第3章

面向对象设计应用——发牌游戏

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。本章介绍面向对象程序设计中类和对象的定义,类的继承、派生与多态,然后通过扑克牌类设计发牌程序来帮助读者掌握面向对象程序设计的理念。

3.1 发牌游戏功能介绍

在扑克牌游戏中,计算机随机将 52 张牌(不含大小王)发给 4 名牌手,在屏幕上显示每位牌手的牌。本章采用扑克牌类设计扑克牌发牌程序,程序的运行效果如图 3-1 所示。

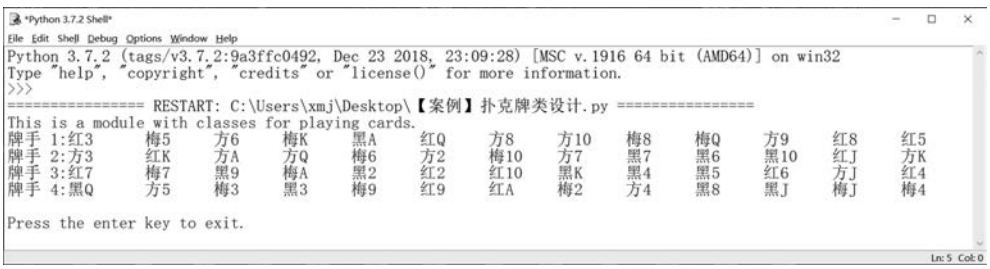

图 3-1 扑克牌发牌程序运行效果

3.2 Python 面向对象设计


视频讲解

现实生活中的每一个相对独立的事物都可以看作一个对象,如一个人、一辆车、一台计算机等。对象是具有某些特性和功能的具体事物的抽象。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；一台计算机由主机、显示器、键盘、鼠标等部件组成，也有开机、运行、关机等行为。

当生产一台计算机的时候，并不是按顺序先生产主机，再生产显示器，再生产键盘、鼠标，而是同时生产和设计主机、显示器、键盘、鼠标等，最后把它们组装起来。将这些部件通过事先设计好的接口进行连接，以便协调工作。这就是面向对象程序设计的基本思路。

每个对象都有一个类型，类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，马路上的汽车都是一个一个的汽车对象，但它们全部归属于汽车类，其中，车身颜色是该类的属性，开动是它的方法，保养或报废是它的事件。

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。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。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。

3.2.1 定义和使用类

1. 类定义

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（成员变量），用函数形式表示的对象行为称为成员函数（成员方法），成员属性和成员方法统称为类的成员。

类定义的最简单形式如下：

```
class 类名:
    属性(成员变量)
    属性
    …
    成员函数(成员方法)
```

例如，定义一个 Person（人员）类。

```
class Person:
    num = 1                  #成员变量(属性)
    def SayHello(self):      #成员函数
        print("Hello!")
```

在 Person 类中定义一个成员函数 SayHello(self)，用于输出字符串"Hello!"。同样，Python 使用缩进标识类的定义代码。

2. 对象定义

对象是类的实例。如果人类是一个类，那么某个具体的人就是一个对象。只有定义了具体的对象，并通过“对象名.成员”的方式才能访问其中的数据成员或成员方法。

Python 创建对象的语法如下：

```
对象名 = 类名()
```

例如，下面的代码定义了一个 Person 类的对象 p：

```
p = Person()
p.SayHello()          # 访问成员函数 SayHello()
```

程序运行结果如下：

```
Hello!
```

3.2.2 构造函数

类可以定义一个特殊的叫作__init__()的方法（构造函数，以两个下画线“__”开头和结束）。一个类定义了__init__()方法以后，类实例化时就会自动为新生成的类实例调用__init__()方法。构造函数一般用于完成对象数据成员设置初值或进行其他必要的初始化工作。如果用户未涉及构造函数，Python 将提供一个默认的构造函数。

例如，定义一个复数类 Complex，构造函数会完成对象变量初始化工作。

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
x = Complex(3.0, -4.5)
print(x.r, x.i)
```

程序运行结果如下：

```
3.0  -4.5
```

3.2.3 析构函数

Python 中类的析构函数是__del__()，用来释放对象占用的资源，在 Python 回收对象空间之前自动执行。如果用户未涉及析构函数，Python 将提供一个默认的析构函数进行必要的清理工作。

例如：

```
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
```

程序运行结果如下：

```
3.0 - 4.5
<_main_.Complex object at 0x01F87C90 >
Complex 不存在了
```

说明：在删除 x 对象变量之前，x 是存在的，在内存中的标识为 0x01F87C90，执行“del x”语句后，x 对象变量不存在了，系统自动调用析构函数，所以出现“Complex 不存在了”。

3.2.4 实例属性和类属性

属性（成员变量）有两种，一种是实例属性，另一种是类属性（类变量）。实例属性是在构造函数__init__中定义的，定义时以 self 作为前缀；类属性是在类中方法之外定义的属性。在主程序中（在类的外部），实例属性属于实例（对象）只能通过对象名访问；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 3-1】 定义含有实例属性（姓名 name，年龄 age）和类属性（人数 num）的 Person 人员类。

```
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)
    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("夏敏捷",42)
P2 = Person("王琳",36)
P1.PrintName()
P2.PrintName()
Person.num = 2                               # 修改类属性
P1.PrintNum()
P2.PrintNum()
```

程序运行结果如下：

```
姓名：夏敏捷 年龄：42
姓名：王琳 年龄：36
2
2
```

num 变量是一个类变量，它的值将在这个类的所有实例之间共享。用户可以在类内部或类外部使用 Person. num 访问。

在类的成员函数(方法)中可以调用类的其他成员函数(方法)，可以访问类属性、对象实例属性。

在 Python 中比较特殊的是，可以动态地为类和对象增加成员，这一点与其他面向对象程序设计语言不同，也是 Python 动态类型特点的一种重要体现。

3.2.5 私有成员和公有成员

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画线“__”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，需要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，对于成员方法也具有同样的性质。这种方式如下：

```
对象名._类名 + 私有成员
```

例如，访问 Car 类私有成员__weight。

```
car1._Car__weight
```

私有属性是为了数据封装和保密而设的属性，一般只能在类的成员方法(类的内部)中使用访问，虽然 Python 支持用一种特殊的方式从外部直接访问类的私有成员，但是并不推荐这样做。公有属性是可以公开使用的，既可以在类的内部进行访问，也可以在外部程序中使用。

【例 3-2】 为 Car 类定义私有成员。

```
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)
print(car1._Car__weight)
print(car1.__weight)            #AttributeError
```

程序运行结果如下：

```
Red
10.5
AttributeError: 'Car' object has no attribute '__weight'
```

3.2.6 方法

在类中定义的方法可以粗略分为3种：公有方法、私有方法、静态方法。其中，公有方法、私有方法都属于对象，私有方法的名字以两个下画线“__”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。如果通过类名来调用属于对象的公有方法，需要为显式的self参数传递一个对象名，用来明确指定访问哪个对象的数据成员。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。

【例3-3】 公有方法、私有方法、静态方法的定义和调用实例。

```
class Person:
    num = 0                                  #类属性
    def __init__(self, str,n,w):             #构造函数
        self.name = str                      #对象实例属性(成员)
        self.age = n
        self.__weight = w                    #定义私有属性__weight
        Person.num += 1
    def __outputWeight(self):                #定义私有方法 outputWeight
        print("体重：",self.__weight)        #访问私有属性__weight
    def PrintName(self):                     #定义公有方法(成员函数)
        print("姓名：", self.name, "年龄：", self.age, end = " ")
        self.__outputWeight()                #调用私有方法 outputWeight
    def PrintNum(self):                      #定义公有方法(成员函数)
        print(Person.num)                    #由于是类属性,所以不写 self.num
    @ staticmethod
    def getNum():                            #定义静态方法 getNum
        return Person.num
#主程序
P1 = Person("夏敏捷",42,120)
P2 = Person("张海",39,80)
P1.PrintName()
P2.PrintName()
Person.PrintName(P2)
print("人数：",Person.getNum())
print("人数：",P1.getNum())
```

程序运行结果如下：

```
姓名：夏敏捷 年龄：42 体重：120
姓名：张海 年龄：39 体重：80
姓名：张海 年龄：39 体重：80
人数：2
人数：2
```

继承是为代码复用和设计复用而设计的，是面向对象程序设计的重要特性之一。在设

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，无疑会大幅减少开发工作量。

3.2.7 类的继承


视频讲解

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。

类继承的语法如下：

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

在 Python 中，类继承的特点如下：

(1) 在继承中基类的构造函数(__init__()方法)不会被自动调用，它需要在其派生类的构造中专门调用。

(2) 需要在派生类中调用基类的方法时，通过“基类名.方法名()”的方式来实现，需要加上基类的类名前缀，且需要带 self 参数变量(在类中调用普通函数时并不需要带 self 参数)。也可以使用内置函数 super()实现这一目的。

(3) Python 总是首先查找对应类型的方法，如果它不能在派生类中找到对应的方法，它才开始到基类中逐个查找(先在本类中查找调用的方法，找不到再去基类中找)。

【例 3-4】 设计 Person 类，并根据 Person 派生 Student 类，分别创建 Person 类与 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
    def setAge(self, age):
        if type(age)!= int:
            print('年龄必须是整数.')
            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)
#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
        #super(Student, self).show()               #也可以这样调用基类 show()方法
        print('入学年份：', self.__schoolyear)
#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()
```

程序运行结果如下：

```
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
```

方法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，即可以在派生类重写基类的方法。

【例 3-5】 重写父类（基类）的方法。

```
class Animal:                                  #定义父类
   def run(self):
      print("Animal is running…")              #调用父类方法
class Cat(Animal):                             #定义子类
   def run(self):
      print("Cat is running…")                 #调用子类方法
class Dog(Animal):                             #定义子类
   def run(self):
      print("Dog is running…")                 #调用子类方法

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

程序运行结果如下：

```
Dog is running…          #调用子类方法
```

当子类 Dog 和父类 Animal 都存在相同的 run()方法时，子类的 run()覆盖了父类的 run()，在代码运行时，总是会调用子类的 run()。这样，就获得了继承的另一个优点：多态。

3.2.8 多态


视频讲解

要理解什么是多态，首先要对数据类型再做一点说明。定义一个类时，实际上就定义了一种数据类型。定义的数据类型和 Python 自带的数据类型(如 string、list、dict)没什么区别。例如：

```
a = list()               #a 是 list 类型
b = Animal()             #b 是 Animal 类型
c = Dog()                #c 是 Dog 类型
```

判断一个变量是否是某个类型，可以用 isinstance()判断：

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

即 a、b、c 分别对应着 list、Animal、Dog 这三种类型。

```
>>> isinstance(c, Animal)
True
```

因为 Dog 是从 Animal 继承下来的，当创建了一个 Dog 的实例 c 时，判定 c 的数据类型是 Dog 没错，但 c 同时也是 Animal 也没错，因为 Dog 本来就是 Animal 的一种。

所以，在继承关系中，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作父类。反之则错误，如下：

```
>>> b = Animal()
>>> isinstance(b, Dog)
False
```

其中，Dog 可以看成 Animal，但 Animal 不可以看成 Dog。

要理解多态的优点，还需要再编写一个函数接受一个 Animal 类型的变量：

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

当传入 Animal 的实例时，run_twice()就输出：

```
>>> run_twice(Animal())
Animal is running…
Animal is running…
```

当传入 Dog 的实例时，run_twice()就输出：

```
>>> run_twice(Dog())
Dog is running…
Dog is running…
```

当传入 Cat 的实例时，run_twice()就输出：

```
>>> run_twice(Cat())
Cat is running…
Cat is running…
```

现在，如果再定义一个 Tortoise 类型，也从 Animal 派生：

```
class Tortoise(Animal):
    def run(self):
        print('Tortoise is running slowly…')
```

当调用 run_twice()时，传入 Tortoise 的实例：

```
>>> run_twice(Tortoise())
Tortoise is running slowly…
Tortoise is running slowly…
```

此时，会发现新增一个 Animal 的子类，不必对 run_twice()做任何修改。实际上，任何依赖 Animal 作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

多态的好处在于，当需要传入 Dog、Cat、Tortoise……时，只需要接收 Animal 类型即可。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，这就是多态的意义。

对于一个变量，只需要知道它是 Animal 类型，无须确切地知道它的子类型，就可以放心地调用 run()方法，而具体调用的 run()方法是作用在 Animal、Dog、Cat 还是 Tortoise 对象上，由运行时该对象的确切类型决定。这就是多态真正的作用：调用方只管调用，不管细节。而当新增一种 Animal 的子类时，只要确保 run()方法编写正确，不用管原来的代码是如何调用的，这就是著名的“开闭”原则，说明如下。

（1）对扩展开放：允许新增 Animal 子类。

（2）对修改封闭：不需要修改依赖 Animal 类型的 run_twice()等函数。


视频讲解

3.3 扑克牌发牌程序设计的步骤

3.3.1 设计类

对发牌程序设计了三个类：Card 类、Hand 类和 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“梅”为梅花，“方”为方块，“红”为红桃，“黑”为黑桃。

其中：

(1) 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。

(2) __str__()方法用来输出牌面大小和花色。

(3) 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 40～52 的顺序编号(未洗牌之前)。也就是说梅花 2 的编号为 2，方块 A 的编号为 14，方块 K 的编号为 26。这个方法是为图形化显示牌面预留的方法。

(4) flip()是翻牌方法，改变牌正面是否显示的属性值。

```
#Cards Module
class Card():
    """ A playing card. """
    RANKS = ["A", "2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K"]
                                        #牌面数字 1～13
    SUITS = ["梅", "方", "红", "黑"]    #"梅"为梅花,"方"为方块,"红"为红桃,"黑"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,True 为牌正面,False 为牌背面

    def __str__(self):                  #重写 print()方法,打印一张牌的信息
        if self.is_face_up:
            rep = self.suit + self.rank
        else:
            rep = "XX"
        return rep

    def pic_order(self):                    #牌的顺序号
        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

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

2. Hand 类

Hand 类代表一手牌（一位玩家手里拿的牌），可以认为是一位牌手手里的牌，其中 cards 列表变量存储牌手手里的牌，可以增加牌、清空手里的牌或把一张牌给别的牌手。

```
class Hand():
    """ A hand of playing cards. """
    def __init__(self):
        self.cards = []                    #cards 列表变量存储牌手的牌
    def __str__(self):                     #重写 print()方法，打印出牌手的所有牌
        if self.cards:
            rep = ""
            for card in self.cards:
                rep += str(card) + "\t"
        else:
            rep = "无牌"
        return rep
    def clear(self):                       #清空手里的牌
        self.cards = []
    def add(self, card):                   #增加牌
        self.cards.append(card)
    def give(self, card, other_hand):                  #把一张牌给别的牌手
        self.cards.remove(card)
        other_hand.add(card)
```

3. Poke 类

Poke 类代表一副牌，可以看作有 52 张牌的牌手，所以继承 Hand 类。由于其中 cards 列表变量要存储 52 张牌，而且要有发牌、洗牌操作，所以增加如下的方法：

（1）populate(self)生成存储了 52 张牌的一手牌，这些牌按梅花 1～13、方块 14～26、红

桃 27～39、黑桃 40～52 的顺序(未洗牌之前)存储在 cards 列表变量中。

(2) shuffle(self)洗牌,使用 Python 的 random 模块 shuffle()方法打乱牌的存储顺序即可。

(3) deal(self, hands, per_hand=13)是完成发牌动作,发给 4 位玩家,每人默认 13 张牌。若给 per_hand 传 10,则每人发 10 张牌,只不过牌没发完。

```
#Poke 类
class Poke(Hand):
    """ 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))
    def shuffle(self):                               #洗牌
        import random
        random.shuffle(self.cards)                   #打乱牌的顺序

    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) #上两句可以用此语句替换
                else:
                    print("不能继续发牌了,牌已经发完!")
```

注意:Python 子类的构造函数默认是从父类继承过来的,所以如果没在子类中重写构造函数,则是从父类调用的。

3.3.2 主程序

主程序比较简单,因为有 4 位玩家,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 poke1,调用 populate()方法生成有 52 张牌的一副牌,调用 shuffle()方法洗牌打乱顺序,调用 deal(players,13)方法发给每位牌手 13 张牌,最后显示 4 位牌手所有的牌。

```
#主程序
if __name__ == "__main__":
    print("This is a module with classes for playing cards.")
    #四位玩家
    players = [Hand(),Hand(),Hand(),Hand()]
    poke1 = Poke()
    poke1.populate()            #生成一副牌
    poke1.shuffle()             #洗牌
    poke1.deal(players,13)      #发给每位牌手 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.")
```

3.4 拓展练习——斗牛扑克牌游戏

3.4.1 斗牛游戏功能介绍

斗牛游戏分为庄家和闲家（即玩家）两位牌手。游戏的规则是：庄家和玩家每人发 5 张牌，庄家和玩家比拼牌面点数大小决定输赢。

1. 计算点数

斗牛游戏计算点数的规则如下：

（1）J、Q、K 牌都算 10 点，A 算 1 点，其余的牌按牌面值计算点数。游戏中不出现大小王牌。

（2）牌局开始时，每人发 5 张牌，庄家和玩家需要将手中任意 3 张牌能组成 10 的倍数（如 A27、334、55J、91K、10JQ、JJK 等），称为“牛”，剩余的两张牌加起来算点数（超过 10 则去掉十位数只留个位数），点数是几点就是牌面点数，也称为牛几（如剩余的两张牌加起来为 5 点，则为“牛 5”），如果剩下两张牌正好是 10 点则为“牛牛”。如果 5 张牌中任意 3 张牌之和都不是 10 的倍数，则牌面点数为 0（即无牛）。

2. 比较大小

大小比较规则如下：

（1）牛牛＞牛 9＞牛 8＞牛 7＞牛 6＞牛 5＞牛 4＞牛 3＞牛 2＞牛 1＞无牛。

（2）计算输赢倍率的规则和点数有关，点数为 10（即牛牛）时，倍率为 3；点数为 7～9 时，倍率为 2；点数为 0～6 时，倍率为 1。

（3）庄家和玩家的点数相同时，需要继续比较各自牌面中最大的牌，花色大小一般规则是按黑桃、红桃、梅花和方块的顺序，所以黑桃 K 最大，方块 A 最小。

3. 游戏过程

每局游戏玩家首先下赌注（欢乐豆数），然后给庄家和玩家各发 5 张牌，游戏显示双方牌手的牌，计算出牌面点数，并比较出谁的牌面点数大。

如果玩家赢则玩家得到（赌注×点数大小对应的倍率）个欢乐豆，玩家输则减去（赌注×点数大小对应的倍率）个欢乐豆。欢乐豆数量小于 0，则游戏结束；或者玩家输入的赌注小于或等于 0 时，则游戏也结束（这时主动退出游戏）。

游戏运行过程如下：

```
游戏开始
请输入玩家的初始欢乐豆：10
第 1 局游戏开始!
请玩家输入赌注:2
庄家的牌为：['K♠', 'A◆', '9♣', 'Q♣', '6◆']
庄家的点数：6
玩家的牌为：['J♣', '5◆', '7♣', '2♥', 'K♣']
玩家的点数：0
这把您输了
您的赌资还剩下：8
游戏继续

第 2 局游戏开始!
请玩家输入赌注:4
庄家的牌为：['8♥', '2♣', '3♠', 'K♥', '10♣']
庄家的点数：3
玩家的牌为：['6♣', '5♣', '7◆', '2♠', '9♠']
玩家的点数：9
这把您赢了
您的赌资还剩下：16
游戏继续

第 3 局游戏开始!
请玩家输入赌注:
```

3.4.2 程序设计的思路

斗牛游戏分为庄家和玩家，所以设计 player 类来创建庄家和玩家这两个对象。player 类封装牌面点数的计算、倍率的计算和获取牌面中最大牌等方法。

在 main()函数中实现整个游戏逻辑，对庄家和玩家的点数进行比较，判断游戏结束的条件是否满足，以及玩家输入赌注和主动退出游戏的功能。

Python 使用 itertools 库提供的 permutations()和 combinations()函数，可以实现元素的排列和组合。例如：

```
from itertools import combinations
test_data = ['a', 'b', 'c', 'd']
print("两个元素的组合")
for i in combinations(test_data, 2):
    print(i,end = ",")
print("\n 三个元素的组合")
for i in combinations(test_data, 3):
    print(i,end = ",")
```

程序运行结果如下：

```
两个元素的组合
('a', 'b'),('a', 'c'),('a', 'd'),('b', 'c'),('b', 'd'),('c', 'd'),
```

```
三个元素的组合
('a', 'b', 'c'),('a', 'b', 'd'),('a', 'c', 'd'),('b', 'c', 'd'),
```

从结果可知，实现了任意两个和三个元素的组合，本程序即采用该方法实现任意 3 张牌的组合并计算牌面点数。

```
for i in permutations(test_data, 2):
    print(i,end = ",")
```

程序运行结果如下：

```
('a', 'b'),('a', 'c'),('a', 'd'),('b', 'a'),('b', 'c'),('b', 'd'),('c', 'a'),('c', 'b'),('c', 'd'),
('d', 'a'),('d', 'b'),('d', 'c'),
```

从结果可知，实现了任意两个元素的全排列。

注意，在 Python 3 中，permutations()和 combinations()函数返回值已经不是 list 列表，而是 iterator 迭代器（是一个对象），所以在需要 list 列表时，需要用 list()函数将 iterator 迭代器转换成 list 列表。例如：

```
list1 = list(permutations(test_data, 2))     #得到列表
```

3.4.3 程序设计的步骤

1. 设计 player 类

在 player 类中，构造函数定义了 self.poker 和 self.bet_on 属性，它们分别存储每位牌手的 5 张牌及其下的赌注。sum_value(self)方法计算出 5 张牌中每张牌的点数。poker_point(self) 计算是牛几，即点数是几。当双方的点数相同时，需要比较庄家和玩家手中最大牌面的牌，此时调用 sorted_index(self) 方法获取牌手手中最大牌面的牌。level_rate(self) 计算点数大小对应的倍率。

```
import itertools
import random
class player(object):                                    #player 类
    def __init__(self, poker,bet_on):                   #构造函数
        super(player, self).__init__()
        self.poker = poker                               #自己的 5 张牌
        self.bet_on = bet_on                             #赌注
    def sum_value(self):                                 #计算出 5 张牌中每张牌的点数
        L = []
        for i in self.poker:                             #i 是'J♣''10♥'这样的字符串
            #if i[0] == 'J' or i[0] == 'Q' or i[0] == 'K' or i[:2] == '10':
            if i[:-1] in ['10','J','Q','K']:             #判断牌面是否是 10、J、Q、K
                num = 10
            elif i[0] == 'A':
```

```
                num = 1
            else:
                num = int(i[0])
            L.append(num)
        return L
    #计算牛几,即点数是几
    def poker_point(self):
        lst = self.sum_value()
        #排列组合: 三张牌的点数之和为 10 的倍数
        maxn = 0
        for j in itertools.combinations(lst, 3):        #任意三张牌组合
            if sum(j) % 10 == 0:                        #三张可以组合成牛牌
                k = sum(lst) % 10                       #根据总点数计算余下两张牌的点数和
                if k == 0:                              #即牛牛
                    return 10
                if k > maxn:                            #即牛 k
                    maxn = k
        return maxn
    #获取最大牌的索引,索引值大则牌面就大
    def sorted_index(self):                             #找 5 张牌中牌面最大的牌
        L = []
        for i in self.poker:
            index = poker_list().index(i)               #获取某张牌在一副牌没洗牌前的索引
            L.append(index)
        return max(L)                                   #获取 5 张牌中最大的索引值
    def level_rate(self):                               #点数的大小对应的倍率
        point = self.poker_point()
        if point == 10:
            self.bet_on *= 3
        elif 7 <= point < 10:
            self.bet_on *= 2
        elif point < 7:
            self.bet_on *= 1
        return self.bet_on
```

2. 生成所有的牌

生成一副牌中的 52 张花色牌。

```
def poker_list():                                   #生成所有的牌
    color = ['◆', '♣', '♠', '♥']                    #['\u2666', '\u2663', '\u2660','\u2665']
    num_poker = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']
    #Joker = ['big_Joker','small_Joker']
    sum_poker = [i + j for i in num_poker for j in color]
    return sum_poker
```

3. 设计主程序

实现游戏的逻辑,玩家下赌注后每次都重新生成一副原始牌的 52 张牌,洗牌后,按顺序

发给庄家 5 张牌(前 5 张牌)和玩家 5 张牌(一副牌的第 6～10 张牌)。按游戏点数规则判断输赢,计算出玩家剩余欢乐豆。如果欢乐豆数量小于 0 则游戏结束；或者玩家输入的赌注小于或等于 0 时,游戏结束(玩家主动退出游戏)。

```
def main():
    print("游戏开始")
    value = int(input('请输入玩家的初始欢乐豆: '))
    i = 1                                                    #统计玩家游戏局数
    while True:
        print('第 %s 局游戏开始!' % i)
        bet_on = input('请玩家输入赌注:')
        bet_on = int(bet_on)
        #输入的赌注小于或等于 0 时,游戏结束(玩家主动退出游戏)
        if bet_on <= 0:
            print('游戏结束')
            print('您剩下的欢乐豆为: ',value)
            return
        else:
            L = poker_list()                                 #重新生成一副原始牌的 52 张牌
            random.shuffle(L)                                #把牌随机洗好
            #按顺序发牌(因为牌已经打乱顺序,所以顺序发牌)
            hostpoker = L[:5]                                #一副牌的前 5 张牌
            host_value = player(hostpoker,bet_on)    #创建庄家对象,把 5 张牌和赌注作为参数
            host_sumPoint = host_value.poker_point()  #庄家牌面的点数
            print("庄家的牌为:", hostpoker)
            print("庄家的点数:", host_sumPoint)

            poker = L[5:10]                                  #一副牌的第 6～10 张牌
            player_value = player(poker,bet_on)      #创建玩家对象,把 5 张牌和赌注作为参数
            player_sumPoint = player_value.poker_point()     #玩家牌面的点数
            print("玩家的牌为:", poker)
            print("玩家的点数:", player_sumPoint)

            #比较双方的点数,然后按照情况更改赌注和欢乐豆
            #点数相同时,比较牌面中最大的牌
            if player_sumPoint == host_sumPoint:
                if player_value.sorted_index() > host_value.sorted_index():
                    print("这把您赢了")
                    value += player_value.level_rate()
                else:
                    print("这把您输了")
                    value -= host_value.level_rate()
            elif player_sumPoint > host_sumPoint:
                print("这把您赢了")
                value += player_value.level_rate()
            else:
                print("这把您输了")
                value -= host_value.level_rate()
            #判断游戏结束的条件
```

```
            if value > 0:                          # 欢乐豆数量
                print('您的赌资还剩下: ',value)
                print('游戏继续\r\n')
            else:
                print('您已经输光了欢乐豆,游戏结束!!!')
                return
        i += 1 # 游戏局数加 1
# 主程序
if __name__ == '__main__':
    main()
```

至此,完成了斗牛扑克牌游戏设计。

思考与练习

使用面向对象设计思想重新设计背单词软件,功能要求如下。

(1) 录入单词。输入英文单词及相应的中文意思,例如:

```
China  中国
Japan  日本
```

(2) 查找单词的中文或英文意思(即输入中文可查对应的英文意思,输入英文可查对应的中文意思)。

(3) 随机测试,每次测试 5 题,系统随机显示英文单词,用户输入中文意思,要求能够统计回答的准确率。

提示:可以设计 word 类,实现单词信息的存储。