

第六章 面向对象编程





主要内容

- 面向对象编程的基本概念
- 类和对象
- 封装
- 继承
- 多态



面向对象编程范式





编程范式

编写程序的一种基本风格或模式。不同的编程范式就是以不同的思考方式，指导我们组织和构建代码。

- 面向过程：程序被看作是一系列顺序执行的指令，就像一个食谱，一步一步地告诉计算机做什么。（流程图）
- 面向对象：程序被组织成一组相互作用的‘对象’，每个对象都有自己的数据和行为。（关联图）





Object-Oriented Programming (OOP)

程序代码通过“对象”来模拟现实世界或抽象概念的实体和事物。

■ 特点：以“对象”为核心组织代码

每个对象包含数据（属性）和操作数据的方法（行为），二者被封装为一个独立的逻辑单元。

■ 优点：

更好的代码组织、代码重用

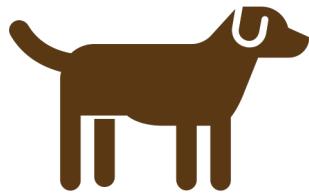
提高可维护性、代码的灵活性





现实世界的对象

- 通常指我们能够感知到的具体的、实在的物体或事物。
- 这些对象拥有各种属性和行为。





Python中的对象

■ 面向对象编程的模拟特性：

程序代码通过“对象”来模拟现实世界或抽象概念的实体和事物。

■ 一切皆为对象：

语言本身将所有组成部分（基本数据、数据结构、函数、类、异常等）也都被视为对象。

用户自定义各种类的对象



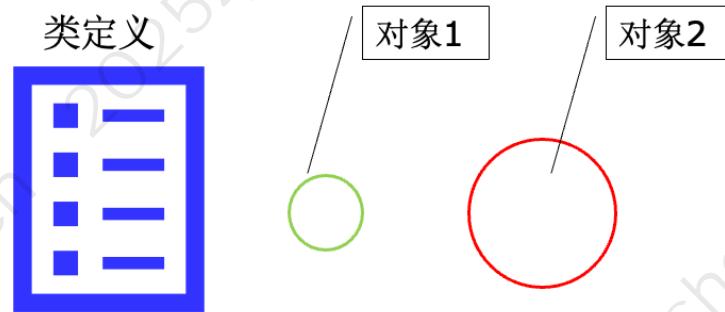
定义类和创建对象





对象和类

- **类:**用来描述具有相同特征和行为的对象的模板或蓝图。
- **对象:**根据类创建出来的具体实例，它拥有该类定义的对象属性和对象方法。



- **属性:** 描写对象静态特性的数据元素。
- **方法:** 用于描写对象动态特性（行为特性）的一组操作。
- **实例化:** 是指在类定义的基础上构造具体对象的过程。





类的定义

Python 使用 **class** 关键字来定义类。

示例：创建一个圆类，包括半径属性以及计算面积和周长两个方法

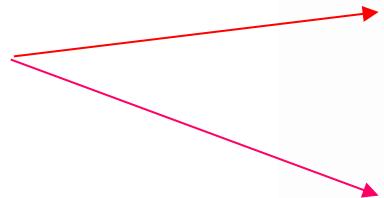
class Classname :  **class Circle:**

initializer



```
def __init__(self, radius):  
    self.radius = radius
```

methods



```
def calculate_area(self):  
    return math.pi * self.radius**2
```

```
def calculate_perimeter(self):  
    return 2 * math.pi * self.radius
```





构造函数

- 构造函数（方法）`__init__`：当创建类的对象即实例化时，它被自动调用，用于初始化对象的属性。
- 它的第一个参数必须是self，用于引用当前创建的对象。红圈是对
象的属性，也可以称为成员变量。

```
def __init__(self, radius):  
    self.radius = radius
```





对象方法

- 对象方法是对象（实例）可以执行的操作或行为，本质上是定义在类中的一种函数。
- 定义对象方法的第一个参数必须是 `self`，指示调用该方法的对象。
- 调用对象方法时，无需显式传递该参数。

```
def calculate_area(self):  
    return math.pi * self.radius**2
```

```
def calculate_perimeter(self):  
    return 2 * math.pi * self.radius
```

- 访问或修改对象属性需要以 `self` 为前缀。





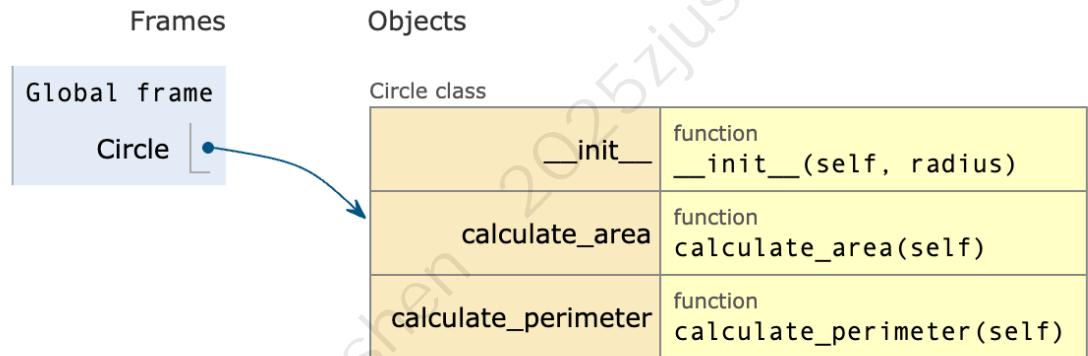
可视化类定义 (pythontutor)

class Circle:

```
def __init__(self, radius):  
    self.radius = radius
```

```
def calculate_area(self):  
    return math.pi * self.radius**2
```

```
def calculate_perimeter(self):  
    return 2 * math.pi * self.radius
```



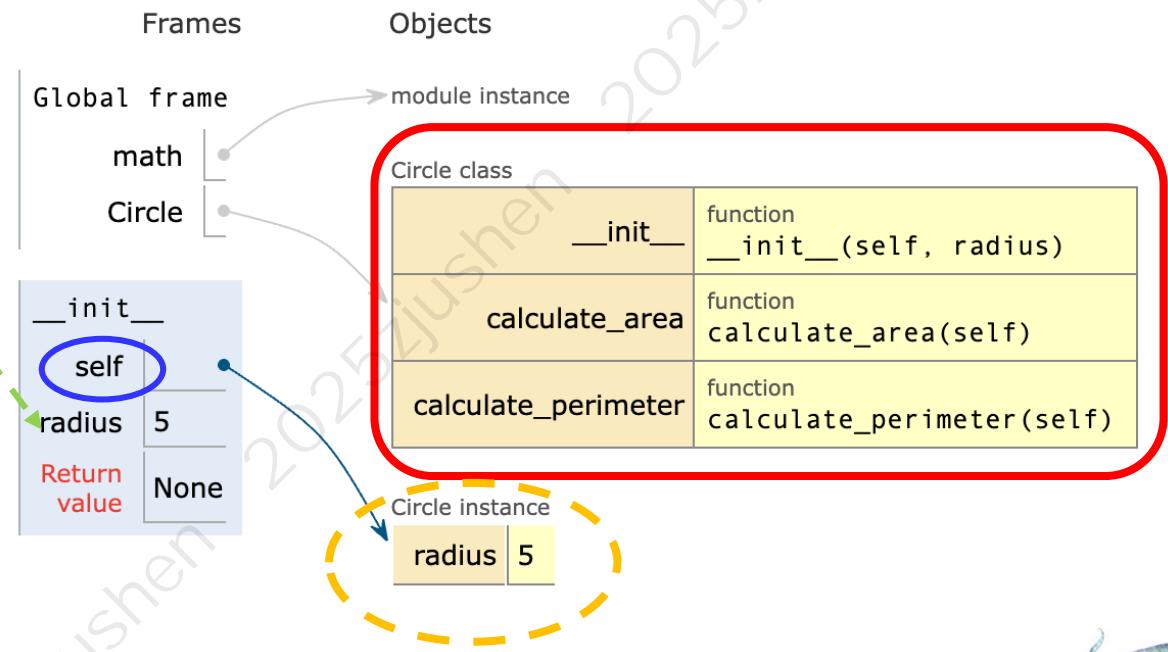


实例化：创建对象

■ 创建一个半径为5的圆

```
circle1 = Circle(5)      # 创建半径为5的圆（实例化）
```

```
isinstance(circle1,Circle) : True
```





访问对象属性

■ 访问属性：

对象.属性 circle1.radius

■ 给属性指定默认值

```
class Circle:
```

```
    def __init__(self, radius = 1):  
        self.radius = radius
```

circle2 = Circle() #创建了半径为1的圆

■ 修改属性的值

circle2.radius = 10 #修改circle2的半径为10





调用对象方法

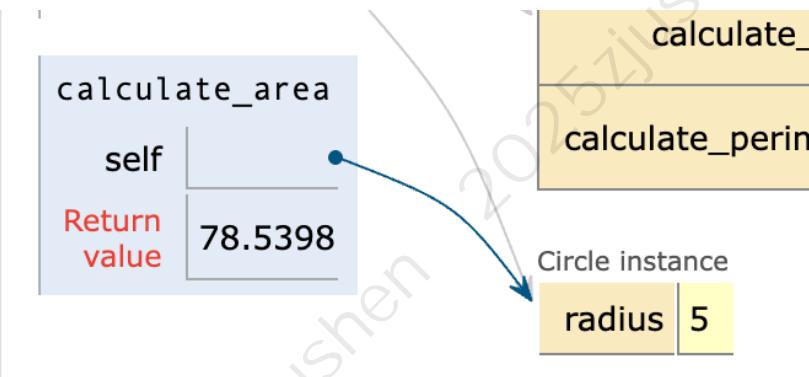
■ 调用方法：

对象.方法() circle1.calculate_area()

不需要显式传递self参数

```
def calculate_area(self):  
    return math.pi * self.radius**2
```

```
print(f'面积为:{circle1.calculate_area():.2f}') #计算圆的面积  
print(f'周长为:{circle1.calculate_perimeter():.2f}') #计算圆的周长
```





练习：创建学生类（Student）

- 该类包括四个成员变量（实例变量）

Course_Grade
GPA
name
number

- 一个成员方法

getInfo

- 实例化s1与s2两名学生（2个实例）

Student instance	
Course_Grade	
GPA	0
name	"wang"
number	"317000010"

Student instance	
Course_Grade	
GPA	0
name	"zhang"
number	"317000011"



参考代码：创建学生类

```
class Student:      #学生类: 包含成员变量和成员方法
    def __init__(self,mname,mnumber):  #构造方法
        self.name = mname    #成员变量
        self.number = mnumber
        self.Course_Grade = {}
        self.GPA = 0
    def getInfo(self):          #成员方法
        print(self.name,self.number)

s1 = Student("wang","317000010")  #创建s1对象
s1.getInfo()
s2=Student("zhang","317000011")  #创建s2对象
s2.getInfo()
```



进阶操作





类属性

类属性（变量）用于存储类级别的数据，所有实例共享。

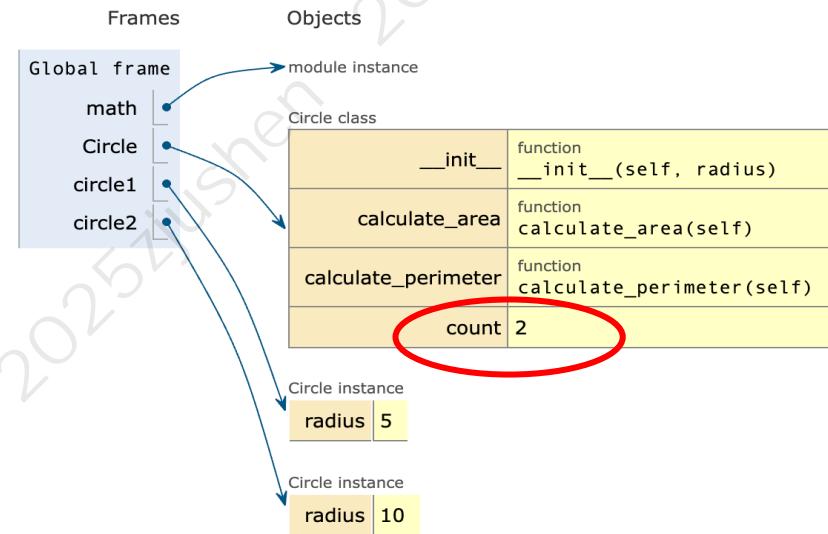
- 定义位置：直接定义在类内部，但不在任何方法中。
- 访问方式：通过类名或对象名都可以访问。
- 适用场景：全局配置、计数器、常量管理等需要共享数据的场景。

class Circle:

```
count = 0 #圆的数量
```

```
def __init__(self, radius = 1):  
    self.radius = radius  
    Circle.count += 1
```

```
circle1 = Circle(5) #创建半径为5的圆  
circle2 = Circle(10) #创建半径为10的圆
```



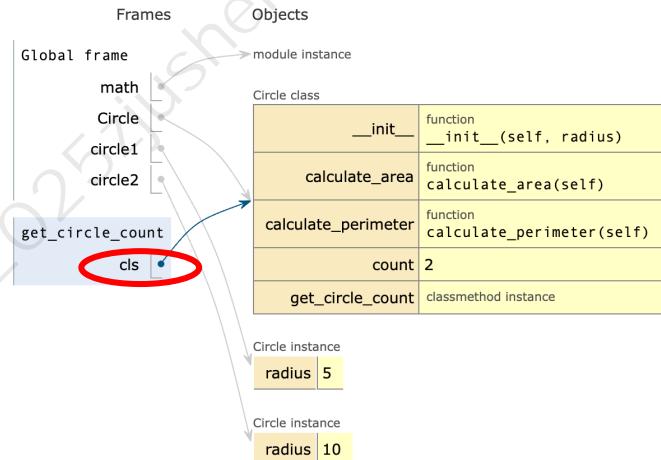


类方法(*)

类方法与类本身相关联，而不与类的特定实例（对象）关联。

- 定义：用`@classmethod`装饰器定义
- 第一个参数：强制使用`cls`，表示类本身，而不是实例。
- 调用方式：通常通过类名调用。
- 适用场景：定义操作类属性的方法，或者在子类中重写父类的方法实现多态等。

```
@classmethod  
def get_circle_count(cls):  
    return cls.count  
  
print(Circle.get_circle_count())
```





静态方法(*)

静态方法在类中定义，但不与类或实例（对象）关联。

- 定义：用`@staticmethod`装饰器定义
- 参数：不接收`self`（实例本身）或`cls`（类本身）作为第一个参数。因此，它们无法直接访问实例属性或类属性。
- 调用方式：通常通过类名调用。
- 适用场景：作为工具函数出现。

```
@staticmethod  
def is_valid_radius(radius):  
    if radius > 0:  
        return True  
    else:  
        return False
```

```
r=int(input())  
if Circle.is_valid_radius(r):  
    circle1 = Circle(r)  
else:  
    print("半径需大于0")
```





特性一：封装与名字空间

特点/方面	类名字空间	实例名字空间
定义	存储类级别变量和方法的字典	存储特定实例变量的字典
作用范围	类级别，所有实例共享	实例级别，每个实例独有
创建时机	类定义时创建	类实例化时创建
访问方式	可通过类名或实例访问类变量和方法	只能通过实例访问实例变量
存储内容	类变量、类方法、静态方法等	实例变量（不包括类变量）
生命周期	与类的生命周期一致	与实例的生命周期一致
修改影响	修改类变量影响所有实例	修改实例变量只影响该实例
<code>__dict__</code> 属性	类有 <code>__dict__</code> ，包含类名字空间的内容	实例有 <code>__dict__</code> ，包含实例名字空间的内容
名字冲突	类名字空间中的名字不会被实例名字空间覆盖	实例名字空间中的名字优先于类名字空间中的同名项





Python是面向对象语言

Python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、多态等重要特性。

■ 封装：

将数据和对数据的操作组合起来构成类，类是一个不可分割的独立单位。类中既要提供与外部联系的接口，同时又要尽可能隐藏类的实现细节。有利于移植。

■ 继承：

从一个通用类（父类），扩展更多特定的类（子类）实现代码重用

■ 多态：

可以对不同类型的对象进行相同的操作，并可以正常运行，提高程序灵活性。





示例：汽车类

```
class Car:  
    model_count = 0 # 类变量，用于跟踪创建的Car实例数量  
  
    def __init__(self, name, color='黑色', price=0):  
        # 初始化实例变量  
        self.name = name  
        self.color = color  
        self.price = price  
        # 每创建一个新实例，就将model_count加1  
        Car.model_count += 1  
  
    # 定义一个方法，用于设置汽车的颜色  
    def set_color(self, color):  
        self.color = color  
  
    # 定义一个方法，用于设置汽车的价格  
    def set_price(self, price):  
        self.price = price
```





```
# 定义一个类方法，用于打印创建的Car实例数量
@classmethod
def Carmodels(cls):
    print(f'有{cls.model_count}款车')

# 定义一个静态方法，用于根据车价格计算分12期每月的还款额（假设无利息）
@staticmethod
def calculate_monthly_payment(price):
    # 假设没有利息，简单地将价格除以12得到每月还款额
    return price / 12
```





```
# 创建两个Car类的实例
car1 = Car("比亚迪")
car2 = Car("小米")

# 打印创建的Car实例数量
print("创建的Car实例数量:", Car.model_count) # 输出应该是2
print("car1的名字:", car1.name)

# 使用set_color方法为car1设置颜色，并使用set_price方法设置价格，同时修改car1的名字
car1.set_color('红色')
car1.set_price(330000)
car1.name = "新比亚迪"

# 打印car1的名字、颜色和价格
print("car1的名字、颜色和价格:", car1.name, car1.price, car1.color)

# 打印car2的名字、颜色和价格（注意car2的颜色和价格还未设置，所以使用默认值）
print("car2的名字、颜色和价格:", car2.name, car2.price, car2.color)

# 调用类方法打印Car实例数量
Car.Carmodels()

# 调用静态方法计算并打印car1和car2分12期每月的还款额
print("car1分12期每月的还款额:", Car.calculate_monthly_payment(car1.price))
```





Car类和car1对象的名字空间

```
>>> dir(Car)
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'price']
```

```
>>> dir(car1)
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', 'color', 'price']
```





补充：Pass语句

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”。

```
>>> class A:  
        pass  
>>> def demo():  
        pass  
>>> if 5>3:  
        pass
```

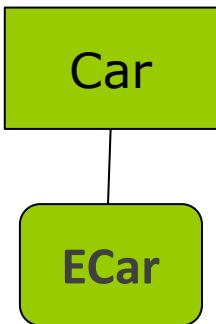




特性二：继承

从一个通用类（父类），扩展更多特定的类（子类）实现代码重用。

super() 函数是一个非常重要的内置函数，它主要用于调用父类（或超类）的方法。
当子类需要扩展或修改从父类继承来的方法时，经常会用到它。



```
class Car():
    price = 300000
    def __init__(self, name):
        self.name = name
        self.color = ""

    def setColor(self, color):
        self.color = color

class ECar(Car):
    def __init__(self, name):
        super().__init__(name)
        self.battery_size = 300

    def getEcar(self):
        print("我是电动汽车" + self.name + "电瓶容量为" + str(self.battery_size) + "公里")

car = ECar("曹操专车")
car.getEcar()
```





示例： Mylist新类

定义一个Mylist新类，它来自list的继承，并增加随机选取功能。

```
import random
class Mylist(list): #list是父类名
    def choice(self):
        return random.choice(self)

lst=Mylist("good morning")
print(Mylist.__bases__)
print(lst.choice())
```





对象继承类的方法

```
>>> dir(list)
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__',  
'__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',  
'_ subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',  
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__',  
'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

```
>>> dir(Mylist)
```

```
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dict__', '__dir__', '__doc__',  
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',  
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__module__', '__mul__', '__ne__', '__new__', '__reduce__',  
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subc  
lasshook__', '__weakref__', 'append', 'choice', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'rev  
erse', 'sort']
```





特性三：多态

python的多态性是可以表现在相同的运算符（函数）具有不同功能

```
>>> 12 + 6  
18  
>>> '12' + '6'  
'126'  
>>>
```

```
class Animal:  
    def speak(self):  
        return  
  
class Dog(Animal):  
    def speak(self):  
        return "Woof!"  
  
class Cat(Animal):  
    def speak(self):  
        return "Meow!"  
  
# 多态性体现  
def animal_speak(animal):  
    print(animal.speak())  
  
Tom = Dog()  
Alice = Cat()  
  
animal_speak(Tom) # 输出: Woof!  
animal_speak(Alice) # 输出: Meow!
```





案例一：计算身体质量指数BMI

- 定义类：创建一个名为BMI的类，用于封装与BMI计算相关的属性和方法。
 - 构造函数：定义`__init__`方法，它是类的构造函数，用于初始化对象时设置其属性。在这个构造函数中，我们接收姓名、年龄、体重和身高作为参数，并将它们分别赋值给对象的属性。
 - 计算BMI值：定义`calculate_bmi`方法，它不使用任何外部参数，而是使用对象的`weight`和`height`属性来计算BMI值。
 - 获取BMI类别：定义`get_bmi_category`方法，它调用`calculate_bmi`方法来获取BMI值，并根据BMI值返回对应的健康状态类别。
 - 字符串表示：定义`__str__`方法，它返回一个格式化的字符串，包含BMI值和对象的健康状态。这个方法使得我们可以直接打印BMI对象来获取易读的信息。





代码

```
# 定义一个名为BMI的类
class BMI:
    # 类的构造函数, 用于初始化BMI对象的属性
    def __init__(self, name, age, weight, height):
        self.name = name          # 存储用户的姓名
        self.age = age            # 存储用户的年龄
        self.weight = weight      # 存储用户的体重 (千克)
        self.height = height      # 存储用户的身高 (米)

    # 计算BMI值的方法
    def calculate_bmi(self):
        # 使用体重除以身高的平方来计算BMI值
        return self.weight / (self.height ** 2)

    # 根据BMI值返回对应的健康状态类别
    def get_bmi_category(self):
        # 首先计算BMI值
        bmi = self.calculate_bmi()
        # 根据BMI值判断并返回对应的健康状态
        if bmi < 18.5:
            return "偏瘦"
        elif 18.5 <= bmi < 24:
            return "正常"
        elif 24 <= bmi < 28:
            return "超重"
        else:
            return "肥胖"

    # 返回BMI对象的字符串表示形式
    def __str__(self):
        # 格式化字符串, 包含BMI值和对应的健康状态
        return (f"BMI值为: {self.calculate_bmi():.2f}\n"
                f"根据中国16岁以上人群的BMI标准, {self.name}属于: {self.get_bmi_category()}")
```





```
if __name__ == "__main__":
    name = input("请输入您的姓名: ")
    age = int(input("请输入您的年龄: "))
    weight = float(input("请输入您的体重(千克): "))
    height = float(input("请输入您的身高(米): "))
    bmi1 = BMI(name, age, weight, height)
    print(bmi1)
```

运行结果:

```
请输入您的姓名: Liming
请输入您的年龄: 20
请输入您的体重(千克): 70
请输入您的身高(米): 1.75
BMI值为: 22.86
根据中国16岁以上人群的BMI标准, Liming属于: 正常
```





课后练习一

设计一个计数器类



属性：读数

方法：置零

按一下

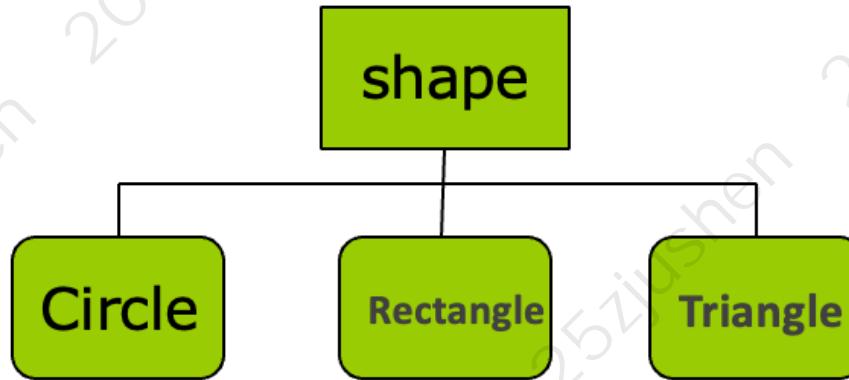
读数





课后练习二

设计一个Shape父类，扩展如圆类、矩形类、三角形等子类，实现求面积、周长等功能。



补充内容（不做要求）





编码风格

你必须熟悉有些与类相关的编码风格问题，在你编写的程序较复杂时尤其如此。

类名应采用驼峰命名法，即将类名中的每个单词的首字母都大写，而不使用下划线。实例名和模块名都采用小写格式，并在单词之间加上下划线。

对于每个类，都应紧跟在类定义后面包含一个文档字符串。这种文档字符串简要地描述类的功能，并遵循编写函数的文档字符串时采用的格式约定。每个模块也都应包含一个文档字符串，对其中的类可用于做什么进行描述。

可使用空行来组织代码，但不要滥用。在类中，可使用一个空行来分隔方法；而在模块中，可使用两个空行来分隔类。

摘自《Python 从编程到实践》





Python约定

■ **命名约定:** 在Python中，通常通过命名约定来表明变量或方法的访问级别：

- **无下划线:** 通常用于公共变量或方法，意味着它们可以被类的外部自由访问。
- **下划线前缀（`_xxx`）:** 通常用于表示“受保护的”或“内部的”变量或方法，意味着它们不应该被类的外部直接访问，但仍然是可访问的。这只是一种约定，并不真正阻止外部访问。
- **双下划线前缀（`__xxx`）:** 用于实现名称改写（name mangling），这是一种避免子类意外覆盖父类方法的机制。当类的外部尝试访问这样的变量时，它会被改写为`_ClassName __xxx`，从而在一定程度上避免了直接访问。
- **前后双划线（`__xxx__`）:** 系统定义的特殊成员；不要创建这种标识符

■ **属性访问方法:** 除了命名约定外，Python还提供了属性访问方法（如`@property`装饰器）来进一步控制对类属性的访问。通过定义`getter`、`setter`和`deleter`方法，可以控制对属性的读取、修改和删除操作，从而实现封装。





思考题：输出结果是什么？

```
class A:  
    def __init__(self):  
        self.x = 1  
        self._y = 1  
        self.__z = 1  
  
    def getX(self):  
        return self.x  
  
    def getY(self):  
        return self._y  
  
    def getZ(self):  
        return self.__z
```

```
a = A()  
a.x = 30  
a._y = 45  
a.__z = 60  
print(a.getX(),a.getY(),a.getZ())  
a._A_z = 75  
print(a.getZ())
```

30	45	1
75		





案例：设计向量类

■ 定义类：创建一个Vector3D类，用于表示一个三维向量。

- 初始化方法：`__init__`方法用于初始化向量的x, y, z坐标，默认值为0。

- 运算符重载：

- ▶ `__add__`：实现向量加法，返回新的Vector3D对象。
- ▶ `__sub__`：实现向量减法，返回新的Vector3D对象。
- ▶ `__mul__`：实现向量与标量的乘法，返回新的Vector3D对象。
- ▶ `__truediv__`：实现向量与标量的除法，返回新的Vector3D对象。

- 类型检查：在每个运算符重载方法中，都进行了类型检查，以确保操作数的类型正确。

- 异常处理：如果操作数的类型不正确，将抛出`TypeError`异常。

- 字符串表示：`__repr__`方法定义了向量的字符串表示形式，便于打印和调试





代码

```
class Vector3D:  
    def __init__(self, x=0, y=0, z=0):  
        self.x = x  
        self.y = y  
        self.z = z  
  
    def __add__(self, other):  
        if isinstance(other, Vector3D):  
            return Vector3D(self.x + other.x, self.y + other.y, self.z + other.z)  
        else:  
            raise TypeError("Operand must be of type Vector3D")  
  
    def __sub__(self, other):  
        if isinstance(other, Vector3D):  
            return Vector3D(self.x - other.x, self.y - other.y, self.z - other.z)  
        else:  
            raise TypeError("Operand must be of type Vector3D")  
  
    def __mul__(self, scalar):  
        if isinstance(scalar, (int, float)):  
            return Vector3D(self.x * scalar, self.y * scalar, self.z * scalar)  
        else:  
            raise TypeError("Operand must be a scalar (int or float)")  
  
    def __truediv__(self, scalar):  
        if isinstance(scalar, (int, float)) and scalar != 0:  
            return Vector3D(self.x / scalar, self.y / scalar, self.z / scalar)  
        else:  
            raise TypeError("Operand must be a non-zero scalar (int or float)")  
  
    def __repr__(self):  
        return f"Vector3D(x={self.x}, y={self.y}, z={self.z})"
```





```
if __name__ == "__main__":
    v1 = Vector3D(1, 2, 3)
    v2 = Vector3D(4, 5, 6)
    scalar = 2

    # 向量加法
    v_add = v1 + v2
    print(f"v1 + v2 = {v_add}")

    # 向量减法
    v_sub = v1 - v2
    print(f"v1 - v2 = {v_sub}")

    # 向量与标量的乘法
    v_mul = v1 * scalar
    print(f"v1 * {scalar} = {v_mul}")

    # 向量与标量的除法
    v_div = v1 / scalar
    print(f"v1 / {scalar} = {v_div}")
```

运行结果:

```
v1 + v2 = Vector3D(x=5, y=7, z=9)
v1 - v2 = Vector3D(x=-3, y=-3, z=-3)
v1 * 2 = Vector3D(x=2, y=4, z=6)
v1 / 2 = Vector3D(x=0.5, y=1.0, z=1.5)
```

