《Python统计计算》

（2021年秋季学期）

翟祥
北京林业大学

E-mail: zhaixbh@126.com

编程基础

第6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，并且使得代码具有更好的可读性和可扩展性。
- 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。
- 类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。
- 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

- Python完全采用了面向对象程序设计的思想，是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。
- 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。

6.1.1 类定义语法

- Python使用class关键字来定义类，class关键字之后是一个空格，然后是类的名字，再然后是一个冒号，最后换行并定义类的内部实现。类名的首字母一般大写。例如：

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

或

```
class Car(object): #新式类必须有至少一个基类
    def infor(self):
        print(" This is a car ")
```

6.1.1 类定义语法

- 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：

```
>>> 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
```

6.1.1 类定义语法

- 最后，Python提供了一个关键字“pass”，类似于空语句，可以用在类和函数的定义中或者选择、循环结构中。当暂时没有确定如何实现功能，或者为以后的软件升级预留空间，或者其他类型功能时，可以使用该关键字来“占位”。例如下面的代码都是合法：

```
>>> class A:
        pass

>>> def demo():
        pass

>>> if 5>3:
        pass
```

6.1.2 self参数

- 类的所有实例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”参数代表将来要创建的对象本身
- 在类的实例方法中访问实例属性时需要以“self”为前缀
- 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对象方法则需要显式为self参数传值。

6.1.2 self参数

- Python的类中定义实例方法时将第一个参数定义为“self”只是一个习惯，而实际上类的实例方法中第一个参数的名字是可以变化的，而不必使用“self”这个名字，例如下面的代码：

```
>>> class A:
        def __init__(hahaha, v):
            hahaha.value = v
        def show(hahaha):
            print(hahaha.value)
>>> a = A(3)
>>> a.show()
3
```

6.1.3 类成员与实例成员

- 这里主要指数据成员，或者属性。可以说属性有两种，一种是实例属性，另一种是类属性
- 实例属性一般是指在构造方法__init__()中定义的，定义和使用时必须以self作为前缀
- 类属性是在类中所有方法之外定义的数据成员
- 在主程序中（或类的外部），实例属性属于实例(对象)，只能通过对象名访问
- 而类属性属于类，可以通过类名或对象名访问

6.1.3 类成员与实例成员

- 在类的方法中可以调用类本身的其他方法，也可以访问类属性以及对象属性。
- 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。

6.1.3 类成员与实例成员

```
import types
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")
car2 = Car("Blue")
print(car1.color, Car.price)
Car.price = 110000 #修改类属性
Car.name = 'QQ'     #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)
def setSpeed(self, s):
    self.speed = s
#动态为对象增加成员方法
car1.setSpeed = types.MethodType(setSpeed, Car)
#Car.setSpeed=setSpeed      #或者采用此方法，动态添加方法
car1.setSpeed(50)           #调用对象的成员方法
print(car1.speed)
```

6.1.3 类成员与实例成员

- 在Python中，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          #前缀用对象变量，绑定的是function，无法自动传递self参数
>>> t.test
<function test at 0x0000000034B7EA0>
>>> t.test(t, 3)          #调用时需要为self变量传递对象名
>>> print(t.value)
3
>>> t.test = types.MethodType(test, t)  #此方法绑定的是method，能自动传递self参数
>>> t.test
<bound method test of <__main__.Demo object at 0x000000000074F9E8>>
>>> t.test(5)             #自动传递self参数
>>> print(t.value)
5
```

```
>>> class Test:
        pass

>>> t=Test()

>>> def test(self,v):
      self.value=v


>>> Test.test=test  #前缀用类名，绑定的是method
>>> t.test(5)
>>> t.test
<bound method test of <__main__.Test object at
0x0000000031375C0>>
```

6.1.4 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。
- 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支持一种特殊的方式来从外部直接访问类的私有成员，但是并不推荐您这样做。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。

6.1.4 私有成员与公有成员

- ”单下划线”开始的成员变量叫做保护变量，意思是只有类对象和子类对象自己能访问到这些变量；模块中，单下划线开始的变量不能用’from module import *’导入
- "双下划线" 开始的是私有成员，意思是只有类自己能访问，连子类也不能访问到这个数据。

```
>>> 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
```

6.1.4 私有成员与公有成员

- 在IDLE环境中，在对象或类名后面加上一个圆点“.”，稍等一秒钟则会自动列出其所有公有成员，模块也具有同样的特点。
- 而如果在圆点“.”后面再加一个下划线，则会列出该对象或类的所有成员，包括私有成员。

6.1.4 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。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，只有类对象和子类对象能访问这些成员；
  - __xxx：类中的私有成员，只有类自己能访问，子类也不能访问到这个成员，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。Python中不存在严格意义上的私有成员。
  - __xxx__：系统定义的特殊成员；

6.1.4 私有成员与公有成员

- 另外，在IDLE交互模式下，一个下划线“_”表示解释器中最后一次显示的内容或最后一次语句正确执行的输出结果。例如：

```
>>> 3 + 5
8
>>> _ + 2
10
>>> _ * 3
30
>>> _ / 5
6
>>> 3
3
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    1/0
ZeroDivisionError: integer division or modulo by zero
>>> _
3
```

6.1.4 私有成员与公有成员

```
>>> class Fruit:
        def __init__(self):
            self.__color = 'Red'
            self.price = 1
>>> apple = Fruit()
>>> apple.price #显示对象公开数据成员的值
1
>>> apple.price = 2 #修改对象公开数据成员的值
>>> apple.price
2
>>> print(apple.price, apple._Fruit__color) #显示对象私有数据成员的值
2 Red
>>> apple._Fruit__color = "Blue" #修改对象私有数据成员的值
>>> print(apple.price, apple._Fruit__color)
2 Blue
>>> print(apple.__color) #不能直接访问对象的私有数据成员，出错
AttributeError: Fruit instance has no attribute '__color'
```

6.2 方法

- 在类中定义的方法可以粗略分为四大类：公有方法、私有方法、静态方法和类方法。
- 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数据成员。

- 静态方法和类方法都可以通过类名或对象名调用，但不能直接访问属于对象的成员，只能访问属于类的成员。
- 一般将"cls"作为类方法的第一个参数名称，但也可以使用其他的名字作为参数，并且在调用类方法时不需要为该参数传递值。

6.2 方法

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

        def show(self):
                print('self.__value:', self.__value)
                print('Root.__total:', Root.__total)

        @classmethod
        def classShowTotal(cls): #类方法
                print(cls.__total)

        @staticmethod
        def staticShowTotal(): #静态方法
                print(Root.__total)
```

6.2 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 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实例方法并访问实例成员
self.__value: 3
Root.__total: 2
>>> r.show()
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
>>> rr.show()
self.__value: 5
Root.__total: 2
```

6.3 属性

- Python 2.x和Python 3.x对属性的实现和处理方式不一样，内部实现有较大的差异，使用时应注意二者之间的区别。

6.3.1 Python 2.x中的属性

- 在Python 2.x中，使用@property或property()来声明一个属性，然而属性并没有得到真正意义的实现，也没有提供应有的访问保护机制。正如前面所说，在Python中，可以为类和对象动态增加新成员。在Python 2.x中，为对象增加新的数据成员时，将隐藏同名的已有属性。
- #以下代码在Python 2.x中运行

```
>>> class Test:
        def __init__(self, value):
                self.__value = value
        @property
        def value(self):
                return self.__value
>>> a = Test(3)
>>> a.value
3
>>> a.value = 5 #动态添加了新成员，隐藏了定义的属性
>>> a.value
5
>>> a._Test__value#原来的私有变量没有改变
3
```

6.3.1 Python 2.x中的属性

- 除了动态增加成员时会隐藏已有属性，下面的代码从表面看来是修改属性的值，而实际上也是增加了新成员，从而隐藏了已有属性。

```
>>> 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
```

6.3.1 Python 2.x中的属性

```
>>> t = Test(3)
>>> t.value
3
>>> t.value += 2 #动态添加了新成员
>>> t.value #这里访问的是新成员
5
>>> t.show() #访问原来定义的私有数据成员
3
>>> del t.value #这里删除的是刚才添加的新成员
>>> t.value #访问原来的属性
3
>>> del t.value #试图删除属性
AttributeError: Test instance has no attribute 'value'
>>> del t._Test__value #删除私有成员
>>> t.value #访问属性，但该属性对应的私有成员已不存在
AttributeError: Test instance has no attribute '_Test__value'
```

6.3.1 Python 2.x中的属性

```
>>> class Test:
        def show(self):
            print self.value
            print self.__v
>>> t = Test()
>>> t.show()
AttributeError: Test instance has no attribute 'value'
>>> t.value = 3          #添加新的数据成员
>>> t.show()
3
AttributeError: Test instance has no attribute '_Test__v'
>>> t.__v = 5
>>> t.show()
3
AttributeError: Test instance has no attribute '_Test__v'
>>> t._Test__v = 5     #添加私有数据成员
>>> t.show()
3
5
```

6.3.2 Python 3.x中的属性

- 在Python 3.x中，属性得到了较为完整的实现，支持更加全面的保护机制。例如下面的代码所示，如果设置属性为只读，则无法修改其值，也无法为对象增加与属性同名的新成员，同时，也无法删除对象属性。

6.3.2 Python 3.x中的属性

```
>>> 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
>>> 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
```

6.3.2 Python 3.x中的属性

- 下面的代码则把属性设置为可读、可修改，而不允许删除。

```
>>> 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)
>>> 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
```

6.3.2 Python 3.x中的属性

- 也可以将属性设置为可读、可修改、可删除。

```
>>> 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)
```

6.3.2 Python 3.x中的属性

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

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

    @property   #使用属性函数
    def value(self):
        return self.__value

    @value.setter  #set方法装饰符
    def value(self,value):
        self.__value=value

if __name__ == '__main__':
    t=Test(3)
    print(t.value)
    t.value=5
    print(t.value)
    print(t._Test__value)
```

执行结果：
3
5
5

6.4.1 常用特殊方法

- 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。
  - 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，可以通过为构造函数定义默认值参数来实现类似于其他语言中构造函数重载的目的。
  - 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。
  - 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。

6.4.1 常用特殊方法

方法	功能说明
__init__()	构造函数，生成对象时调用
__del__()	析构函数，释放对象时调用
__add__()	+
__sub__()	-
__mul__()	*
__div__()、truediv__()	/
__floordiv__()	整除
__mod__()	%
__pow__()	**
__cmp__()	比较运算
__repr__()	打印、转换
__setitem__()	按照索引赋值
__getitem__()	按照索引获取值
__len__()	计算长度
__call__()	函数调用
__contains__()	测试是否包含某个元素
__eq__()、__ne__()、__lt__() __le__()、__gt__()	==、!=、<、<=、>
__str__()	转化为字符串
__lshift__()	<<
__and__()	&
__iadd__()	+=

6.4.2 案例精选

- 在MyArray.py文件中，定义了一个数组类，重写了一部分特殊方法以支持数组之间、数组与整数之间的四则运算以及内积、大小比较、成员测试和元素访问等运算符。

6.4.2 案例精选

```
>>> import MyArray
>>> a = MyArray.MyArray(1, 2, 3, 4, 5, 6)
>>> b = MyArray.MyArray(6, 5, 4, 3, 2, 1)
>>> len(a)
6
>>> a.dot(b)
56
>>> a < b
True
>>> a > b
False
>>> a == a
True
>>> 3 in a
True
```

6.4.2 案例精选

```
>>> a * 3
[3, 6, 9, 12, 15, 18]
>>> a + 2
[3, 4, 5, 6, 7, 8]
>>> a ** 2
[1, 4, 9, 16, 25, 36]
>>> a / 2
[0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
>>> a
[1, 2, 3, 4, 5, 6]
>>> a[0] = 8
>>> a
[8, 2, 3, 4, 5, 6]
```

6.5 继承机制

- 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。当我们设计一个新类时，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非私有成员。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。

6.5 继承机制

例1：设计Person类，并根据Person派生Teacher类，分别创建Person类与Teacher类的对象。

```
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) != types.StringType:
            print 'name must be string.'
            return
        self.__name = name

    def setAge(self, age):
        if type(age) != types.IntType:
            print 'age must be integer.'
            return
        self.__age = age
    def setSex(self, sex):
        if sex != 'man' and sex != 'woman':
            print 'sex must be "man" or "woman"'
            return
        self.__sex = sex
    def show(self):
        print self.__name
        print self.__age
        print self.__sex
```

6.5 继承机制

```
class Teacher(Person):
    def __init__(self, name='', age = 30, sex = 'man', department = 'Computer'):
        #调用基类构造方法初始化基类的私有数据成员
        super(Teacher, self).__init__(name, age, sex)
        #Person.__init__(self, name, age, sex) #也可以这样初始化基类的私有数据成员
        self.setDepartment(department) #初始化派生类的数据成员

    def setDepartment(self, department):
        if type(department) != types.StringType:
            print 'department must be a string.'
            return
        self.__department = department

    def show(self):
        super(Teacher, self).show()
        print self.__department
```

6.5 继承机制

```
if __name__ =='__main__':
    zhangsan = Person('Zhang San', 19, 'man')
    zhangsan.show()
    lisi = Teacher('Li Si', 32, 'man', 'Math')
    lisi.show()
    lisi.setAge(40) #调用继承的方法修改年龄
    lisi.show()
```

6.5 继承机制

- 为了更好地理解Python类的继承机制，我们来看下面的Python 2.7.11代码，请认真体会构造函数、私有方法以及普通公开方法的继承原理。

```
>>> class A():
        def __init__(self):
            self.__private()
            self.public()
        def __private(self):
            print '__private() method of A'
        def public(self):
            print 'public() method of A'
>>> class B(A):
        def __private(self):
            print '__private() method of B'
        def public(self):
            print 'public() method of B'
```

6.5 继承机制

```
>>> b = B() #自动调用从基类A继承的构造函数
__private() method of A
public() method of B
>>> print '\n'.join(dir(b)) #查看对象b的成员
_A__private
_B__private
__doc__
__init__
__module__
Public
```

6.5 继承机制

```
>>> class C(A):
        def __init__(self):
            self.__private()
            self.public()
        def __private(self):
            print '__private() method of C'
        def public(self):
            print 'public() method of C'
>>> c = C() #自动调用派生类自己的构造函数
__private() method of C
public() method of C
>>> print '\n'.join(dir(c))
_A__private
_C__private
__doc__
__init__
__module__
public
```

谢谢Q/A