



Python程序设计

—从基础到开发

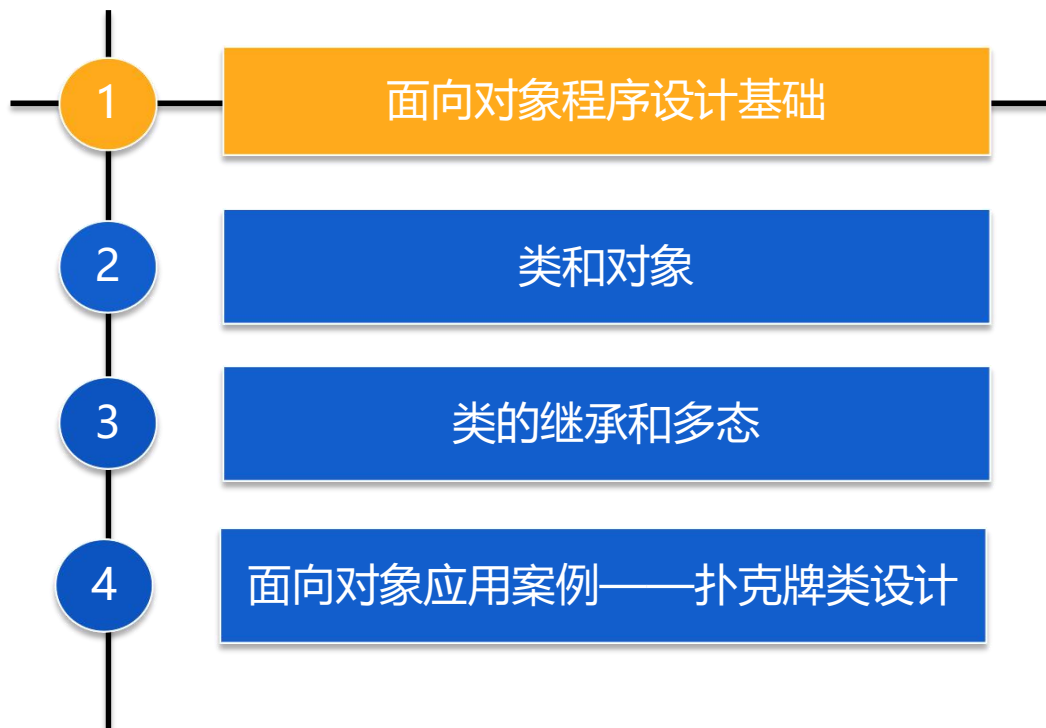
第6章 面向对象程序设计

面向对象思想

- 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, 并且使得代码具有更好的可读性和可扩展性。
- 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, 即**对象**。
- 对于相同类型的对象进行分类、抽象后得出共同的特征而形成了**类**。
- 面向对象程序设计的关键就是如何合理地定义和组织这些类以及类之间的关系。

目录

第6章 面向对象程序设计



6.1 面向对象程序设计基础

对象的概念

- 对象 —— 程序代码的整个结构的基础和组成元素。
 - 它采用数据抽象和信息隐藏技术，将对象及对象的操作抽象成一种新的数据类型——类，并且考虑不同对象之间的联系和对象类的重用性。
- 简而言之：
 - 对象就是现实世界中的一个实体，而类就是对象的抽象和概括。
- 现实生活中的每一个相对独立的事物都可以看做一个对象，例如：一个人，一辆车，一台电脑等。

6.1 面向对象程序设计基础

对象的概念

- 每个对象都具有描述其特征的**属性**及附属于它的**行为**。例如，
 - 一辆车有颜色、车轮数、座椅数等**属性**，也有启动、行驶、停止等**行为**。
 - 一个人是有姓名、性别、年龄身高、体重等特征描述，也有走路、说话学习、开车等行为。
- 面向对象程序设计的基本思路：
 - 当人们生产一台车的时候，并不是先要生产车架、再生产座椅、再生产车门、车轮、大灯等，即不是顺序执行的（面向过程的）。
 - 而是分别生产车架、座椅、车轮等，最后把它们组装起来。这些部件通过事先设计好的接口连接，以便协调的工作。

6.1 面向对象程序设计基础

类的概念

- 每个对象都有一个类型——类，它是创建对象实例的模板，是对对象的抽象和概括，它包含对所创建对象的属性描述和行为特征的定义。
- 例如，我们在马路上看到的汽车都是一个一个的汽车对象，它们通通归属于一个汽车类，那么车身颜色就是该类的属性，开动是它的方法，该保养了或者该报废了就是它的事件。

6.1 面向对象程序设计基础

面向对象程序设计的基本特征

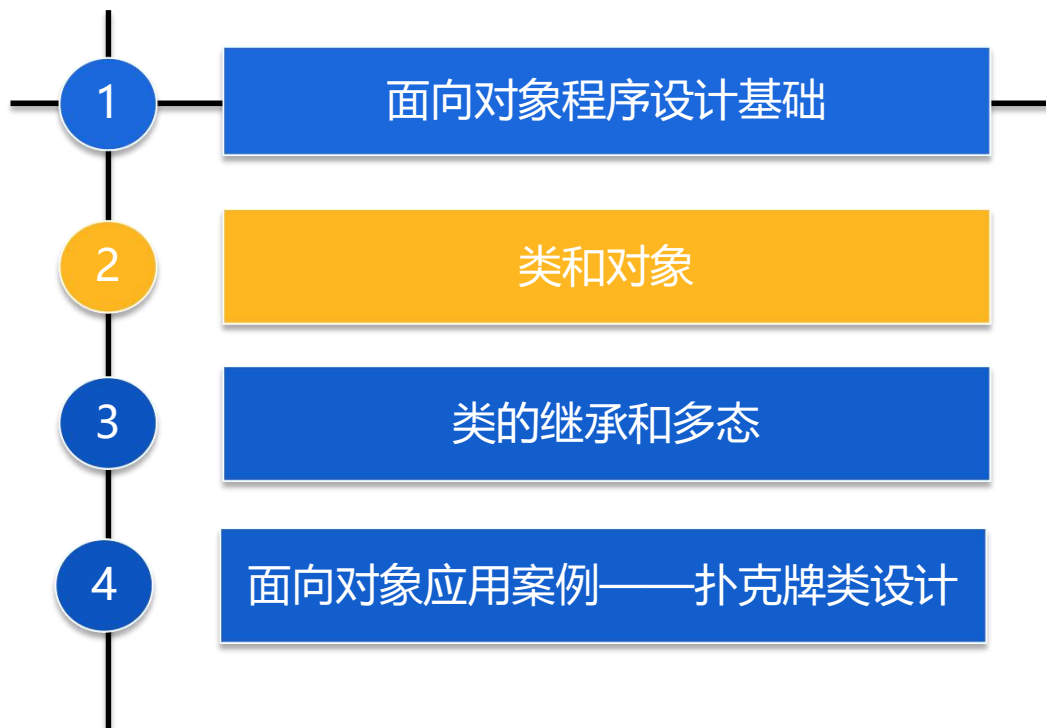
- 面向对象程序设计是一种计算机编程架构，它具有以下3个基本特性。
 - (1) 封装性 (Encapsulation)
 - (2) 继承性 (Inheritance)
 - (3) 多态性 (Polymorphism)
- Python完全采用了面向对象程序设计的思想，是真正面向对象的高级动态编程语言，完全支持面向对象的基本功能，如封装、继承、多态以及对基类方法的覆盖或重写。

6.1 面向对象程序设计基础

- 但与其他面向对象程序设计语言不同的是，Python中对象的概念很广泛，**Python中的一切内容都可以称为对象，而不一定必须是某个类的实例。**
 - ❑ 例如，字符串、列表、字典、元组等内置数据类型都具有和类完全相似的语法和用法。
- 创建类时：
 - ❑ 用变量形式表示的对象属性称为**数据成员或成员属性**；
 - ❑ 用函数形式表示的对象行为称为**成员函数或成员方法**；
 - ❑ 成员属性和成员方法统称为类的成员。

目录

第6章 面向对象程序设计



6.2 类和对象

6.2.1 类和对象的定义

- Python使用**class**关键字来**定义类**。
- 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尤其重要。例如：

class 类名:

属性 (成员变量)

属性

.....

成员函数 (成员方法)

class Car:

price = 100000

def infor(self):

print(" This is a car ")

6.2.1 类和对象的定义

普通函数与成员函数

- Python中，函数和成员函数（成员方法）是有区别的。
- 成员函数一般指与特定实例绑定的函数，通过对象调用。对象本身将被作为第一个参数传递过去。普通函数没有这个特点。
- 成员函数中必须有一个参数`self`，并且必须位于参数列表的第一个。`self`代表类实例（对象）本身。
- 在类的成员函数中若要访问**实例属性**，需要**以 `self` 为前缀**。

```
class Car:  
    price = 100000  
    def infor(self):  
        print(" This is a car ")
```

6.2.1 类和对象的定义

对象定义

- 定义了类之后，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，例如下面的代码：

```
class Car:  
    price = 100000  
    def infor(self):  
        print(" This is a car ")
```

```
>>> car = Car()  
>>> car.infor()  
This is a car
```

- 在Python中，可以使用内置方法 `isinstance()` 来测试一个对象是否为某个类的实例，下面的代码演示了 `isinstance()` 的用法。

```
>>> isinstance(car, Car)  
True  
>>> isinstance(car, str)  
False
```

6.2.2 构造函数 `__init__`

构造函数完成对象变量初始化工作

- 类可以定义一个特殊的方法 `__init__()`，叫**构造函数**。当类实例化（创建对象）时，就会自动为新生的对象调用该构造函数。
 - 构造函数一般用于完成对象数据成员设置初值或进行必要的初始化工作。
 - 如果没有给类定义构造函数，Python会提供一个默认的构造函数。
- 【例】定义一个复数类Complex，构造函数完成对象变量初始化工作。

```
class Complex:           #类定义
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

x = Complex(3.0,-4.5)    #类实例化，创建对象
print(x.r, x.i)
```

6.2.3 析构函数__del__

析构函数用来释放对象占用的资源

- 对象的类中定义的一个特殊函数`__del__()`，叫**析构函数**。当对象被删除之后，会自动执行。
 - 析构函数会收回对象空间，也就是释放对象占用的资源。
 - 如果用户没有设计析构函数，Python将提供一个默认的析构函数来进行必要的清理工作。

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
    def __del__(self):
        print("Complex不存在了")
```

```
x = Complex(3.0,-4.5)
print(x.r, x.i)
print(x)
del x    #删除x对象变量
```

输出结果：

```
3.0 -4.5
<__main__.Complex object at
0x0000019905429550>
Complex不存在了
```

6.2.4 实例属性和类属性

➤ 属性（成员变量）有两种：

- ❑ 实例属性：属于实例(对象)，只能通过对象名访问。可以在构造函数中定义，定义时以 `self` 作为前缀；也可以在创建对象后，增加实例属性。
- ❑ 类属性：属于类，可通过类名访问，也可以通过对象名访问，为类的所有实例共享。

➤ 例

```
class Person:
    num = 1                                #类属性
    def __init__(self, str, n):            #构造函数
        self.name = str                   #实例属性
        self.age = n
    def SayHello(self):                     #成员函数
        print("Hello!")
    def PrintName(self):                    #成员函数
        print("姓名: ", self.name, "年龄: ", self.age) #实例属性访问
    def PrintNum(self):                     #成员函数
        print(Person.num)                  #类属性访问
```

6.2.4 实例属性和类属性

```
class Person:
    num = 1                                #类属性
    def __init__(self, str, n):            #构造函数
        self.name = str                   #实例属性
        self.age = n
    def PrintName(self):                   #成员函数
        print("姓名: ", self.name, "年龄: ", self.age)
    def PrintNum(self):                   #成员函数
        print(Person.num)                #由于是类属性，所以不写self.num

#主程序*****
P1= Person("张三", 42)                   #实例属性赋值
P2= Person("李四", 36)
P1.PrintName()                          #输出: 姓名: 张三 年龄: 42
P2.PrintName()                          #输出: 姓名: 李四 年龄: 36
Person.num = 2                          #类属性赋值
P1.PrintNum()                           #输出: 2
P2.PrintNum()                           #输出: 2
```


6.2.5 动态增加成员

- 在Python中比较特殊的是，可以动态地为类和对象增加成员（属性和方法）
- 例：动态增加**实例属性**

```
class Student(object):  
    name = 'Student'
```

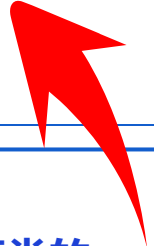
```
s = Student()  
print(s.name)
```

```
# 创建实例 s  
# 打印name属性，因为实例并没有name属性，  
# 所以会继续查找class的name属性  
print(Student.name)  
# 打印类的name属性
```

输出结果：
Student
Student



输出结果：
xiaoming
Student



```
s.name = 'xiaoming'      # 给实例增加name属性  
print(s.name)           # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性  
print(Student.name)     # 但是类属性并未消失，用Student.name仍然可以访问
```

6.2.5 动态增加成员

- 在Python中比较特殊的是，可以动态地为类和对象增加成员（属性和方法）
- 例：动态增加**实例属性**

```
class Student(object):
```

```
    name = 'Student'
```

```
s = Student()
```

创建实例 s

```
s.name = 'xiaoming'
```

给实例增加name属性

```
print(s.name) # 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性
```

```
print(Student.name) # 但是类属性并未消失，用Student.name仍然可以访问
```

输出结果：

xiaoming

Student

输出结果：

Student

Student

```
del s.name # 如果删除实例的name属性
```

```
print(s.name) # 再次调用s.name，由于实例的name属性没有找到，类的name属性就显示出来了
```

```
print(Student.name)
```

6.2.5 动态增加成员

- 在Python中比较特殊的是，可以动态地为类和对象增加成员（属性和方法）
- 例：动态增加**实例属性**

```
class Student(object):  
    name = 'Student'
```

```
s = Student()
```

创建实例 s

```
s.score = 90
```

给实例增加score属性

```
print(s.name)
```

```
print(s.score)
```

```
print(Student.score)
```

#类没有增加score属性，所以报错

输出结果：

Student

90

AttributeError: type object 'Student' has no attribute 'score'

6.2.5 动态增加成员

- 在Python中比较特殊的是，可以动态地为类和对象增加成员（属性和方法）
- 例：动态增加**类属性**

```
class Student(object):  
    name = 'Student'
```

```
s1 = Student()
```

```
# 创建实例 s1
```

```
s2 = Student()
```

```
# 创建实例 s2
```

```
Student.score = 90
```

```
# 增加类属性score
```

```
print(s1.score)
```

```
print(s2.score)
```

```
s1.score = 80
```

```
# 增加实例属性score
```

```
print(s1.score)
```

```
print(s2.score)
```

输出结果：

90

90

80

90

6.2.5 动态增加成员

➤ 动态增加实例方法

```
import types
def set_age(self, age):    #定义一个普通函数
    self.age = age

class Student:
    pass
s1 = Student()
s1.set_age = types.MethodType(set_age, s1)
s1.set_age(23)
print(s1.age)    #输出23
s2=Student()
s2.set_age(24)    #AttributeError: no attribute 'set_age'
```

➤ 只是将set_age函数绑定到了s1对象上, 而没有绑定到Student类上

要绑定的对象

要绑定的方法

6.2.5 动态增加成员

➤ 动态增加类方法

```
import types
def set_age(self, age):    #定义一个普通函数
    self.age = age

class Student:
    pass
```

➤ 将set_age函数绑定到了Student类上

```
Student.set_age=types.MethodType(set_age, Student)
s1=Student()
s2=Student()
s1.set_age(12)
s2.set_age(13)
print(s1.age)    #输出13
print(s2.age)    #输出13
```

要绑定的类

要绑定的方法

6.2.6 私有成员与公有成员

Python并没有对私有成员提供严格的访问保护机制

- 在定义类的属性时，如果属性名以两个下划线 “__” 开头则表示是私有属性，否则是公有属性。
- 私有属性在类的外部不能直接访问
 - ❑ 需通过调用对象的公有成员方法来访问
 - ❑ 或者通过Python支持的特殊方式来访问
 - Python提供了访问私有属性的特殊方式，可用于程序的测试和调试，对于成员方法也有同样性质。方式如下：**对象名._类名+私有成员**
 - 例如：**访问Car类的私有成员__weight**

```
>>> car1._Car__weight
```

6.2.6 私有成员与公有成员

示例

```
class Car:
    price = 100000          #定义类属性
    def __init__(self, c, w):
        self.color = c      #定义公有属性color
        self.__weight = w   #定义私有属性__weight

#主程序
car1 = Car("Red", 10.5)
car2 = Car("Blue", 11.8)
print(car1.color)          #公有属性color可以直接访问
print(car1._Car__weight)
print(car1.__weight)       # AttributeError
```


6.2.6 私有成员与公有成员

私有成员

- 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。用下划线作为变量名和方法名前缀和后缀来表示类的特殊成员：
 - xxx 这样的对象叫做保护成员，不能用 'from module import *' 导入，只有类和子类内部成员方法(函数)能访问这些成员；
 - xxx 系统定义的特殊成员；
 - xxx 类中的私有成员，只有类自己内部成员方法(函数)能访问，子类内部成员方法也不能访问到这个私有成员，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。
- Python 中不存在严格意义上的私有成员。

6.2.7 方法

在类中定义的方法可以粗略分为3大类：

- 公有方法(类方法)、私有方法、静态方法(普通方法)
- 公有方法、私有方法都**属于对象**，（私有方法的名字以两个下划线 “__” 开始），每个对象都有自己的公有方法和私有方法，在这两类方法中可以访问属于类和对象的成员；
- 公有方法可以通过对象名直接调用，或者通过类名调用（要传递self参数，就是对象名）
- 私有方法不能通过对象名直接调用，只能在属于对象的方法中通过“self”调用或在外部通过Python支持的特殊方式来调用。
- 静态方法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。

6.2.7 方法

```
class Fruit:
    price=0
    def __init__(self):
        self.__color = 'Red'
        self.__city = 'Kunming'
    def __outputAttr(self):
        print(self.__color, self.__city)
    def output(self):
        self.__outputAttr()
    @staticmethod
    def getPrice():
        return Fruit.price
    @staticmethod
    def setPrice(p):
        Fruit.price = p

#主程序
apple = Fruit()
apple.output()
print(Fruit.getPrice())
apple.setPrice(9)
print(Fruit.getPrice())
```

#定义和设置私有属性color, city

#定义私有方法outputAttr
#访问私有属性color, city

#定义公有方法（类方法）output
#调用私有方法outputColor

#定义静态方法getPrice

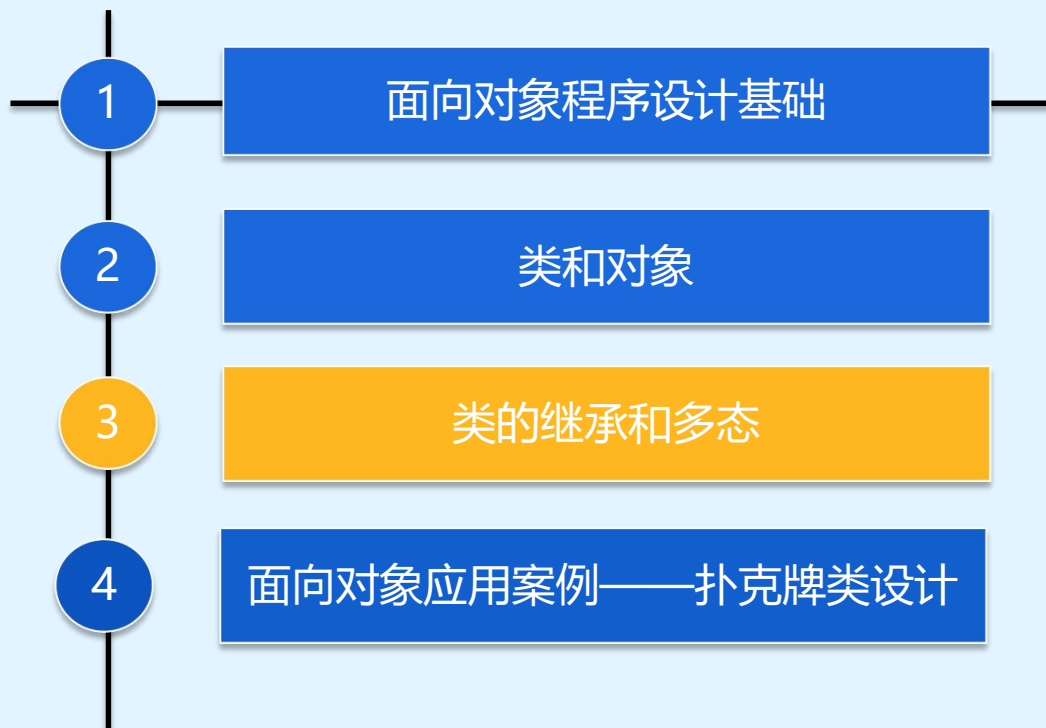
#定义静态方法setPrice

#创建对象
#通过对象调用公有方法，不用传参(对象apple传进去了)
#通过类名调用静态方法
#通过对象名调用静态方法

输出结果：
Red Kunming
0
9

目录

第6章 面向对象程序设计



6.3 类的继承和多态

6.3.1 类的继承

➤ 继承是为了代码复用和设计复用，是面向对象程序设计的重要特征之一。

➤ 类继承语法：

```
class 派生类名 (基类名) : #基类名写在括号里  
    派生类成员
```

➤ 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，不能继承其私有成员

6.3 类的继承和多态

6.3.1 类的继承

➤ 在Python中继承的一些特点：

- ① 基类的构造函数不会被自动调用，需要在其派生类的构造中亲自专门调用
- ② 如果需要在派生类中调用基类的方法时，通过“基类名.方法名()”来实现，还需要带上self参数变量。
- ③ Python总是首先查找对应类型的方法，如果它不能在派生类(本类)中找到对应的方法，才开始到基类中逐个查找。

6.3.1 类的继承

```
class Parent:      #定义父类
    parentAttr = 100
    def __init__(self):
        print ("父类构造函数")
    def parentMethod(self):
        print("父类方法")
    def setAttr(self, attr):
        Parent.parentAttr = attr
    def getAttr(self):
        print( "父类属性 :", Parent.parentAttr)

class Child(Parent):      #定义子类
    def __init__(self):
        print( "子类构造函数")
    def childMethod(self):
        print("子类方法 child method")
```

#主程序

```
c = Child()      #实例化子类
c.childMethod()  #调用子类方法
c.parentMethod() #调用父类方法
c.setAttr(200)   #调用父类方法
c.getAttr()      #调用父类方法
```

输出结果:

```
子类构造函数
子类方法 child method
父类方法
父类属性 : 200
```

6.3.1 类的继承

示例：设计Person类，并根据Person派生Student类，分别创建Person类和Student类的对象

#定义基类： Person类

import types

class Person(object): #基类必须继承于object， 否则在派生类中将无法使用super()函数

def __init__(self, name = '', age = 20, sex = 'man'):

self.setName(name)

self.setAge(age)

self.setSex(sex)

def setName(self, name):

if type(name) != str: #内置函数type()返回被测对象的数据类型

print ('姓名必须是字符串.')

return

self.__name = name

6.3.1 类的继承

示例-续1

```
#定义基类: Person类
#####接上页#####
def setAge(self, age):
    if type(age) != int:
        print ('年龄必须是整型.')
    return
    self.__age = age

def setSex(self, sex):
    if sex != '男' and sex != '女':
        print ('性别输入错误')
    return
    self.__sex = sex

def show(self):
    print ('姓名: ', self.__name, '年龄: ', self.__age, '性别: ', self.__sex)
```

6.3.1 类的继承

示例-续2

#定义子类 (Student类) , 其中增加一个入学年份私有属性 (数据成员) 。

class Student (Person):

def __init__(self, name='', age = 20, sex = 'man', schoolyear = 2016):

#调用基类构造方法初始化基类的私有数据成员

super(Student, self).__init__(name, age, sex)

#Person.__init__(self, name, age, sex) #也可以这样初始化基类私有数据成员

self.setSchoolyear(schoolyear) #初始化派生类的数据成员

def setSchoolyear(self, schoolyear):

self.__schoolyear = schoolyear

def show(self):

Person.show(self)

#调用基类show()方法, 基类名.方法名()

#super(Student, self).show()

#也可以这样调用基类show()方法

print ('入学年份: ', self.__schoolyear)

6.3.1 类的继承

示例-续3

#主程序

```
if __name__ == '__main__':  
    zhangsan = Person('张三', 19, '男')  
    zhangsan.show()  
    lisi = Student('李四', 18, '男', 2015)  
    lisi.show()  
    lisi.setAge(20)           #调用继承的方法修改年龄  
    lisi.show()
```

输出结果：

姓名： 张三 年龄： 19 性别： 男
姓名： 李四 年龄： 18 性别： 男
入学年份： 2015
姓名： 李四 年龄： **20** 性别： 男
入学年份： 2015

6.3.2 类的多继承

- Python的类可以继承多个基类。继承的基类列表跟在类名之后。
- 类的多继承语法：

class SubClassName (ParentClass1[, ParentClass2, ...]):

派生类成员（包括属性和方法）

```
class A:           # 定义类 A  
  
    ....  
class B:         # 定义类 B  
  
    ....  
class C(A, B):   # 派生类C 继承类 A 和 B  
  
    ....
```

6.3.3 方法重写

- 重写必须出现在继承中。它是指当派生类继承了基类的方法之后，如果基类方法的功能不能满足需求，需要对基类中的某些方法进行修改，可以在派生类重写基类的方法，这就是重写。

```
class Animal:                # 定义父类
    def run(self):
        print ('动物在跑')
class Cat (Animal):          # 定义子类
    def run (self):
        print ('猫在跑.....')
class Dog (Animal):          # 定义子类
    def run (self):
        print ('狗在跑.....')

c = Dog()                    # 子类实例
c.run ()                     # 子类调用重写方法
```

6.3.4 多态

术语多态来自希腊语，意思是“有多种形式”

- 要理解什么是多态，要对数据类型进一步理解。
- 在Python中，当定义一个class时，实际上就定义了一种数据类型。它和Python自带的数据类型（如string，list等），没什么区别。

```
a = list()
b = Animal()
c = Dog()
```

```
>>> isinstance(a, list)
True
>>> isinstance(b, Animal)
True
>>> isinstance(c, Dog)
True
>>> isinstance(c, Animal)
True
```

- a,b,c 确实对应这三种类型
- 同时，c也属于Animal
- 所以，继承关系中，如果一个

实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。

6.3.4 多态

要理解多态的好处，请看实例：

```
def run_twice(animal):  
    animal.run()  
    animal.run()
```

```
>>> run_twice(Animal())
```

输出结果：
动物在跑
动物在跑

```
>>> run_twice(Dog())
```

输出结果：
狗在跑.....
狗在跑.....

```
>>> run_twice(Cat())
```

输出结果：
猫在跑.....
猫在跑.....

```
Class Tortoise(Animal):  
    def run(self)  
        print ('乌龟跑得很慢.....')
```

```
>>> run_twice(Tortoise())
```

输出结果：
乌龟跑得很慢.....
乌龟跑得很慢.....

6.3.4 多态

- 多态的好处就是，当我们需要传入Dog、Cat、Tortoise.....时，我们只需要接收Animal类型就可以了，因为Dog、Cat、Tortoise.....都是Animal类型，然后，按照Animal类型进行操作即可。
- 由于Animal类型有run()方法，因此，传入的任意类型，只要是Animal类或者子类，就会自动调用实际类型的run()方法，这就是多态的意思。

6.3.5 运算符重载

- 在Python中可以通过运算符重载来实现对象之间的运算。Python把运算符与类的方法关联起来，每个运算符对应一个函数，因此重载运算符就是实现函数。常用的运算符与函数方法的对应关系如表所示。

函数方法	重载的运算符	说明	调用举例
<u>__add__</u>	+	加法	Z=X+Y, X+=Y
<u>__sub__</u>	-	减法	Z=X-Y, X-=Y
<u>__mul__</u>	*	乘法	Z=X*Y, X*=Y
<u>__div__</u>	/	除法	Z=X/Y, X/=Y
<u>__lt__</u>	<	小于	X < Y
<u>__eq__</u>	=	等于	X = Y
<u>__len__</u>	len	对象长度	len(X)
<u>__str__</u>	print	输出对象时调用	print(X), str(X)
<u>__or__</u>		或运算	X Y, X =Y

6.3.5 运算符重载

实例

```
class Vector:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __str__(self):          #返回一个对象的描述信息
#相当于, 重写print()方法, 打印Vector对象实例信息
        return 'Vector (%d, %d)' % (self.a, self.b)
    def __add__(self, other):   #重载加法+运算符
        return Vector(self.a + other.a, self.b + other.b)
    def __sub__(self, other):   #重载减法-运算符
        return Vector(self.a - other.a, self.b - other.b)

#主程序
v1 = Vector(2,10)
v2 = Vector(5,-2)
print (v1 + v2)
```

输出结果:
Vector (7,8)

6.3.5 运算符重载

- 在python中方法名如果是__xxxx__()的，那么就有特殊的功能，因此叫做“魔法”方法
- 当使用print输出对象的时候，只要自己定义了__str__(self)方法，那么就会打印从在这个方法中return的数据
- __str__方法需要返回一个字符串，当做这个对象的描写

目录

第6章 面向对象程序设计



面向对象应用——扑克牌类设计

【案例】采用扑克牌类设计扑克牌发牌程序

- 4名牌手打牌，电脑随机将52张牌（不含大小鬼）发给4名牌手，在屏幕上显示每位牌手的牌。程序的运行效果如图所示：

This is a module with classes for playing card.

牌手1: ♠ 6 ♠ 9 ♠ A ♣ 10 ♣ 4 ♣ A ♥ 2 ♥ 3 ♥ 7 ♦ 2 ♦ 5 ♦ K ♦ Q

牌手2: ♠ 2 ♠ J ♠ K ♠ Q ♣ 5 ♣ 6 ♣ 7 ♣ 8 ♣ J ♣ K ♥ 4 ♥ 5 ♥ A

牌手3: ♠ 4 ♠ 7 ♣ 9 ♣ Q ♥ 9 ♥ J ♥ K ♥ Q ♦ 3 ♦ 8 ♦ 9 ♦ A ♦ J

牌手4: ♠ 10 ♠ 3 ♠ 5 ♠ 8 ♣ 2 ♣ 3 ♥ 10 ♥ 6 ♥ 8 ♦ 10 ♦ 4 ♦ 6 ♦ 7

面向对象应用——扑克牌类设计

关键技术——random模块

- random模块是内建 (built-in) 函数，作用是产生随机数，发牌游戏案例中使用这些函数实现洗牌等功能。常用的随机函数方法如下：
- random.random(): 在 $[0,1.0)$ 范围内随机生成一个小数；
- random.uniform(a,b): 在 $[a,b]$ 或 $[b,a]$ 范围内随机生成一个小数；
- random.randint(a,b): 在 $[a,b]$ 范围内随机生成一个整数；
- random.randrange([start,]stop[,step]): 在 $[start,stop)$ 范围内，按指定基数递增的集合中产生随机数；
- random.choice(seq): 从序列中获取一个随机元素；
- random.shuffle(x[,random]): 将列表中的元素打乱；
- random.sample(sequence,k): 从指定序列中随机获取指定个数的元素。

面向对象应用——扑克牌类设计

关键技术——random模块

```
>>> import random
>>> random.choice('abcdefg&#x21%^*f')
'd'
>>> random.sample('abcdefghij',3)
['i', 'g', 'c']
>>> " ".join(random.sample(['a','b','c','d','e','f','g','h','i','j'],3)).replace(" ", "")
'haf'
>>> random.randrange(0,101,2)
54
>>> random.uniform(1,100)
76.78421995428887
```

面向对象应用——扑克牌类设计

程序设计思路：设计出3个类：Card类、Hand类、Poke类

1. Card类

➤ Card类代表一张牌，其中FaceNum字段指的是牌面数字1~13，Suit字段指的是花色，值"♣"为梅花，"♦"为方钻，"♥"为红心，"♠"为黑桃。

❶ Card构造函数根据参数初始化封装的成员变量，实现牌面大小和花色的初始化，以及是否显示牌面，默认True为显示牌正面。

❷ _str_()方法用来输出牌面大小和花色。

❸ pic_order()方法获取牌的顺序号，牌面按梅花1--13，方块14--26，红桃27--39，黑桃0--52顺序编号(未洗牌之前)。这个方法为图形化显示牌面预留的方法。

❹ flip()是翻牌方法，改变牌面是否显示的属性值。

面向对象应用——扑克牌类设计

1. Card类

Basic classes for a game with playing cards

class Card():

""" A playing card. """

RANKS = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K'] #牌面数字

SUITS = ['♣', '♦', '♥', '♠'] #♣ 为梅花, ♦ 为方钻, ♥ 为红心, ♠ 为黑桃

def __init__(self, rank, suit, face_up = True):

self.rank = rank #指的是牌面数字 1--13

self.suit = suit #suit指的是花色

self.is_face_up = face_up #是否显示牌正面, T为正面, F为牌背面

def __str__(self): #输出牌面大小和花色

if self.is_face_up:

rep = self.suit + self.rank #+" "+ str(self.pic_order())

else:

rep = "XX"

return rep

面向对象应用——扑克牌类设计

1. Card类

```
def flip(self):                #翻牌方法
    self.is_face_up = not self.is_face_up

def pic_order(self):           #返回牌的顺序号1-52
    if self.rank=="A":         FaceNum=1
    elif self.rank=="J":       FaceNum=11
    elif self.rank=="Q":       FaceNum=12
    elif self.rank=="K":       FaceNum=13
    else:
        FaceNum=int(self.rank)

    if self.suit=="梅":         Suit=1
    elif self.suit=="方":       Suit=2
    elif self.suit=="红":       Suit=3
    else:
        Suit=4
    return (Suit - 1) * 13 + FaceNum
```

面向对象应用——扑克牌类设计

程序设计思路：设计出3个类：Card类、Hand类、Poke类

2. Hand类

- Hand类代表一手牌（一个玩家手里拿的牌），其中列表变量cards，用来存储牌手手里的牌。可以增加牌、清空手里的牌、把一张牌给别的牌手。

```
class Hand():  
    """ A hand of playing cards. """  
    def __init__(self):  
        self.cards = []  
  
    def __str__(self):          #重写print()方法  
        if self.cards:  
            rep = ""  
            for card in self.cards:  
                rep += str(card) + "\t"  
        else:  
            rep = "无牌"  
        return rep
```

面向对象应用——扑克牌类设计

2. Hand 类

```
def clear(self):  
    self.cards = []  
  
def add(self, card):  
    self.cards.append(card)  
  
def give(self, card, other_hand):  
    self.cards.remove(card)  
    other_hand.add(card)
```

面向对象应用——扑克牌类设计

程序设计思路：设计出3个类：Card类、Hand类、Poke类

3. Poke类

- Poke类代表一副牌，可以把一副牌看成是有52张牌的牌手，所以继承Hand类。但是由于Poke类中的cards列表变量要存储52张牌，还要发牌、洗牌操作，所以增加如下方法：
 - ❶ populate(self)方法，生成存储了52张牌的一手牌，当然这些牌是按照一定顺序（未洗牌之前）存储在cards列表中。
 - ❷ shuffle()方法，用来洗牌。使用random.shuffle()打乱牌的顺序即可。
 - ❸ deal(self, hands, per_hand=13)方法，完成发牌动作，发给4个玩家，每人默认13张牌。当然可以用per_hand参数设定每人发牌量。

面向对象应用——扑克牌类设计

3. Poke类

```
class Poke(Hand):          #继承Hand类
    """ A deck of playing cards. """
    def populate(self):      #生成一副牌
        for suit in Card.SUITS:
            for rank in Card.RANKS:
                self.add(Card(rank, suit))    #Card(rank, suit)生成一张牌

    def shuffle(self):       #洗牌
        import random
        random.shuffle(self.cards)          #打乱牌的顺序
```

面向对象应用——扑克牌类设计

3. Poke类

```
def deal(self, hands, per_hand = 13): #发牌，发给玩家，每人默认13张牌
    for rounds in range(per_hand):
        for hand in hands:
            if self.cards:
                top_card = self.cards[0]
                self.cards.remove(top_card)
                hand.add(top_card)
                #self.give(top_card, hand)
            else:
                print("Can't continue deal. Out of cards!")
```

面向对象应用——扑克牌类设计

主程序

- 主程序比较简单
- 因为是4个玩家，所以生成players列表存储初始化的4位牌手。
- 生成一副牌（对象实例） poker1
- 调用populate()方法生成有52张牌的一副牌
- 调用shuffle()方法洗牌
- 调用deal()方法发牌
- 最后显示4位牌手所有的牌。

面向对象应用——扑克牌类设计

主程序

```
if __name__ == "__main__":  
    print("This is a module with classes for playing cards.")  
    players = [Hand(),Hand(),Hand(),Hand()]      #四个玩家  
    poke1 = Poke()  
    poke1.populate()                             #生成一副牌  
    poke1.shuffle()                              #洗牌  
    poke1.deal(players,13)                       #发给玩家每人13张牌  
#显示4位牌手的牌  
    n=1  
    for hand in players:  
        print("牌手",n,end=":")  
        print(hand)  
        n=n+1  
    input("\nPress the enter key to exit.")
```