例化

像大多数高级语言一样，Python使用关键词class来定义一个类。基本语法如下：

```
class 类名：
    """类文档字符串"""
    类的实现
```

7.3.1 类的定义与实例化

对于类的定义，有如下规定：

•类名必须是一个合法的Python标识符，且按照Python的编码规范，类名的首字母要求大写。当然，也可以根据自己或团队的代码规范来命名，基本原则是保证项目的代码规范一致性。

•类的文档字符串是位于类体最前面的、一个由三引号包括起来的字符串，作为类的帮助文档。定义好后可以使用类的__doc__属性来获取该字符串。在一些集成开发环境中，在实例化类时会自动提示这个文档字符串。

7.3.1 类的定义与实例化

- 类的实现部分包括属性和方法的定义。
- 类实现部分的每一条定义语句相对class关键字都必须有相同的缩进。
- 类属性的定义就是将变量的定义移到类的内部，只需要给出相应的变量名和初始值。
- 方法的定义语法与函数一样，也是使用def关键字，相当于定义在类里面的函数。
- 但是，与类外的函数不同的是，类方法的第一个参数都是指向调用者实例的引用，在定义时一般都习惯用self作为参数名（也可以使用其它标识符作为参数名）。
- 属性和方法统称为类的成员。

7.3.1 类的定义与实例化

下面是一个简单的自定义类：

```
01  class MyFirstClass:
02      """A simple example class"""
03      state = 12345
04      def fun(self):
05          return 'Hello World'
```

上面的实例代码定义了一个名为MyFirstClass的类，它包含了一个属性state和一个方法fun()。定义了一个类后，就可以使用类名加括号的形式生成一个该类的对象。一旦实例化成功后，相应的实例可以使用“.”操作符来访问类的成员。

7.3.1 类的定义与实例化

下面的例子中，首先实例化了一个名为c的MyFirstClass类型的对象，然后依次访问了属性state和方法fun()。在方法的访问中，不需要提供第一个self参数，相应的调用者对象c自动被绑定到了self。

```
>>>  MyFirstClass.__doc__  #返回类的文档字符串
'A simple example class'
>>> c = MyFirstClass ()      #实例化对象
>>> c.state
12345
>>> c.fun()
Hello World
```

7.3.1 类的定义与实例化

在类的内部，成员之间的相互访问也需要使用“.”操作符，而不能直接用成员名进行访问。举例如下：

```
>>> class MyTestClass:
        value = 456
        def fun1(self):
            print("My value is ", self.value)  #不能省略self直接访问value
        def fun2(self):
            fun1()  #执行时将出现NameError
>>> a = MyTestClass()
>>> a.fun1()
My value is 456
>>> a.fun2()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
 File "<stdin>", line 7, in fun2
NameError: name 'fun1' is not defined
```

7.3.2 构造器

•当实例化一个类时，Python会自动调用一个名为“__new__”的方法，创建一个最原始的对象，该方法继承自父类object。

•有了这个原始对象后，再使用该对象调用一个名为“__init__”的特殊方法进行对象的初始化。

•这些方法并不需要由用户显式调用，而是在实例化类时自动被调用，称为“类的构造器（constructor）方法”。

•在实际应用中，用户一般不需要重写“__new__”方法，只需要重新实现“__init__”方法来执行一些具体的初始化操作。

7.3.2 构造器

“__init__”方法可以有参数，这些参数将在实例化时被提供，当然也可以像普通函数一样，给这些参数提供默认值。实例化时只需要将相应的值放在类名后的括号里面。例如：

```
>>> class Student:
        def __init__(self,name="无名氏"):
            print("开始实例化一个Student对象,名为",name)
>>> a = Student('张三')
开始实例化一个Student对象,名为张三
>>> b = Student()
开始实例化一个Student对象,名为无名氏
```

7.3.3 类属性与实例属性

•Python的类定义里有两种属性，分别称为“类属性”和“实例属性”。
•类属性在类内部的所有方法之外进行定义。
•例如，上一节的state就属于MyFirstClass的类属性。
•类属性是由类及类的所有实例共用的，可以通过类名或实例名进行访问。
•在C++及Java等一些高级语言中，将这种由类的所有实例所共用的数据成员称为“静态（static）成员”。
•Python没有沿用这一说法，而是改用“类属性”来表述。

•除了类属性，Python还有实例属性的概念。
•与类属性不同的是，实例属性是由一个具体的实例所独有的，不同实例的实例属性之间是完全独立的。
•实例属性一般在构造器“__init__”方法里面初始化。

7.3.3 类属性与实例属性

举例如下：

```
>>> class Car:
        wheels = 4      # wheels是类属性
        def __init__(self, owner):    #构造器方法
            self.owner = owner    # owner是实例属性
>>> a = Car('Mike')
>>> b = Car('Tom')
>>> a.owner,b.owner   #通过实例访问实例属性
('Mike', 'Tom')
>>> a.wheels,b.wheels,Car.wheels  #通过实例或类访问类属性
(4, 4, 4)
```

7.3.3 类属性与实例属性

•得益于Python的动态类型特性，除了在类的内部进行定义，类属性和实例属性都可以在使用时动态添加。

•使用“类名.新的类属性名 = 初始值”来动态添加类属性，添加成功后，类的所有实例都拥有这个新增加的类属性。

•使用“类的实例.新的实例属性名 = 初始值”来动态添加实例属性，添加成功后，该实例属性只属于相应的实例，类的其它实例并没有相应的实例属性。

7.3.3 类属性与实例属性

接着上面的代码，举例如下：

```
>>> Car.brand = 'BYD'   #增加一个新的类属性brand
>>> a.brand,b.brand
('BYD', 'BYD')
>>> a.nickname =  'Rocket'   #给实例a增加一个新的实例属性nickname
>>> a.nickname
'Rocket'
>>> b.nickname   #实例b并没有实例属性nickname，属于非法访问
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Car' object has no attribute 'nickname ‘
```

7.3.3 类属性与实例属性

•实例属性是属于一个具体的实例所独有，对其修改当然不会影响到其它实例。

•类属性属于类和该类实例化出来的所有实例。

•类可以通过“.”运算符读取或修改类属性。

•类实例可以读取类属性，但不能直接修改类属性。

•当类实例企图用“.”运算符直接修改类属性时，实际发生的是，为实例对象动态地添加了一个与类属性同名的实例属性，这时如果还需要访问同名的类属性，就需要用到名为“__class__”的特殊类属性，该属性将返回实例所属的类对象，这样，可以使用“__class__”的返回值间接修改类属性。

7.3.3 类属性与实例属性

接着上面的代码，举例如下：

```
>>> a.owner = 'Alice'   #实例属性的修改不会影响其它实例
>>> a.owner,b.owner
('Alice', 'Tom')
>>> Car.wheels = 2   #通过类修改类属性，修改值对所有实例可见
>>> a.wheels,b.wheels,Car.wheels
(2, 2, 2)
>>> a.wheels = 3   #给实例a动态添加了一个同名的实例属性，不会影响类属性
>>> a.wheels,b.wheels,Car.wheels
(3, 2, 2)
>>> a.wheels,a.__class__.wheels   #分别访问同名的实例属性和类属性
(3, 2)
```

7.3.3 类属性与实例属性

•需要指出的是，上面的例子仅仅是为了解释类属性和实例属性的使用区别，在实际应用中，为了增强代码的可读性并减少可能出现的漏洞，要尽量避免使用同名的类属性和实例属性。

•在创建一个类时，Python会自动创建一个名为“__dict__”的特殊实例属性，该属性是一个字典对象，它保存了实例的所有其它实例属性名及其相应的值。另外，还可以使用del关键词删除类属性或者实例属性。

7.3.3 类属性与实例属性

接着上面的代码，举例如下：

```
>>> a.__dict__    #a实例包含3个实例属性，后两个属性都是动态添加的
{'owner': 'Alice', 'wheels': 3, 'nickname': 'Rocket' }
>>>        del a.wheels  #删除a实例的wheels属性
>>> a.__dict__    #现在a实例只包含2个实例属性
{'owner': 'Alice', 'nickname': 'Rocket'}
>>> b.__dict__   #b实例只包含一个实例属性
{'owner': 'Tom'}
```

7.3.3 类属性与实例属性

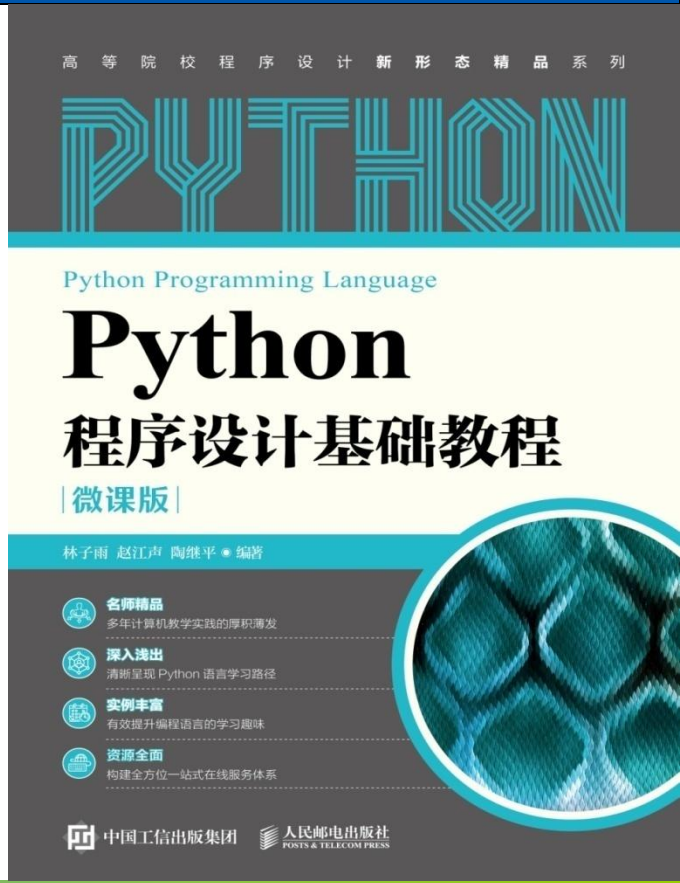
•需要说明的是，类属性和实例属性这两个概念在Python的官方文档里对应的是“class variables”和“instance variables”，因此有些中文文档里采用了类变量和实例变量作为对应词，本课程将遵循面向对象语言中的一般术语，统一将类的数据成员称为“属性”。类属性和实例属性是初学者容易混淆的概念，简单总结如下：

•类属性是由类及类的所有实例所共有的，实例属性是实例所独有的；

•实例属性会覆盖同名的类属性，这种使用会增加调试困难，同时可能会带来潜在的漏洞，因此要尽量避免。

7.4 成员的可见性

7.4.1 公有成员与私有成员
7.4.2 保护类型成员
7.4.3 property类



本PPT是如下教材的配套讲义：
《Python程序设计基础教程（微课版）》

厦门大学 林子雨,赵江声,陶继平 编著，人民邮电出版社

《 Python程序设计基础教程（微课版） 》教材官方网站：
http://dblab.xmu.edu.cn/post/python

7.4.1 公有成员与私有成员

•在面向对象编程中，将在类的内外都能访问的成员称为“公有（public）成员”。

•与公有成员对应的概念是“私有（private）成员”，私有成员只能在类的内部进行访问。

•通过设置私有成员，可以将类的相关信息隐藏起来，对外只保留必要的访问接口。

7.4.1 公有成员与私有成员

•Python并没有像C++等高级语言那样用特定的关键字来严格定义成员的可见性，而是采用了约定的命名规范来表示成员是公有还是私有。

•当一个成员的名称以至少两个下划线开头，且结尾至多有一个下划线时，则表明该成员是私有成员，只能在类的内部访问，而不能在类的外面通过实例对象直接访问。

•但是，Python没有严格意义下的私有成员，它实际上是通过一种称为“命名改写（name mangling）”的手段实现名义上的隐藏，这些私有成员对外的实际名字被附加了一个由下划线和类名所构成的前缀（即“_类名”），这样，用户仍然可以通过这个改写后的名字访问到私有成员。

7.4.1 公有成员与私有成员

```
>>> class MyClass():
        def __init__(self):
            self.__private_value = 123   #私有属性
        def fun(self):
            self.__foo()   #类内部可以直接访问私有方法
            print(self.__private_value)  #类内部可以直接访问私有属性
        def __foo(self):    #私有方法
            print('Hello World')
>>> a = MyClass()
>>> a.fun()  #类外部可以直接访问公有方法
Hello World
123
```

7.4.1 公有成员与私有成员

```
>>> a.__private_value   #类外面不能直接访问私有属性
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass' object has no attribute '__private_value'
>>> a._MyClass__private_value   #通过改写后的名字访问私有属性
123
>>> a.__foo() # 类外面不能直接访问私有方法
Traceback (most recent call last):
File "< stdin >", line 1, in <module>
AttributeError: 'Student' object has no attribute '__foo'
>>> a._MyClass__foo() # 通过改写后的名字访问私有方法
Hello World
```

7.4.2 保护类型成员

•针对类的成员，除了这种双下划线开头的命名约定，Python还有单下划线开头的命名约定，表示“保护（protected）”类型的成员。

•在面向对象编程术语里，保护型成员是指那些只能在类及其派生类内部进行访问的成员。

•从这个要求看，Python没有真正意义下的保护型成员（没有硬性保护）。

•Python中的保护型成员实质上和公有型成员没有任何区别，在类的内外都能直接访问，而且可以被子类继承。

•Python仅仅是用这种命名方式提醒用户，该成员具有特殊的作用（应该被保护），不要在类及其派生类的外部修改甚至读取它的值。

7.4.3 property类

当直接用“.”运算符对实例的公有实例属性进行读写时，一个明显的缺点是不能对读写进行额外的控制。由于Python的动态类型特性，如果出现了非法的赋值，并不会在赋值的时候提示错误，但是后续的运算可能就会出现类型或者逻辑错误，因此，这种直接读写的方式可能会给程序埋下潜在的漏洞。例如，下面的实例将Student类的出生年属性设置成公有，但修改时不小心多输入了一个0，但程序不会提示任何错误，这显然是不合理的。

```
>>> from datetime import date
>>> class Student:
        def __init__(self,name,year):
            self.name = name
            self.birthyear = year
 >>> mike = Student("Mike",19820)
>>> mike.birthyear = 19820
```

7.4.3 property类

•在面向对象的编程里，为了在读写属性时增加其它的诸如合法性检查的额外操作，并且提高程序的可扩展性，一般都不直接读写属性，而是设置相应的读和写方法。

•Python提供了一个名为property的类来实现这些功能。

•在定义类时，将需要设置特别读写的实例属性设置为私有成员，并提供相应的读写方法来完成一些额外的操作，然后再使用这些读写方法来实例化一个property类型的类属性（名称通常取私有实例属性名双下划线后面的部分）。

•在使用过程中，用户只对这个property属性进行读写，Python将自动访问相应的读写方法来实现实例属性的合法读写。

7.4.3 property类

```
>>> class Student:
        def __init__(self,name):
            self.__name = name
            self.__score = 0
        def getname(self):
            print("调用getname方法")
            return self.__name
        def getscore(self):
            print("调用getscore方法")
            return self.__score
        def setscore(self,score):
            print("调用setscore方法")
            if score>=0 and score<=100:
                self.__score = score
            else:
                raise ValueError("错误的参数值") #出现非法参数时抛出一个异常
        score = property(getscore,setscore)
        #第一个参数为读方法，第二个参数为写方法， 用于控制可读写属性
        name = property(getname) #只提供读方法，用于控制只读属性
```

7.4.3 property类

```
>>> a = Student('Mike')
>>> a.score = 50   #相当于执行a.setscore(50)
调用setscore方法
>>> a.score = 500   #相当于执行a.setscore(500)
调用setscore方法
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 16, in setscore
    raise ValueError("错误的参数值")
ValueError: 错误的参数值
>>> a.score # 相当于执行a.getscore()
调用getscore方法
50
>>> a.name  #相当于执行a.getname()
调用getname方法
'Mike'
>>> a.name = 'John'  #对只读属性进行写操作，返回错误
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

7.4.3 property类

•为了简化property对象的构造，避免使用实例中诸如getscore及setscore等额外的读写方法名，Python提供了“装饰器（decorator）”语法糖来进一步简化上述代码，只需要直接用对外暴露的property对象名来定义同名的读写方法，并在读方法前加上@property，在写方法前加上@[property对象名].setter。

•针对上面的Student类，采用装饰器语法可以改写成如下形式：

7.4.3 property类

```
>>> class Student:
        def __init__(self,name):
            self.__name = name
            self.__score = 0
        @property
        def name(self):
             print("调用name的读方法")
             return self.__name
        @property
        def score(self):
            print("调用score的读方法")
            return self.__score
        @score.setter
        def score(self,score):
            print("调用score的写方法")
            if score>=0 and score<=100:
                self.__score = score
            else:
                raise ValueError("错误的参数值“)
```

7.4.3 property类

```
>>> a = Student('Mike')
>>> a.score = 50
调用score的写方法
>>> a.score = 500
调用score的写方法
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 16, in setscore
    raise ValueError("错误的参数值")
ValueError: 错误的参数值
>>> a.score
调用score的读方法
50
>>> a.name
调用name的读方法
'Mike'
>>> a.name = 'John'   #对只读属性进行写操作，返回错误
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

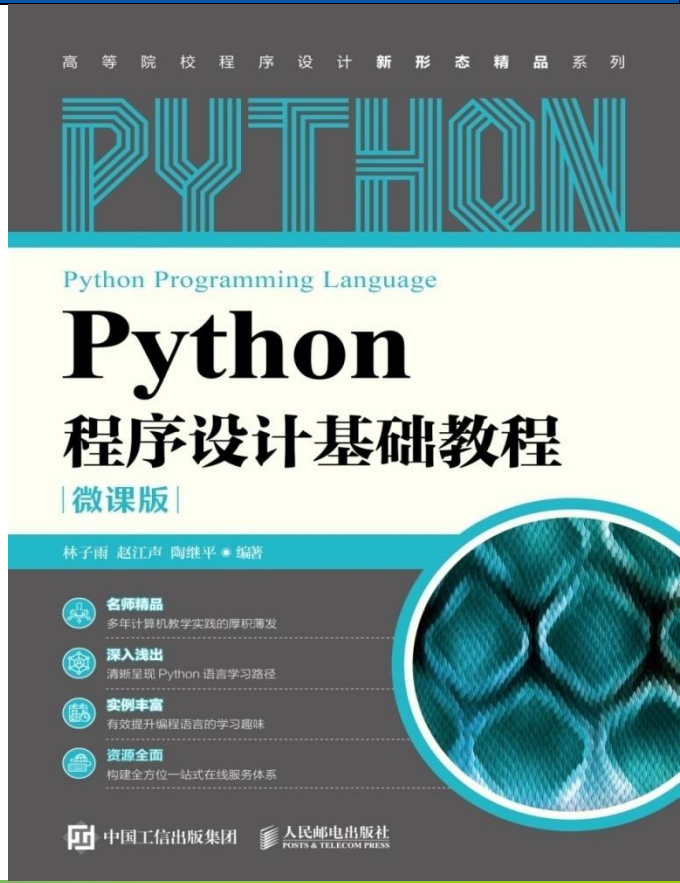
7.5 方法

7.5.1 类方法
7.5.2 静态方法
7.5.3 魔法方法

本PPT是如下教材的配套讲义：
《Python程序设计基础教程（微课版）》

厦门大学 林子雨,赵江声,陶继平 编著，人民邮电出版社

《 Python程序设计基础教程（微课版） 》教材官方网站：
http://dblab.xmu.edu.cn/post/python



7.5.1 类方法

•在调用实例方法之前，必须有一个相应类型的实例。但在有些场景下，我们可能需要在没有实例的情况下通过类本身来调用一些方法。

•Python提供了名为“类方法（classmethod）”的概念来实现这一功能。

•为了将一个方法定义为类方法，只需要在方法定义前加上内置的“@classmethod”装饰器。

•与实例方法类似，类方法的第一个参数也是自动传入的，但该参数是对类本身的引用，按照惯例，在定义时一般将该参数命名为cls。

•类方法可以通过类或者类的实例进行调用，Python会自动将相应的类对象作为第一参数传入。

7.5.1 类方法

举例如下：

```
>>> class TestClassMethod:
        value = 4    #value是类属性
        def __init__(self):
            self.name = 'something'   #name是实例属性
        @classmethod  #定义类方法
        def fun(cls):   #类方法中只能调用类属性或其它类方法
            print("calling a class method...")
            print("TestClassMethod.value=",cls.value)
            # print(cls.name)       #如果不注释该句，执行下面的测试语句将提示AttributeError
```

错误。原因是，不管是通过类还是实例调用，传入的都是类对象，因此不能访问实例属性

```
>>> TestClassMethod.fun()
calling a class method...
TestClassMethod.value= 4
>>> a = TestClassMethod()
>>> a.fun()
calling a class method...
TestClassMethod.value= 4
```

7.5.1 类方法

•类方法最常见的作用是充当辅助构造器的角色来创建实例。

•前文已经提到，我们在实例化一个类时是通过调用构造器方法“__init__”完成初始化工作。

•但有时候我们希望通过传入不同的参数形式进行实例化。例如，有一个成员Member类，其包含一个名为age的年龄属性，正常可以通过直接传入年龄进行初始化，但作为一个对外设计友好的类，应该也可以通过传入出生年进行初始化。

•为了实现这个需求，可以定义一个接受出生年作为参数的类方法，在这个类方法里计算出年龄后再显式调用构造器方法返回相应的实例。

•使用时，用户可以根据自己提供的参数选择相应的实例化方法。

7.5.1 类方法

具体实现如下：

```
01  # -*- coding: utf-8 -*-
02  # testclassmethod.py
03  from datetime import date
04  class Member:
05      def __init__(self,name,age):
06          self.name = name
07          self.age = age
08      @classmethod
09      def from_birthyear(cls,name,birthyear):
10          age = date.today().year - birthyear
11          return cls(name,age) # 通过类对象实例化一个该类的对象
12      def hello(self):
13          print('I am %s. I am %d years old'%(self.name,self.age))
```

7.5.1 类方法

在IDLE中打开testclassmethod.py，按一下键盘的“F5”键运行代码，然后，可以在解释器中继续执行如下代码：

```
>>> mike = Member('Mike',23)
>>> john = Member.from_birthyear('John',1980)
>>> mike.hello()
I am Mike. I am 23 years old
>>> john.hello()
I am John. I am 41 years old
```

7.5.2 静态方法

•除了实例方法和类方法外，还有一种方法，它仅仅是将一个普通函数的定义移到了类的内部，因此需要用类或者类的实例进行调用，但是调用时不会隐式传入调用者信息。

•之所以需要这类方法，通常的原因是，这个函数在业务逻辑上只与这个类相关，因此，为了保持代码的层次整洁，不希望把它作为一个全局函数定义在类外；同时，当作为实例或类方法定义在类的内部时，它的功能又与调用者（类或者类的实例）无关，因此，为了减少隐式传入参数的开销，无需将调用者作为第一参数隐式传入。

•这类方法称为“静态方法”，定义时需在方法前加上内置的“@staticmethod”装饰器。

7.5.2 静态方法

举例如下：

```
>>> class A:
        @staticmethod
        def hello(s):   #第一个参数就是普通显式参数，不是调用者
            print('Hello',s)
>>> a = A()
>>> A.hello('Mike') # 静态方法一般用类调用
Hello Mike
>>> a.hello('Mike') # 静态方法也可以用类实例调用
Hello Mike
```

7.5.3 魔法方法

•如果一个方法采用双下划线开头和双下划线结尾，Python将其视为一种特殊方法，具有特定的调用约定。

•这类方法一般不是被用户直接调用，而是按照某种约定被自动地间接调用。Python官方文档中将这类自动被调用的方法称为“魔法方法（magic method）”。

•例如前文提到的“__init__”就是一个常用的魔法方法。魔法方法主要用于对象的构造、运算符重写及访问控制等。

7.5.3 魔法方法

1.__str__方法

该魔法方法在所有类的基类object中定义，用于返回类实例的字符串表示，内置的函数str()就是通过调用该方法实现任意对象向字符串的转换。可以为自定义类重写该方法，返回可读性更好的字符串。

7.5.3 魔法方法

```
>>> class Point1:  # 没有重写__str__方法
        def __init__(self,x,y):
            self.x = x
            self.y = y
>>> class Point2:
        def __init__(self,x,y):
            self.x = x
            self.y = y
        def __str__(self):    #重写__str__方法
            return 'Point('+str(self.x)+','+str(self.y)+')'
>>> a,b = Point1(3,4),Point2(3,4)
>>> str(a)
'<__main__.Point1 object at 0x0000021B9996A550>'
>>> str(b)
'Point(3,4)'
```

7.5.3 魔法方法

2.__eq__方法

该方法也在object类中定义，用于判断类的两个实例是否相等。比较运算符“==”就是调用了该方法。object中“__eq__”的默认实现为引用比较，即比较两个实例对象的整数身份号是否相等。如果要实现内容值的比较，必须重写该方法。

```
>>> class Point1:  #没有重写“__eq__”方法
        def __init__(self,x,y):
            self.x = x
            self.y = y
>>> class Point2:
        def __init__(self,x,y):
            self.x = x
            self.y = y
        def __eq__(self,other): # 重写“__eq__”方法，用内容值进行比较
            return self.x==other.x and self.y==other.y
```

7.5.3 魔法方法

```
>>> a1,b1=Point1(3,4),Point1(3,4)
>>> a1==b1    #默认比较id(a1)与id(b1)
False
>>> a2,b2=Point2(3,4),Point2(3,4)
>>> a2==b2   #调用重写的“__eq__”方法
True
>>> a3,b3=Point2(3,4),Point2(4,5)
>>> a3==b3
False
```

7.5.3 魔法方法

类似于__eq__方法是对比较运算符“==”的重写，Python为其它的比较运算符及算术运算符都提供了相应的魔法方法。表7-1列出了常用的运算符魔法方法，用户可以根据需要进行重写。

表7-1 常用运算符对应的魔法方法

魔法方法签名	运算符使用样式
__lt__(self,other)	x<y
__le__(self,other)	x<=y
__gt__(self,other)	x>y
__ge__(self,other)	x>=y
__eq__(self,other)	x==y
__ne__(self,other)	x!=y
__add__(self,other)	x+y
__sub__(self,other)	x-y
__mul__(self,other)	x*y
__truediv__(self,other)	x/y
__floordiv__(self,other)	x//y
__mod__(self,other)	x%y
__pow__(self,other)	x**y

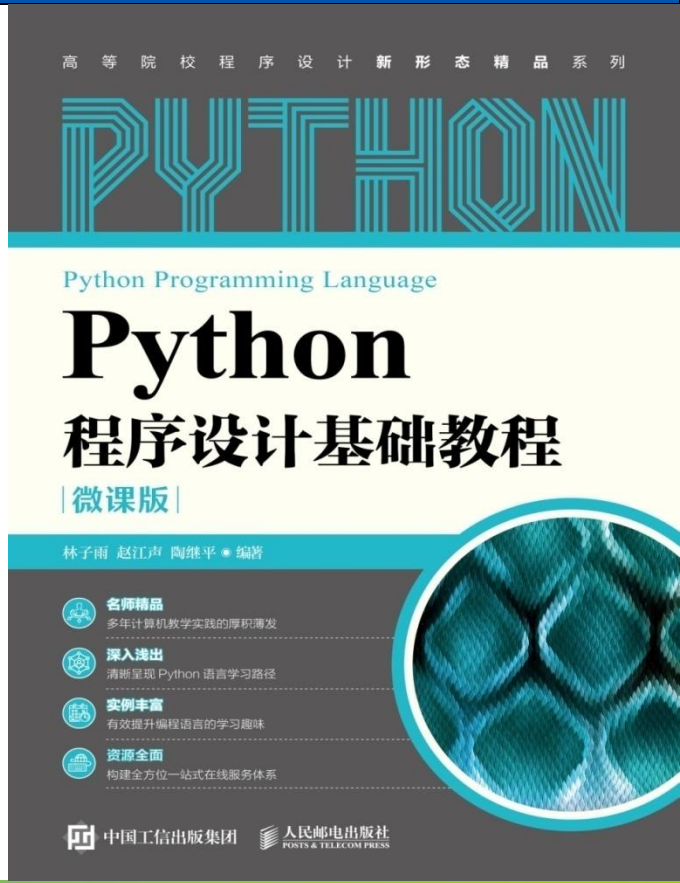
7.6 类的继承

7.6.1 继承
7.6.2 多态

本PPT是如下教材的配套讲义：
《Python程序设计基础教程（微课版）》

厦门大学 林子雨,赵江声,陶继平 编著，人民邮电出版社

《 Python程序设计基础教程（微课版） 》教材官方网站：
http://dblab.xmu.edu.cn/post/python



7.6.1 继承

为了实现类的继承，只需要在自定义类名后添加一个括号，并将需要继承的父类名放在括号里。除了拥有自定义的属性和方法外，子类将自动继承父类的所有成员。具体规则如下：

- 子类可以直接访问父类的公有成员和保护类型成员，也可以重写父类的公有和保护类型成员；

- 子类不能直接访问父类的私有成员，但可以通过父类名前缀间接访问，由于私有成员的命名改写（name mangling）规则，子类不会重写父类的私有成员；

7.6.1 继承

•如果子类没有实现构造器方法，则实例化子类时将自动调用父类的构造器方法，如果子类重写了构造器方法，则必须显式调用父类的构造器方法，这将用到内建的super()函数，该函数返回一个临时的父类对象；

•当对一个对象调用某个成员时，如果该对象所属的类没有定义该成员，Python将自动在对象的父类中依次查找，直到找到该成员，如果一直都没有找到，则会提示AttributeError异常。

7.6.1 继承

下面是一个具体实例：

```
01  # -*- coding: utf-8 -*-
02  # inheritance.py
03  class A:
04      def __init__(self):
05          self.public_value_1 = 'pulic value 1 in class A'
06      def public_method(self):
07          print('calling public method in class A')
08      def __private_method(self):
09          print('calling private method in class A')
10  class B1(A):
11      pass
```

7.6.1 继承

```
12  class B2(A):
13      def __init__(self):  #重写构造器，但没有显式调用父类的构造器
14          self.public_value_2 = 'pulic value 2 in class B2'
15      def public_method(self):  #重写父类公有方法
16          print('calling public method in class B2')
17      def __private_method(self):  #定义自己的私有方法
18          print('calling private method in class B2')
19  class B3(A):
20      def __init__(self):  #重写构造器，并显式调用父类的构造器
21          super().__init__()
22          self.public_value_2 = 'pulic value 2 in class B3'
```

7.6.1 继承

在IDLE中打开inheritance.py，按一下键盘的“F5”键运行代码，然后，可以在解释器中继续执行如下代码：

```
>>> b1,b2,b3 = (B1(),B2(),B3())
>>> b1.public_value_1   #由于B1没有定义构造器，父类构造器自动被调用，父类的公有属性被初始化并被子类继承
'pulic value 1 in class A'
>>> b1.public_method()   #直接调用继承自父类的公有方法
calling public method in class A
>>> b1.__private_method()  # 子类不能直接访问父类的私有方法
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    b1.__private_method()
AttributeError: 'B1' object has no attribute '__private_method'
>>> b1._A__private_method()   #用父类名前缀间接调用父类的私有方法
calling private method in class A
>>> b2.public_value_2   #调用子类新增的属性
'pulic value 2 in class B2'
```

7.6.1 继承

```
>>> b2.public_value_1   #由于子类B2没有显示调用父类的构造器，父类属性没有被初始化，将提示属性错误
Traceback (most recent call last):
 File "<pyshell>", line 1, in <module>
AttributeError: 'B2' object has no attribute 'public_value_1'
>>> b2.public_method()   #父类的公有方法被重写了
calling public method in class B2
>>> b2._B2__private_method()   #采用子类名前缀间接调用子类的私有方法
calling private method in class B2
>>> b2._A__private_method()   #采用父类名前缀间接调用父类的私有方法
calling private method in class A
>>> b3.public_value_1   #由于子类B3显式调用了父类的构造器，因此父类的公有属性被初始化并被子类继承
'pulic value 1 in class A'
```

7.6.1 继承

•Python允许多重继承，即一个类可以有多个直接父类，定义时只需将不同的父类名放在类名后的括号内，并用逗号隔开。如果每个类都只有一个直接父类，当对一个对象调用某个成员时，Python将自动在对象所属的类及其父类中依次查找，直到找到该成员。

•但在多重继承的情形下，如果某个同名成员出现在多个父类型中，则需要有一个规则来确定沿着哪条路径进行查找，感兴趣的读者可以参考相关文档了解相应规则。

•Python提供了一个名为object的类。如果一个类在定义时没有指明父类，则其直接父类为object类，也就是说，Python的任何一个类都直接或间接派生自object类。object类提供了很多魔法方法的默认实现，包括7.5节中提到的“__str__”和“__eq__”，这两个方法分别实现了对象的字符串表示和相等比较，还有一个常用的方法“__hash__”，其返回对象的哈希值。

7.6.1 继承

```
>>> class DefaultObj:    #等效于 class DefaultObj(object):
        pass
>>> a = DefaultObj()
>>> b = DefaultObj()
>>> a.__str__()
'<__main__.DefaultObj object at 0x000001C2FE75EEF0>'
>>> a == b
False
>>> a.__hash__()
121062776559
>>> b.__hash__()
-92233719157912394 31
```

7.6.1 继承

•一旦一个类直接或间接继承了另外一个类，子类的实例也是父类的实例，可以通过内建的isinstance()函数和issubclass()函数来查看这种对象及类之间的继承关系，前者用于查看一个实例对象是否属于某个类的实例，后者用于查看一个类是否属于另一个类的直接或间接子类。

•在下例中，类A继承自类B，类B继承自类C，类C没有显式指定父类，默认继承自object。

7.6.1 继承

```
>>> class C: # 等效于 class C(object):
        pass
>>> class B(C):
        pass
>>> class A(B):
        pass
>>> issubclass(C,object)
True
>>> issubclass(B,object),issubclass(B,C)
(True, True)
>>> issubclass(A,object),issubclass(A,B),issubclass(A,C)
(True, True, True)
>>> a = A()
>>> isinstance(a,A),isinstance(a,B),isinstance(a,C),isinstance(a,object)
(True, True, True, True)
```

7.6.2 多态

在面向对象编程中，多态一般指的是用一个父类的方法在不同的子类中有不同的具体实现。Python完全支持多态行为，不同子类只需要重写父类的同名方法，实例在调用被重写的方法时，将根据实际的子类类型选择相应子类的方法。

```
01  # -*- coding: utf-8 -*-
02  # test_polymorphism1.py
03  class Animal:
04      kind = "动物"  # 该属性将被子类重写
05      def show(self):
06          print('我是'+self.kind,',我的叫声是',sep="",end="")
07          self.yell() # 将根据传入的具体子类对象调用子类的重写方法
08      def yell(self): # 该方法将被子类重写
09          print("???")
10  class Dog(Animal):
11      kind = "狗"
12      def yell(self):
13          print("汪汪汪.“)
```

7.6.2 多态

```
14  class Cat(Animal):
15      kind = "猫"
16      def yell(self):
17          print("喵喵喵.")
18  class Duck(Animal):
19      kind = "鸭子"
20      def yell(self):
21          print("嘎嘎嘎.")
22  if __name__ == '__main__':
23      animals = [Dog(),Cat(),Duck()]
24      for animal in animals:
25          animal.show()  #传入show方法的self参数为不同的子类对象
```

上面代码的运行结果如下：

```
我是狗,我的叫声是汪汪汪.
我是猫,我的叫声是喵喵喵.
我是鸭子,我的叫声是嘎嘎嘎.
```

7.6.2 多态

•从这个实例也可以看出，Python的多态行为本质是由其动态类型特性决定的。

•因此，不仅在类的继承上可以表现多态，在普通的函数定义上也可以表现多态行为，只要传入函数的对象具有相应的属性和方法，则在函数内部这些对象就会表现出相应的多态行为。

•在下面这个实例中，show_object()并不要求传入参数obj具有某个特定的类型，只要该类型具有kind属性和show()方法。

•后面的Person、Machine及Pig在类型层次上并没有继承关系，但都实现了kind属性和show()方法，因此其实例对象可以作为show_object()的参数，使得该函数表现出多态行为。

7.6.2 多态

```
01  # -*- coding: utf-8 -*-
02  # test_polymorphism2.py
03  def show_object(obj):  #只要传入的obj对象具有kind属性和show方法
04      print('我是'+obj.kind,',我的生活是',sep="",end="")
05      obj.show()
06  # 以下三个类都具有kind属性和show方法
07  # 因此相应实例可以作为show_object的参数
08  class Professor:
09      kind = "人"
10      def show(self):
11          print("吃饭、工作、睡觉.")
```

7.6.2 多态

```
12  class Machine:
13      kind = "机器"
14      def show(self):
15          print("工作、工作、工作.")
16  class Pig:
17      kind = "猪"
18      def show(self):
19          print("吃饭、睡觉.")
20  if __name__ == '__main__':
21      objs = [Professor(),Machine(),Pig()]
22      for obj in objs:
23          show_object(obj)
```

上面代码的运行结果如下：
我是人,我的生活是吃饭、工作、睡觉.
我是机器,我的生活是工作、工作、工作.
我是猪,我的生活是吃饭、睡觉.

7.6.2 多态

•需要强调的是，继承虽然可以有效地减少重复的代码，但继承已经破坏了对象的封装性。

•也就是说，父类的实现细节对于子类来说都是透明的。

•因此，在实际的项目开发中，我们不要滥用继承，需要确信使用继承确实是有效且可行的办法，关于面向对象设计的一些常用思想与规范原则，可以参考面向对象设计或软件工程方面的相关书籍。


Thank You!