



面向对象程序设计





内容

- 面向对象概念回顾
- 类和实例
- 成员
- 继承和多态
- 综合案例



面向对象程序设计

面向对象 (Object Oriented)

- 面向对象是认识事物的一种方法，是以对象为中心的认识方式
- 面向对象程序设计 (OO Programming) 主要是针对大型软件设计提出的，也是一种编程范式



类和对象

- 类是对一类具有共同特征的事物的抽象。类概括事物的属性（成员变量，类的属性）和可以进行的操作（成员函数，类的方法），代表一类事物的共同特点
- 类的实例化即对象，是具体的事物个体



面向对象程序设计的基本特征

- 封装性：将数据和行为（或功能）相结合，形成一个有机的整体——类。通过类的封装隐藏对象的属性和实现细节，仅对外公开接口，实现“高内聚，低耦合”
- 继承性：子类可以继承父类的数据和行为
- 多态性：同一操作（一个接口形式）作用于不同的对象，产生不同的结果（多种实现方法）



Python与面向对象

Python在设计之初，就定位为面向对象的编程语言，完全采用了面向对象程序设计的思想

- “一切皆对象”：数字，字符串，元组，列表，字典，函数，方法，类，模块，代码
- 完全支持面向对象的基本功能：封装，继承，多态，对基类方法的覆盖或重写，是真正面向对象的高级动态编程语言



内容

- 面向对象概念回顾
- 类和实例
- 成员
- 继承和多态
- 综合案例



类的定义

`class <类名>:`

 多个 (≥ 0 个) 属性

 多个 (≥ 0 个) 方法

根据推荐参考惯例，类名的首字母一般大写。如果采用多个单词的组合，则每个单词的首字母大写。如：

Teacher, TheFistDemo



Python中类的属性

`class <类名>:`

多个 (≥ 0 个) 属性

多个 (≥ 0 个) 方法



简单的类定义的示例

```
class Teacher:
    """一个简单的教师类示例"""
    profession = 'education'

    def show_info(self):
        print('This is a teacher')
```

当类定义正常结束时，会创建出一个类



类的实例化

类的实例化指根据类创建具体的实例，即实例对象（对象）

- 类的实例化使用函数表示法

```
teacher_zhang = Teacher()  
teacher_wang = Teacher()
```



类的实例化

- 创建实例对象后，可通过“对象名.成员”的方式访问其数据成员及成员方法

```
>>> teacher_zhang.profession  
'education'  
>>> teacher_zhang.show_info()  
This is a teacher
```

```
class Teacher:  
    """一个简单的教师类示例"""  
    profession = 'education'  
  
    def show_info(self):  
        print('This is a teacher')
```



类的实例化

- 可以通过内置函数isinstance测试一个对象是否是某个类的实例

```
>>> isinstance(teacher_zhang, Teacher)
True
```

```
>>> isinstance(teacher_zhang, dict)
False
```



构造方法

构造方法也称“构造器”，指当实例化一个对象（创建一个对象）时，第一个被调用的方法

```
class <类名>:
```

```
    def __init__(self[, para...]):
```



构造方法

构造方法的作用之一是完成数据成员的初始化

```
class Person:
    def __init__(self, name, age, height, weight):
        self.name = name
        self.age = age
        self.height = height
        self.weight = weight
```

```
>>> print(han.name, han.age)
      Hanmeimei 20
```

```
>>> han = Person('Hanmeimei', 20, 168, 55)
```



self参数

- self代表即将被创建的对象自身
- 在类的实例方法中要访问其数据属性时需要以self为前缀
- 类的所有实例方法都至少有一个为self的参数，且必须是方法的第一个形参
- 在外部通过实例（对象）调用实例方法时，不需要为self传递参数；通过类调用实例方法时需要显式为self传值
- 在python中，以“self”代表对象自身只是一个习惯，实际上并不必须使用“self”这个名字

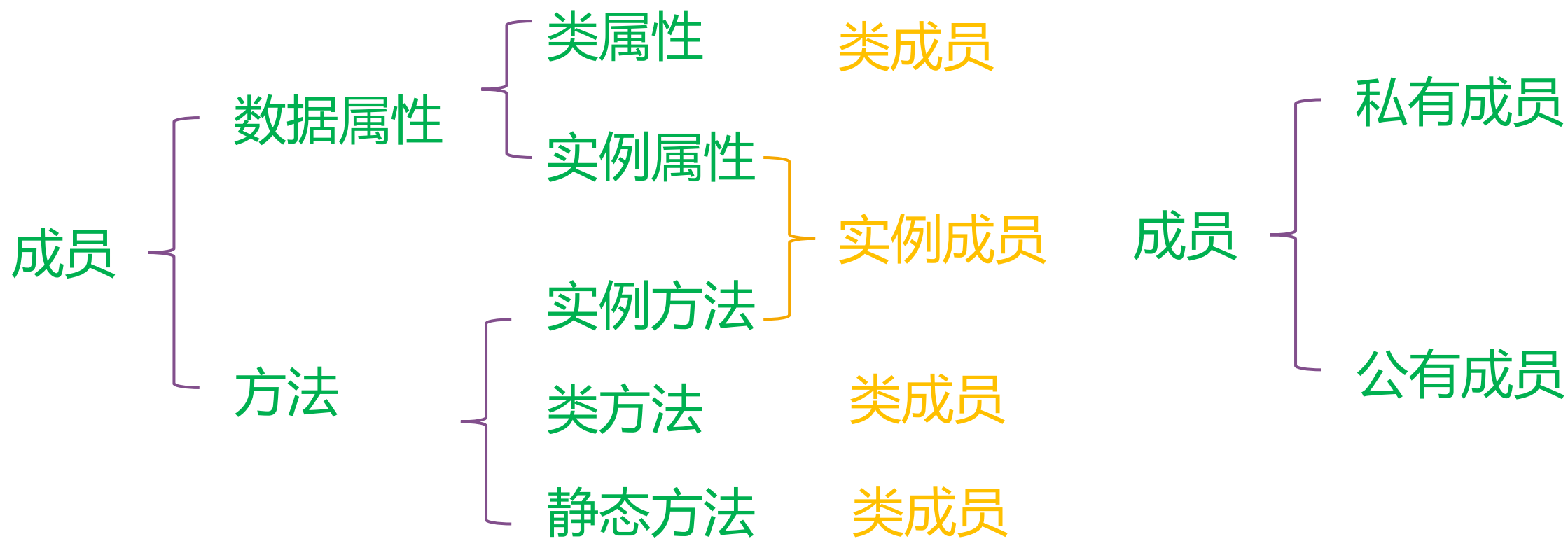


内容

- 面向对象概念回顾
- 类和实例
- 成员
- 继承和多态
- 综合案例



Python中的成员（属性）





数据属性

数据属性用于说明类或对象的一些特性

- 不需要声明，在第一次被赋值时产生
- 分为属于类的属性（类属性）和属于实例的属性（实例属性）



类属性和实例属性

类属性和实例属性都定义在类中，根本区别在于保存位置和调用对象不同

- 类属性在类定义中所有方法之外定义；实例属性一般在构造函数 `__init()` 中定义，在类中定义和使用时必须以 `self` 为前缀
- 类属性被类的所有实例共有，在内存中只有一个副本；实例属性为各个实例私有，互不干扰
- 类属性可以通过类或实例访问；实例属性只能通过实例访问



类属性和实例属性

```
class Person:
    count = 0 #count:类属性
    def __init__(self, name, age):
        self.name = name #name:实例属性
        self.age = age #age:实例属性
        Person.count += 1 #通过类访问类属性
```

```
>>> han = Person('Han Meimei', 13)
>>> li = Person('Li Lei', 14)
>>> han.name
'Han Meimei'
>>> li.name
'Li Lei'
>>> Person.count
2
>>> han.count
2
>>> li.count
2
```



类的方法

类的方法用于描述对象所具有的行为，分为三类：

- 实例方法（对象方法）：类的方法默认是实例方法，定义时不需要特殊的关键字进行标识。以self作为第一个参数，代表实例自身
- 类方法：使用@classmethod说明。以cls作为第一个参数，代表类自身
- 静态方法：使用@staticmethod说明

类方法和静态方法都可以通过类名和对象名调用，实例方法只能通过对象名调用



类方法

```
class Person:
    _location = 'Asia'          #类属性
    def __init__(self, name, age): #实例方法
        self.name = name
        self.age = age

    @classmethod                 #类方法
    def set_location(cls, location):
        cls._location = location

    @classmethod                 #类方法
    def get_location(cls):
        return cls._location
```

```
>>> Person.get_location()
'Asia'
>>> Person.set_location('Europe')
>>> Person.get_location()
'Europe'
```



类方法

```
class Person:
    _location = 'Europe'
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @classmethod
    def set_location(cls, location):
        cls._location = location

    @classmethod
    def get_location(cls):
        return cls._location
```

```
>>> han = Person('Han Meimei', 13)
>>> li = Person('Li Lei', 14)
>>> han.get_location()
'Europe'
>>> Person.get_location()
'Europe'

>>> han.set_location('Africa')
>>> han.get_location()
'Africa'
>>> Person.get_location()
'Africa'
>>> li.get_location()
'Africa'
```




静态方法

```
class Student():  
    def __init__(self, name, age, height, weight): #实例方法  
        self.name = name  
        self.age = age  
        self.height = height  
        self.weight = weight
```

```
    @staticmethod                                #静态方法  
    def computeBMI(height, weight):  
        return round(weight / (height / 100) ** 2, 2)  
  
    def getBMI(self):                             #实例方法  
        return self.computeBMI(self.height, self.weight)
```

```
>>> stu_han.getBMI()  
17.63
```

```
>>> Student.computeBMI(180, 72)  
22.22
```

```
>>> stu_han = Student('Han Meimei', 13, 165, 48)
```



静态方法

```
class Person:
    _location = 'Asia'
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @staticmethod
    def set_location(location):
        Person._location = location

    @staticmethod
    def get_location():
        return Person._location
```

```
>>> Person.get_location()
'Asia'
>>> Person.set_location('Europe')
>>> Person.get_location()
'Europe'
```



使用类方法处理类层级中每个类的数据

```
class Person:
    numInst = 0
    @classmethod
    def count(cls):
        cls.numInst += 1
    def __init__(self):
        self.count()

class Men(Person):
    numInst = 0
    def __init__(self):
        Person.__init__(self)

class Women(Person):
    numInst = 0
```

```
>>> p1 = Person()
>>> m1, m2 = Men(), Men()
>>> w1, w2, w3 = Women(), Women(), Women()

>>> Person.numInst, Men.numInst, Women.numInst
(1, 2, 3)

>>> p1.numInst, m1.numInst, w1.numInst
(1, 2, 3)
```



静态方法处理类层级中每个类的数据

```
class Person:
    numInst = 0
    @staticmethod
    def count():
        Person.numInst += 1
    def __init__(self):
        self.count()
```

```
class Men(Person):
    numInst = 0
    def __init__(self):
        Person.__init__(self)
```

```
class Women(Person):
    numInst = 0
```

```
>>> p1 = Person()
>>> m1, m2 = Men(), Men()
>>> w1, w2, w3 = Women(), Women(), Women()
>>> Person.numInst, Men.numInst, Women.numInst
(6, 0, 0)
```



Python类成员的动态增删

Python中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

```
class Person:
    count = 0  #count:类属性
    def __init__(self,name,age):
        self.name = name  #name:实例属性
        self.age = age    #age:实例属性
        Person.count += 1 #通过类访问类属性
```

```
>>> han = Person('Han Meimei', 13)
>>> li = Person('Li Lei', 14)
>>> han.count = 5
```



私有成员和公有成员

- 在定义类的成员时，如果成员名以两个下划线 “__” 或更多下划线开头且不以两个下划线或更多下划线结尾，表示是私有成员
- 公有成员既可以在类的内部进行访问，也可以在外部程序中使用
- 私有成员在类的外部不能直接访问，需要通过调用对象的公开成员方法访问，但可以[通过Python支持的特殊方式进行访问](#)

[Python并没有对私有成员提供严格的访问保护机制](#)



私有成员和公有成员

```
class A:
    def __init__(self, value1 = 1, value2 = 2):
        self.value1 = value1
        self.__value2 = value2

    def getValue1(self):
        return self.value1

    def getValue2(self):
        return self.__value2
```

```
>>> a = A()
>>> a.value1
1
>>> a.getValue2()
2
>>> a._A__value2
2
```



有特殊含义的成员名

Python中，以下划线开头的变量名或方法名有特殊含义，尤其是在类的定义中

- `_xxx`: 受保护成员，不能用“`from module import`”导入
- `__xxx__`: 系统定义的特殊成员
- `__xxx`: 私有成员，只有类对象自己可以访问，子类对象不能直接访问，但在外部可以通过特殊方式访问



内置的特殊方法

和自定义类相关的常见特殊方法

- `__new__()`: 创建一个类的新实例
- `__init__()`: 实例初始化方法, 相当于构造函数
- `__del__()`: 销毁实例, 相当于析构函数
- `__str__()`: 返回一个字符串, 是对实例的描述



内容

- 面向对象概念回顾
- 类和实例
- 属性
- 继承和多态
- 综合案例



继承

- 继承指在设计一个新类时，继承一个已有类的定义及其成员，再进行二次开发
- 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

`class <派生类名> (基类名) :`

 多个 (≥ 0 个) 属性

 多个 (≥ 0 个) 方法



继承 (inheritance)

- 子类继承父类所有的公有成员（数据属性和方法）
- 子类不能继承父类的私有成员
- 子类可以扩展自己的成员，也可以重写（覆盖）父类的公有成员，但无法覆盖父类的私有成员
- 访问子类成员时，子类总是先在本类中查找，如果找不到，才会在基类中查找



继承示例

```
class Person(object):    #object是所有类的最顶层基类，可以不写
    __count = 0          #私有变量
    def __init__(self, name = "", age = 20):
        self.name = name
        self.age = age
        Person.__count += 1
        self.__show_count() #调用本类的私有方法
    def show_info(self):    #公有方法
        print('Person:', self.name, '; age', self.age)
    @classmethod
    def __show_count(cls): #私有方法
        print(cls.__count)
```



继承示例

```
class Teacher(Person):
    def __init__(self, name = "", age = 30, department = 'SCS'):
        super(Teacher, self).__init__(name, age) #在派生类中调用父类
        # 或可使用下列方法
        # Person.__init__(self,name,age)
        self.__department = department #创建实例时增加私有变量
    def show(self):
        super(Teacher, self).show_info() #调用父类的公有方法
        print('Department:',self.__department) #打印私有变量值
    def change_department(self, department):
        self.__department = department
    def show_info(self):
        print('Teacher:',self.name,', age:',self.age,', Department:',self.__department) # 重写父类中的方法
```



子类对父类成员的继承和覆盖

```
class A:
```

```
    def __init__(self):
```

```
        self.__private()
```

```
        self.public()
```

```
    def __private(self):
```

```
        print('__private() method in A')
```

```
    def public(self):
```

```
        print('public() method in A')
```

```
class B(A):
```

```
    def __private(self):
```

```
        print('__private() method in B')
```

```
    def public(self):
```

```
        print('public() method in B')
```

```
>>> b = B()
```



多态 (polymorphism)

- 多态即“多种表现形式”，指在父类中定义的属性和方法被子类继承之后，可以具有不同的数据类型或表现出不同的行为，这使得同一个属性或方法在父类及其各个子类中具有不同的含义
- 通过多态实现“一个接口，多种实现”
- 关于多态的两种理解：
 - ✓ 重写 (overriding)：派生类对基类的允许访问的方法的实现过程进行重新编写，且返回值和形参都不能改变
 - ✓ 重载 (overloading)：同名方法有不同的属性个数或类型



Python中的多态性

Python是一种弱类型的语言，这使得其多态性的表现比之传统面向对象编程语言的多态有所不同。和Java的多态性相比：

- 不需要也不体现父类引用指向子类对象的多态性
- 不支持方法的重载



“鸭子类型”和Python中的多态性

在程序设计中，鸭子类型(duck typing)是动态类型的一种风格。在这种风格中，一个对象有效的语义，不是由继承自特定的类或实现特定的接口，而是由当前方法和属性的集合决定。这个概念的名字来源于由James Whitcomb Riley提出的鸭子测试：

“当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。”

Python中的多态是“鸭子类型”的体现

- 关注的不是对象的类型本身，而是它是如何使用的
- 并不依靠查找对象类型来确定其是否具有正确的接口，而是直接调用或使用其方法



Python中的多态性

Python中的多态性：让具有不同功能的函数可以使用相同的函数名，这样就可以用一个函数名调用不同内容(功能)的函数

- 只关心对象的实例方法是否同名，不关心对象所属的类型
- 对象所属的类之间，继承关系可有可无
- 多态是调用方法的技巧，不会影响到类的内部设计



多态性示例：通过继承实现多态

```
class Animal:
    def eat(self):
        print("Animal like eat anything")
class Cat(Animal):
    def eat(self):
        print("Cat just like eat finish")
class Dog(Animal):
    def eat(self):
        print("Dog just like eat bone")
animal = Animal()
animal.eat()
animal = Cat()
animal.eat()
animal = Dog()
animal.eat()
```



多态性示例：通过参数传递实现多态

```
class Animal:
    def eat(self, animal):
        animal.eat()
class Cat(Animal):
    def eat(self):
        print("Cat just like eat finish")
class Dog(Animal):
    def eat(self):
        print("Dog just like eat bone")
cat = Cat()
dog = Dog()
animal = Animal()
animal.eat(cat)
animal.eat(dog)
```



多态性示例：通过参数传递实现多态

```
class Animal:
    def eat(self, animal):
        animal.eat()
class Cat:
    def eat(self):
        print("Cat just like eat finish")
class Dog:
    def eat(self):
        print("Dog just like eat bone")
cat = Cat()
dog = Dog()
animal = Animal()
animal.eat(cat)
animal.eat(dog)
```



多态性示例：通过参数传递实现多态

```
class Animal:
    def eat(self, animal=None):
        if animal:
            animal.eat()
        else:
            print("Animal like eat anything")
class Cat:
    def eat(self):
        print("Cat just like eat finish")
class Dog:
    def eat(self):
        print("Dog just like eat bone")
cat = Cat()
dog = Dog()
animal = Animal()
animal.eat()
animal.eat(cat)
animal.eat(dog)
```



多态性示例：运算符重载

运算符重载指的是对已有的运算符再去定义新的操作功能，或者说对一个已有的运算符赋予新的含义，使之实现新的功能

- Python中规定了可重载的内置运算符
- 运算符重载通过重写类中的特殊方法实现



多态性示例：运算符重载

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __lt__(self, record): #重载self<record运算符
        if self.age < record.age:
            return True
        else:
            return False

    def show_info(self):
        print("Name:", self.name, " Age:", self.age)
```



内容

- 面向对象概念回顾
- 类和实例
- 属性
- 继承和多态
- 综合案例



综合案例：计算图形周长和面积

```
#demo 0705.py  
import math
```

基类Shape

```
class Shape:  
    numInst = 0  
    @staticmethod          #静态方法统计处理图形的总数  
    def count():  
        Shape.numInst += 1  
  
    def __init__(self, name):  
        self.__name = name  
        self.count()  
    def __str__(self):  
        return "图形是"+self.__name+","  
    def area(self):  
        return  
    def perimeter(self):  
        return
```



综合案例：计算图形周长和面积

子类Circle

```
class Circle(Shape):
    def __init__(self,r):
        Shape.__init__(self, "圆形")
        self.r = r

    def perimeter(self):
        return 2 * 3.14 * self.r

    def area(self):
        return 3.14 * self.r ** 2

    def __str__(self):
        return Shape.__str__(self)+"半径为: "+str(self.r)
```



综合案例：计算图形周长和面积

子类Triangle

```
class Triangle(Shape):
    def __init__(self, a, b, c):
        Shape.__init__(self, "三角形")
        self.a, self.b, self.c = a, b, c

    def perimeter(self):
        return self.a + self.b + self.c

    def area(self):
        s = (self.a + self.b + self.c) / 2
        return math.sqrt(s * (s - self.a) * (s - self.b) * (s - self.c))

    def __str__(self):
        return Shape.__str__(self) + "三条边长为: " + str(self.a) + "、" + str(self.b) + "、" + str(self.c)
```



综合案例：计算图形周长和面积

子类Rectangle

```
class Rectangle(Shape):
    def __init__(self, a, b):
        Shape.__init__(self, "矩形")
        self.a, self.b = a, b

    def perimeter(self):
        return 2 * (self.a + self.b)

    def area(self):
        return self.a * self.b

    def __str__(self):
        return Shape.__str__(self)+"长和宽为： "+str(self.a)+"、 "+str(self.b)
```



综合案例：计算图形周长和面积

```
# test demo0705
```

测试代码

```
from demo0705 import *
```

#从源程序文件中导入所有自定义类

```
t1 = Triangle(3, 4, 5)
```

```
c1 = Circle(3)
```

```
r1 = Rectangle(3, 9)
```

```
c2 = Circle(7)
```

```
r2 = Rectangle(7, 6)
```

```
t2 = Triangle(9, 8, 6)
```

```
shapes=[t1, c1, r1, c2, r2, t2]
```

```
print("共处理图形"+str(Shape.numInst)+"个")
```

```
i = 1
```

```
for s in shapes:
```

```
    print("第{}个{}, 面积是{:.2f}, 周长是{:.2f}".format(i, s, s.area(), s.perimeter()))
```

```
    i += 1
```



综合案例：计算图形周长和面积

共处理图形6个

第1个图形是三角形,三条边长为: 3、4、5, 面积是6.00, 周长是12.00

第2个图形是圆形,半径为: 3, 面积是28.26, 周长是18.84

第3个图形是矩形,长和宽为: 3、9, 面积是27.00, 周长是24.00

第4个图形是圆形,半径为: 7, 面积是153.86, 周长是43.96

第5个图形是矩形,长和宽为: 7、6, 面积是42.00, 周长是26.00

第6个图形是三角形,三条边长为: 9、8、6, 面积是23.53, 周长是23.00



面向对象程序设计：总结

- Python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
- 在Python中一切皆对象
- Python中可以动态地为类和对象增加或删除成员
- Python中并不存在严格意义上的“私有成员”
- Python中的多态性是“鸭子类型”的体现，不关注对象的具体类型，只要对象存在相应的实例方法，就可以调用执行