桂林理工大学
GUILIN UNIVERSITY OF TECHNOLOGY

第六章

面向对象程序设计

第六章 面向对象程序设计

- 面向对象程序设计（Object Oriented Programming，OOP）的思想主要针对大型软件设计而提出，使得软件设计更加灵活，能够很好地支持代码复用和设计复用，代码具有更好的可读性和可扩展性，大幅度降低了软件开发的难度。
- 面向对象程序设计的一个关键性观念是将数据以及对数据的操作封装在一起，组成一个相互依存、不可分割的整体（对象），不同对象之间通过消息机制来通信或者同步。对于相同类型的对象（instance）进行分类、抽象后，得出共同的特征而形成了类（class），面向对象程序设计的关键就是如何合理地定义这些类并且组织多个类之间的关系。
- Python是面向对象的解释型高级动态编程语言，完全支持面向对象的基本功能，如封装、继承、多态以及对基类方法的覆盖或重写。创建类时用变量形式表示对象特征的成员称为数据成员（attribute），用函数形式表示对象行为的成员称为成员方法（method），数据成员和成员方法统称为类的成员。

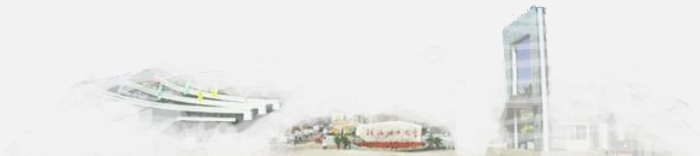
第六章 面向对象程序设计

- 6.1 类的定义与使用
- 6.2 数据成员与成员方法
- 6.3 继承、多态
- 6.4 特殊方法与运算符重载

6.1 类的定义与使用

- Python使用class关键字来定义类，class关键字之后是一个空格，接下来是类的名字，如果派生自其它基类的话则需要把所有基类放到一对圆括号中并使用逗号分隔，然后是一个冒号，最后换行并定义类的内部实现。
- 类名的首字母一般要大写，当然也可以按照自己的习惯定义类名，但是一般推荐参考惯例来命名，并在整个系统的设计和实现中保持风格一致，这一点对于团队合作非常重要。

```
class Car(object):                  #定义一个类，派生自object类
    def infor(self):                #定义成员方法
        print("This is a car")
```

6.1 类的定义与使用

- 定义了类之后，就可以用来实例化对象，并通过“对象名.成员”的方式来访问其中的数据成员或成员方法。

```
>>> car = Car()                    #实例化对象
>>> car.infor()                    #调用对象的成员方法
This is a car
```

6.1 类的定义与使用

- 在Python中，可以使用内置函数isinstance()来测试一个对象是否为某个类的实例，或者使用内置函数type()查看对象类型。

```
>>> isinstance(car, Car)
True
>>> isinstance(car, str)
False
>>> type(car)
<class '__main__.Car'>
```

6.2 数据成员与成员方法

- 创建类时用变量形式表示对象特征的成员称为数据成员（attribute），用函数形式表示对象行为的成员称为成员方法（method），数据成员和成员方法统称为类的成员。

6.2.1 私有成员与公有成员

- 私有成员在类的外部不能直接访问，一般是在类的内部进行访问和操作，或者在类的外部通过调用对象的公有成员方法来访问，而公有成员是可以公开使用的，既可以在类的内部进行访问，也可以在外部程序中使用。
- 从形式上看，在定义类的成员时，如果成员名以两个下划线开头但是不以两个下划线结束则表示是私有成员，否则就不是私有成员。
- Python并没有对私有成员提供严格的访问保护机制，通过一种特殊方式“对象名._类名__xxx”也可以在外部程序中访问私有成员，但这会破坏类的封装性，不建议这样做。

6.2.1 私有成员与公有成员

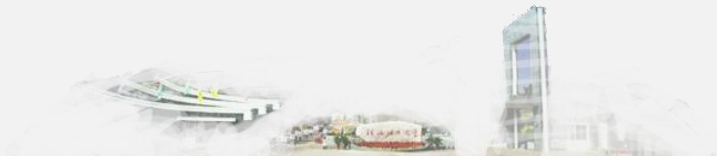
```
>>> class A:
    def __init__(self, value1=0, value2=0):
        self._value1 = value1
        self.__value2 = value2
    def setValue(self, value1, value2):
        self._value1 = value1
        self.__value2 = value2
    def show(self):
        print(self._value1)
        print(self.__value2)

>>> a = A()
>>> a._value1
0
>>> a._A__value2                    #在外部访问对象的私有数据成员
0
```

6.2.1 私有成员与公有成员

- 在Python中，以下划线开头的变量名和方法名有特殊的含义，尤其是在类的定义中。
  - _xxx：受保护成员；
  - __xxx__：系统定义的特殊成员；
  - __xxx：私有成员，只有类对象自己能访问，子类对象不能直接访问到这个成员，但在对象外部可以通过“对象名._类名__xxx”这样的特殊方式来访问。
- 注意：Python中不存在严格意义上的私有成员。

6.2.2 数据成员

- 数据成员可以大致分为两类：属于对象的数据成员和属于类的数据成员。属于对象的数据成员一般在构造方法__init__()中定义，当然也可以在其他成员方法中定义，在定义和在实例方法中访问数据成员时以self作为前缀，同一个类的不同对象（实例）的数据成员之间互不影响；属于类的数据成员是该类所有对象共享的，不属于任何一个对象，在定义类时这类数据成员一般不在任何一个成员方法的定义中。
- 在主程序中或类的外部，对象数据成员属于实例(对象)，只能通过对象名访问；而类数据成员属于类，可以通过类名或对象名访问。



6.2.2 数据成员

- 利用类数据成员的共享性，可以实时获得该类的对象数量，并且可以控制该类可以创建的对象最大数量。例如：

```
>>> class Demo(object):
    total = 0
    def __new__(cls, *args, **kwargs):      #该方法在__init__()之前被调用
        if cls.total >= 3:                  #最多允许创建3个对象
            raise Exception('最多只能创建3个对象')
        else:
            return object.__new__(cls)
    def __init__(self):
        Demo.total = Demo.total + 1
>>> t1 = Demo()
>>> t1
<__main__.Demo object at 0x00000000034A0278>
>>> t2 = Demo()
>>> t3 = Demo()
>>> t4 = Demo()
Exception: 最多只能创建3个对象
>>> t4
NameError: name 't4' is not defined
```

6.2.3 成员方法、类方法、静态方法、抽象方法

- 方法一般指与特定实例绑定的函数，通过对象调用方法时，对象本身将被作为第一个参数自动传递过去，普通函数并不具备这个特点。

```
>>> class Demo:
    pass
>>> t = Demo()
>>> def test(self, v):
    self.value = v
>>> t.test = test                          #动态增加普通函数
>>> t.test
<function test at 0x00000000034B7EA0>
>>> t.test(t, 3)                           #需要为self传递参数
>>> print(t.value)
3
>>> import types
>>> t.test = types.MethodType(test, t)     #动态增加绑定的方法
>>> t.test
<bound method test of <__main__.Demo object at 0x000000000074F9E8>>
>>> t.test(5)                              #不需要位self传递参数
>>> print(t.value)
5
```

6.2.3 成员方法、类方法、静态方法、抽象方法

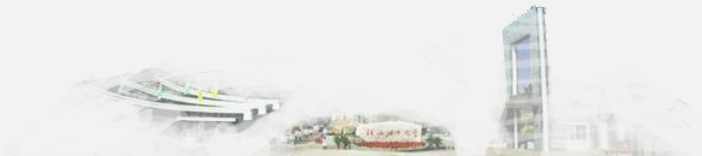
- Python类的成员方法大致可以分为公有方法、私有方法、静态方法、类方法和抽象方法这几种类型。
- 公有方法、私有方法和抽象方法一般是指属于对象的实例方法，私有方法的名字以两个开始，而抽象方法一般定义在抽象类中并且要求派生类必须重新实现。每个对象都有自己的公有方法和私有方法，在这两类方法中都可以访问属于类和对象的成员。公有方法通过对象名直接调用，私有方法不能通过对象名直接调用，只能在其他实例方法中通过前缀self进行调用或在外部通过特殊的形式来调用。

6.2.3 成员方法、类方法、静态方法、抽象方法

- 所有实例方法（包括公有方法、私有方法、抽象方法和某些特殊方法）都必须至少有一个名为self的参数，并且必须是方法的第一个形参（如果有多个形参的话），self参数代表当前对象。
- 在实例方法中访问实例成员时需要以self为前缀，但在外部通过对象名调用对象方法时并不需要传递这个参数。
- 如果在外部通过类名调用属于对象的公有方法，需要显式为该方法的self参数传递一个对象名，用来明确指定访问哪个对象的成员。

6.2.3 成员方法、类方法、静态方法、抽象方法

- 静态方法和类方法都可以通过类名和对象名调用，但不能直接访问属于对象的成员，只能访问属于类的成员。
- 静态方法和类方法不属于任何实例，不会绑定到任何实例，当然也不依赖于任何实例的状态，与实例方法相比能够减少很多开销。
- 类方法一般以cls作为类方法的第一个参数表示该类自身，在调用类方法时不需要为该参数传递值，静态方法则可以不接收任何参数。



6.2.3 成员方法、类方法、静态方法、抽象方法

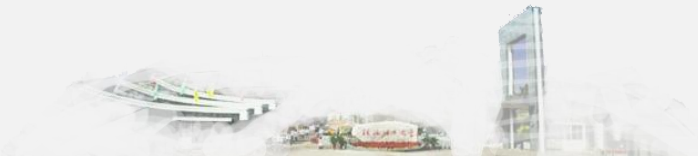
```
>>> class Root:
    __total = 0
    def __init__(self, v):      #构造方法
        self.__value = v
        Root.__total += 1

    def show(self):             #普通实例方法
        print('self.__value:', self.__value)
        print('Root.__total:', Root.__total)

    @classmethod                #修饰器，声明类方法
    def classShowTotal(cls):    #类方法
        print(cls.__total)

    @staticmethod               #修饰器，声明静态方法
    def staticShowTotal():      #静态方法
        print(Root.__total)
```

6.2.3 成员方法、类方法、静态方法、抽象方法

```
>>> r = Root(3)
>>> r.classShowTotal()                    #通过对象来调用类方法
1
>>> r.staticShowTotal()                   #通过对象来调用静态方法
1
>>> r.show()
self.__value: 3
Root.__total: 1
>>> rr = Root(5)
>>> Root.classShowTotal()                 #通过类名调用类方法
2
>>> Root.staticShowTotal()                #通过类名调用静态方法
2
```

6.2.3 成员方法、类方法、静态方法、抽象方法

```
>>> Root.show()      #试图通过类名直接调用实例方法，失败
TypeError: unbound method show() must be called with
Root instance as first argument (got nothing instead)
>>> Root.show(r)     #但是可以通过这种方法来调用方法并访问实例成员
self.__value: 3
Root.__total: 2
>>> Root.show(rr)    #通过类名调用实例方法时为self参数显式传递对象名
self.__value: 5
Root.__total: 2
```

6.2.3 成员方法、类方法、静态方法、抽象方法

- 抽象方法一般在抽象类中定义，并且要求在派生类中必须重新实现，否则不允许派生类创建实例。

```
import abc

class Foo(metaclass=abc.ABCMeta):  #抽象类
    def f1(self):                  #普通实例方法
        print(123)

    def f2(self):                  #普通实例方法
        print(456)

    @abc.abstractmethod            #抽象方法
    def f3(self):
        raise Exception('You musr reimplement this method.')

class Bar(Foo):
    def f3(self):                  #必须重新实现基类中的抽象方法
        print(33333)

b = Bar()
b.f3()
```

6.2.4 属性

- 只读属性

```
>>> class Test:
    def __init__(self, value):
        self.__value = value

    @property
    def value(self):                    #只读，无法修改和删除
        return self.__value
```

6.2.4 属性

```
>>> t = Test(3)
>>> t.value
3
>>> t.value = 5                          #只读属性不允许修改值
AttributeError: can't set attribute
>>> t.v=5                                #动态增加新成员
>>> t.v
5
>>> del t.v                              #动态删除成员
>>> del t.value                          #试图删除对象属性，失败
AttributeError: can't delete attribute
>>> t.value
3
```

6.2.4 属性

- 可读、可写属性

```
>>> class Test:
    def __init__(self, value):
        self.__value = value

    def __get(self):
        return self.__value

    def __set(self, v):
        self.__value = v
    value = property(__get, __set)

    def show(self):
        print(self.__value)
```

6.2.4 属性

```
>>> t = Test(3)
>>> t.value        #允许读取属性值
3
>>> t.value = 5    #允许修改属性值
>>> t.value
5
>>> t.show()       #属性对应的私有变量也得到了相应的修改
5
>>> del t.value    #试图删除属性，失败
AttributeError: can't delete attribute
```

6.2.4 属性

- 可读、可修改、可删除的属性。

```
>>> class Test:
    def __init__(self, value):
        self.__value = value

    def __get(self):
        return self.__value

    def __set(self, v):
        self.__value = v

    def __del(self):
        del self.__value

    value = property(__get, __set, __del)

    def show(self):
        print(self.__value)
```

6.2.4 属性

```
>>> t = Test(3)
>>> t.show()
3
>>> t.value
3
>>> t.value = 5
>>> t.show()
5
>>> t.value
5
```

6.2.4 属性

```
>>> del t.value                    #删除属性
>>> t.value                        #对应的私有数据成员已删除
AttributeError: 'Test' object has no attribute '_Test__value'
>>> t.show()
AttributeError: 'Test' object has no attribute '_Test__value'
>>> t.value = 1                    #为对象动态增加属性和对应的私有数据成员
>>> t.show()
1
>>> t.value
1
```

6.2.5 类与对象的动态性、混入机制

- 在Python中比较特殊的是，可以动态地为自定义类和对象增加或删除成员，这一点是和很多面向对象程序设计语言不同的，也是Python动态类型特点的一种重要体现。

6.2.5 类与对象的动态性、混入机制

```
class Car:
    price = 100000                  #定义类属性
    def __init__(self, c):
        self.color = c              #定义实例属性

car1 = Car("Red")                   #实例化对象
car2 = Car("Blue")
print(car1.color, Car.price)        #查看实例属性和类属性的值
Car.price = 110000                  #修改类属性
Car.name = 'QQ'                     #动态增加类属性
car1.color = "Yellow"               #修改实例属性
print(car2.color, Car.price, Car.name)
print(car1.color, Car.price, Car.name)
```

6.2.5 类与对象的动态性、混入机制

```
import types

def setSpeed(self, s):
    self.speed = s

car1.setSpeed = types.MethodType(setSpeed, car1) #动态增加成员方法
car1.setSpeed(50)                                 #调用成员方法
print(car1.speed)
```

6.2.5 类与对象的动态性、混入机制

- Python类型的动态性使得我们可以动态为自定义类及其对象增加新的属性和行为，俗称混入（mixin）机制，这在大型项目开发中会非常方便和实用。
- 例如系统中的所有用户分类非常复杂，不同用户组具有不同的行为和权限，并且可能会经常改变。这时候我们可以独立地定义一些行为，然后根据需要来为不同的用户设置相应的行为能力。

6.2.5 类与对象的动态性、混入机制

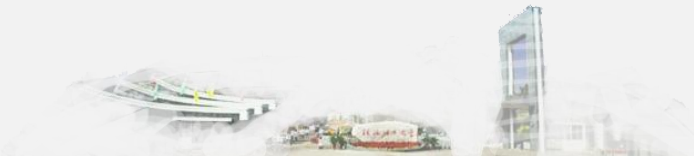
```
>>> import types
>>> class Person(object):
    def __init__(self, name):
        assert isinstance(name, str), 'name must be string'
        self.name = name

>>> def sing(self):
    print(self.name+' can sing.')

>>> def walk(self):
    print(self.name+' can walk.')

>>> def eat(self):
    print(self.name+' can eat.')
```

6.2.5 类与对象的动态性、混入机制

```
>>> zhang = Person('zhang')
>>> zhang.sing()                                    #用户不具有该行为
AttributeError: 'Person' object has no attribute 'sing'
>>> zhang.sing = types.MethodType(sing, zhang) #动态增加一个新行为
>>> zhang.sing()
zhang can sing.
>>> zhang.walk()
AttributeError: 'Person' object has no attribute 'walk'
>>> zhang.walk = types.MethodType(walk, zhang)
>>> zhang.walk()
zhang can walk.
>>> del zhang.walk                                  #删除用户行为
>>> zhang.walk()
AttributeError: 'Person' object has no attribute 'walk'
```

6.3 继承、多态

- 封装、继承、多态是面向对象编程的三大要素。



6.3.1 继承

- 继承是用来实现代码复用和设计复用的机制，是面向对象程序设计的重要特性之一。设计一个新类时，如果可以继承一个已有的设计良好的类然后进行二次开发，无疑会大幅度减少开发工作量。
- 在继承关系中，已有的、设计好的类称为父类或基类，新设计的类称为子类或派生类。派生类可以继承父类的公有成员，但是不能继承其私有成员。如果需要在派生类中调用基类的方法，可以使用内置函数super()或者通过“基类名.方法名()”的方式来实现这一目的。
- Python支持多继承，如果父类中有相同的方法名，而在子类中使用时没有指定父类名，则Python解释器将从左向右按顺序进行搜索。

6.3.1 继承

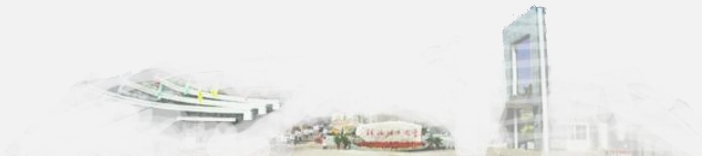
- 示例6-1：在派生类中调用基类方法。

code\AccessMembersOfBaseclass.py

6.3.2 多态

- 所谓多态（polymorphism），是指基类的同一个方法在不同派生类对象中具有不同的表现和行为。派生类继承了基类行为和属性之后，还会增加某些特定的行为和属性，同时还可能会对继承来的某些行为进行一定的改变，这都是多态的表现形式。

- Python大多数运算符可以作用于多种不同类型的操作数，并且对于不同类型的操作数往往有不同的表现，这本身就是多态，是通过特殊方法与运算符重载实现的。



6.3.2 多态

```
>>> class Animal(object):          #定义基类
    def show(self):
        print('I am an animal.')
>>> class Cat(Animal):             #派生类，覆盖了基类的show()方法
    def show(self):
        print('I am a cat.')
>>> class Dog(Animal):             #派生类
    def show(self):
        print('I am a dog.')
>>> class Tiger(Animal):           #派生类
    def show(self):
        print('I am a tiger.')
>>> class Test(Animal):            #派生类，没有覆盖基类的show()方法
    pass
```

6.3.2 多态

```
>>> x = [item() for item in (Animal, Cat, Dog, Tiger, Test)]
>>> for item in x:                    #遍历基类和派生类对象并调用show()方法
    item.show()
```

```
I am an animal.
I am a cat.
I am a dog.
I am a tiger.
I am an animal.
```

6.4 特殊方法与运算符重载

- Python类有大量的特殊方法，其中比较常见的是构造函数和析构函数，除此之外，Python还支持大量的特殊方法，运算符重载就是通过重写特殊方法实现的。

✓Python中类的构造函数是__init__()，一般用来为数据成员设置初值或进行其他必要的初始化工作，在创建对象时被自动调用和执行。如果用户没有设计构造函数，Python将提供一个默认的构造函数用来进行必要的初始化工作。

✓Python中类的析构函数是__del__()，一般用来释放对象占用的资源，在Python删除对象和收回对象空间时被自动调用和执行。如果用户没有编写析构函数，Python将提供一个默认的析构函数进行必要的清理工作。

6.4 特殊方法与运算符重载

方法	功能说明
__new__()	类的静态方法，用于确定是否要创建对象
__init__()	构造方法，创建对象时自动调用
__del__()	析构方法，释放对象时自动调用
__add__()	+
__sub__()	-
__mul__()	*
__truediv__()	/
__floordiv__()	//
__mod__()	%
__pow__()	**
__eq__()、__ne__()、 __lt__()、__le__()、 __gt__()、__ge__()	==、!=、 <、<=、 >、>=
__lshift__()、__rshift__()	<<、>>
__and__()、__or__()、 __invert__()、__xor__()	&、|、 ~、^

6.4 特殊方法与运算符重载

方法	功能说明
__iadd__()、__isub__()	+=、-=，很多其他运算符也有与之对应的复合赋值运算符
__pos__()	一元运算符+，正号
__neg__()	一元运算符-，负号
__contains__ ()	与成员测试运算符in对应
__radd__()、__rsub__	反射加法、反射减法，一般与普通加法和减法具有相同的功能，但操作数的位置或顺序相反，很多其他运算符也有与之对应的反射运算符
__abs__()	与内置函数abs()对应
__bool__()	与内置函数bool()对应，要求该方法必须返回True或False
__bytes__()	与内置函数bytes()对应
__complex__()	与内置函数complex()对应，要求该方法必须返回复数
__dir__()	与内置函数dir()对应
__divmod__()	与内置函数divmod()对应
__float__()	与内置函数float()对应，要求该该方法必须返回实数
__hash__()	与内置函数hash()对应
__int__()	与内置函数int()对应，要求该方法必须返回整数

6.4 特殊方法与运算符重载

方法	功能说明
__len__()	与内置函数len()对应
__next__()	与内置函数next()对应
__reduce__()	提供对reduce()函数的支持
__reversed__()	与内置函数reversed()对应
__round__()	对内置函数round()对应
__str__()	与内置函数str()对应，要求该方法必须返回str类型的数据
__repr__()	打印、转换，要求该方法必须返回str类型的数据
__getitem__()	按照索引获取值
__setitem__()	按照索引赋值
__delattr__()	删除对象的指定属性
__getattr__()	获取对象指定属性的值，对应成员访问运算符“.”

6.4 特殊方法与运算符重载

方法	功能说明
__getattribute__()	获取对象指定属性的值，如果同时定义了该方法与__getattr__()，那么__getattr__()将不会被调用，除非在__getattribute__()中显式调用__getattr__()或者抛出AttributeError异常
__setattr__()	设置对象指定属性的值
__base__	该类的基类
__class__	返回对象所属的类
__dict__	对象所包含的属性与值的字典
__subclasses__()	返回该类的所有子类
__call__()	包含该特殊方法的类的实例可以像函数一样调用
__get__()	定义了这三个特殊方法中任何一个的类称作描述符（descriptor），描述符对象一般作为其他类的属性来使用，这三个方法分别在获取属性、修改属性值或删除属性时被调用
__set__()	
__delete__()	

6.5 精彩案例赏析

- 6.5.1 自定义队列
- 6.5.2 自定义栈

6.5.1 自定义队列

- **例6-2** 自定义队列结构，实现入队、出队操作。

code\myQueue.py

6.5.1 自定义栈

- **例6-3** 自定义栈，实现基本的入栈、出栈操作。

code\stackDfg.py