潍坊科技学院
Weifang University of Science and Technology

第8章 面向对象

授课老师：刘国旭

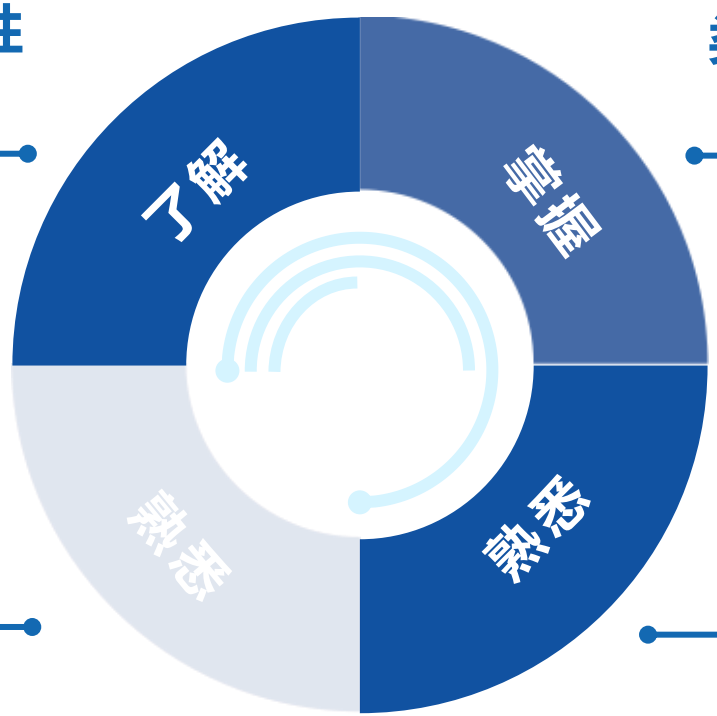
潍坊科技学院

- 面向对象概述
- 类与对象
- 类的成员
- 构造方法和析构方法
- 封装、继承、多态
- 运算符重载

学习目标

1 **了解** 面向对象编程思想及其三大特性

2 **掌握** 类和对象的关系、类的属性和方法



3 **熟悉** 设计和使用类

4 **熟悉** 运算符的重载

目录页

目录页

目录页

8.1 面向对象概述

面向对象是程序开发领域的重要思想，这种思想模拟了人类认识客观世界的思维方式，将开发中遇到的事物皆看作对象。

8.1 面向对象概述

面向过程

- 分析解决问题的步骤
- 使用函数实现每个步骤的功能
- 按步骤依次调用函数

面向对象

- 分析问题，从中提炼出多个对象
- 将不同对象各自的特征和行为进行封装
- 通过控制对象的行为来解决问题。

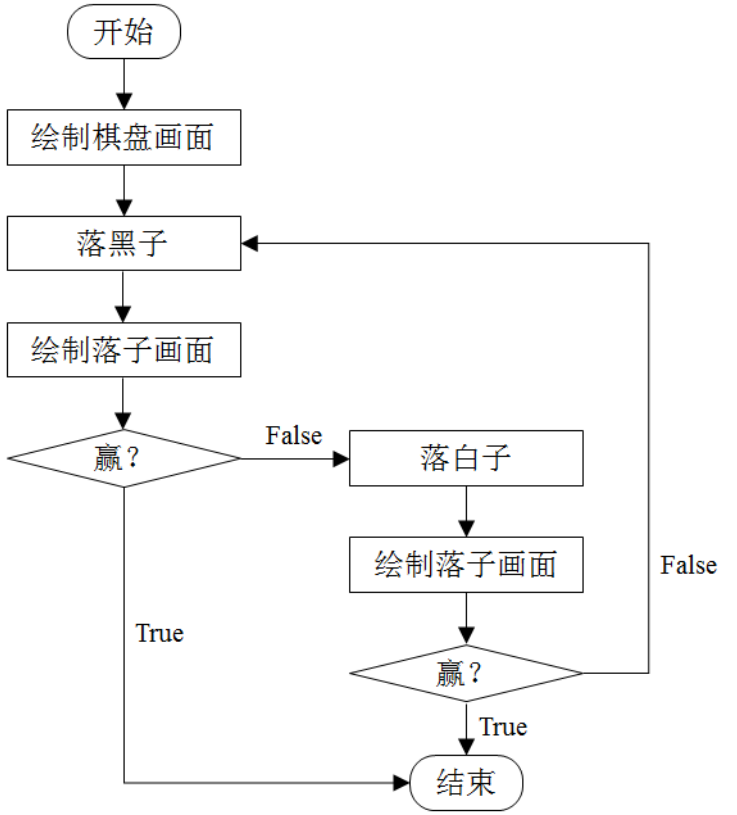
8.1 面向对象概述

分别使用面向过程和面向对象来实现五子棋

编程思想	实现步骤	特点
面向过程	（1）开始游戏。 （2）绘制棋盘画面。 （3）落黑子。 （4）绘制棋盘落子画面。 （5）判断输赢。 （6）落白子。 （7）绘制棋盘落子画面。 （8）判断输赢：赢则结束游戏，否则返回步骤（2）。	每个步骤的操作都可以封装为一个函数，按以上步骤逐个调用函数，即可实现一个五子棋游戏。
面向对象	（1）玩家：黑白双方，负责决定落子的位置。 （2）棋盘：负责绘制当前游戏的画面，向玩家反馈棋盘的状况。 （3）规则系统：负责判断游戏的输赢。	把解决问题的事物分为多个对象，对象具备解决问题过程中的行为。

五子棋游戏——对象特征与行为

面向过程	(1) 开始游戏。 (2) 绘制棋盘画面。 (3) 落黑子。 (4) 绘制棋盘落子画面。 (5) 判断输赢。 (6) 落白子。 (7) 绘制棋盘落子画面。 (8) 判断输赢：赢则结束游戏，否则返回步骤(2)。



五子棋游戏——对象特征与行为

面向对象	（1）玩家：黑白双方，负责决定落子的位置。 （2）棋盘：负责绘制当前游戏的画面，向玩家反馈棋盘的状况。 （3）规则系统：负责判断游戏的输赢。

	玩家	棋盘	规则系统
特征	棋子（黑或白子）	棋盘数据	无
行为	落子	显示棋盘 更新棋盘	判定胜负

8.1 面向对象概述

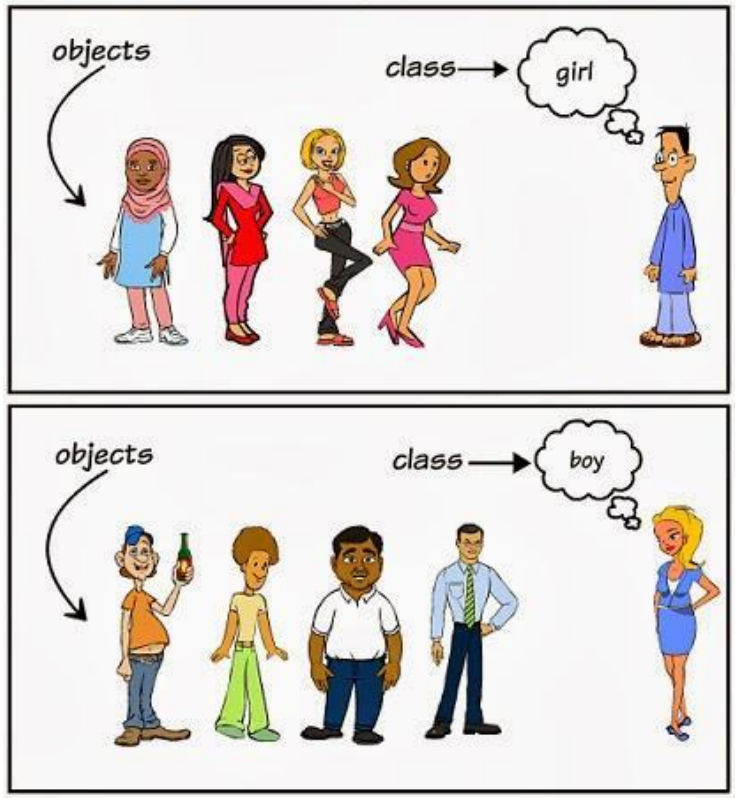
若加入悔棋功能，面向过程和面向对象，分别怎么实现呢？

8.1 面向对象概述

面向过程	面向对象
从输入、判断到显示的一系列步骤都需要改动	只需要改动棋盘对象就可以 更简便!

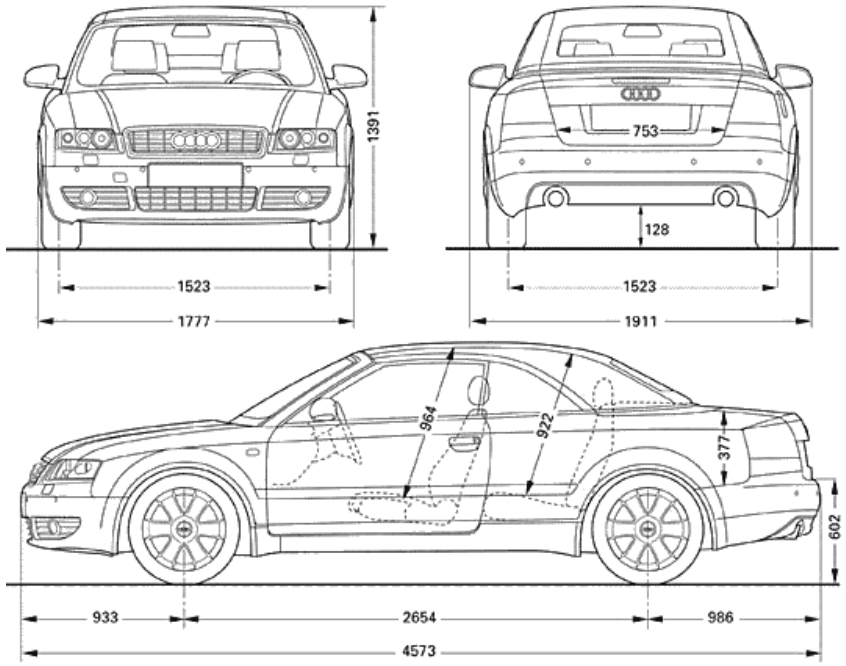
目录页

8.2 类的定义与使用



- 面向对象编程有两个非常重要的概念：类和对象。
- 对象映射现实中真实存在的事物，如一本书。
- 具有相同特征和行为的事物的集合统称为类。
- 对象是根据类创建的，一个类可以对应多个对象。
- 类是对象的抽象，对象是类的实例。

8.2.1 类的定义



类是由3部分组成的：

- 类的名称：大驼峰命名法，首字母一般大写，比如Person。
- 类的属性：用于描述事物的特征，比如性别。
- 类的方法：用于描述事物的行为，比如抬腿。

8.2.1 类的定义

```
class 类名:
    属性名 = 属性值
    def 方法名(self):
        方法体
```

8.2.1 类的定义

语法格式

```
class 类名:
    属性名 = 属性值
    def 方法名(self):
        方法体
```

示例

```
class Car:
    wheels = 4          # 属性
    def drive(self):    # 方法
        print('行驶')
```

8.2.2 对象的创建与使用

根据类创建对象的语法格式如下：

对象名 = 类名()	car = Car()

使用对象的本质是访问对象成员：

对象名.属性名	car.wheels
对象名.方法名()	car.drive()

目录页

8.3.1 属性

属性按声明的方式可以分为两类：类属性和实例属性。

8.3.1 属性

1. 类属性

- 声明在类内部、方法外部的属性。
- 可以通过类或对象进行访问，但只能通过类进行修改。

示例

```
car = Car()                                  # 创建对象car
print(Car.wheels)        4                   # 通过类Car访问类属性
print(car.wheels)        4                   # 通过对象car访问类属性
Car.wheels = 3                               # 通过类Car修改类属性wheels
print(Car.wheels)        3
print(car.wheels)        3
car.wheels = 4                               # 通过对象car修改类属性wheels
print(Car.wheels)        3
print(car.wheels)        4
```

car对象不能修改类属性的值

为什么对象访问的属性值为4?

2. 实例属性

- 实例属性是在方法内部声明的属性。
- Python支持动态添加实例属性。

（1）访问实例属性——只能通过对象访问

示例

```
class Car:
    def drive(self):
        self.wheels = 4                    # 添加实例属性
car = Car()                                # 创建对象car
car.drive()
print(car.wheels)                          # 通过对象car访问实例属性
print(Car.wheels)                          # 通过类Car访问实例属性，错误！！！
```

2. 实例属性

- 实例属性是在方法内部声明的属性。
- Python支持动态添加实例属性。

（2）修改实例属性——通过对象修改

示例

```
class Car:
    def drive(self):
        self.wheels = 4                    # 添加实例属性
car = Car()                                # 创建对象car
car.drive()
car.wheels = 3                             # 修改实例属性
print(car.wheels)                          # 通过对象car访问实例属性
```

8.3.1 属性

2. 实例属性

- 实例属性是在方法内部声明的属性。
- Python支持动态添加实例属性。

（3）动态添加实例属性——类外部使用对象动态添加实例属性

示例

```
class Car:
    def drive(self):
        self.wheels = 4              # 添加实例属性
car = Car()                          # 创建对象car
car.drive()
car.wheels = 3                       # 修改实例属性
print(car.wheels)                    # 通过对象car访问实例属性
car.color = "红色"                   # 动态地添加实例属性
print(car.color)
```

结果

```
3
红色
```

8.3.2 方法

Python中的方法按定义方式和用途可以分为三类：实例方法、类方法和静态方法。

1. 实例方法

- 形似函数，但它定义在类内部。
- 以self为第一个形参，self参数代表对象本身
- 只能通过对象调用

示例

```
class Car:
    def drive(self):                    # 实例方法
        print("我是实例方法")
car = Car()
car.drive()                             # 通过对象调用实例方法
Car.drive()                             # 通过类调用实例方法
```

8.3.2 方法

Python中的方法按定义方式和用途可以分为三类：实例方法、类方法和静态方法。

2. 类方法

- 类方法是定义在类内部
- 使用装饰器@classmethod修饰的方法
- 第一个参数为cls，代表类本身
- 可以通过类和对象调用

示例

```
class Car:
    @classmethod
    def stop(cls):                    # 类方法
        print("我是类方法")
car = Car()
car.stop()                            # 通过对象调用类方法
Car.stop()                            # 通过类调用类方法
```

8.3.2 方法

Python中的方法按定义方式和用途可以分为三类：实例方法、类方法和静态方法。

2. 类方法

- 类方法中可以使用cls访问和修改类属性的值

示例

```
class Car:
    wheels = 3                                  # 类属性
    @classmethod
    def stop(cls):                              # 类方法
        print(cls.wheels)           3           # 使用cls访问类属性
        cls.wheels = 4                          # 使用cls修改类属性
        print(cls.wheels)           4
car = Car()
car.stop()
```

8.3.2 方法

Python中的方法按定义方式和用途可以分为三类：实例方法、类方法和静态方法。

3. 静态方法

- 静态方法是定义在类内部
- 使用装饰器@staticmethod修饰的方法
- 没有任何默认参数

示例

```
class Car:
    @staticmethod
    def test():                          # 静态方法
        print("我是静态方法")
```

8.3.2 方法

Python中的方法按定义方式和用途可以分为三类：实例方法、类方法和静态方法。

3. 静态方法

- 静态方法可以通过类和对象调用

示例

```
class Car:
    @staticmethod
    def test():                          # 静态方法
        print("我是静态方法")
car = Car()
car.test()                               # 通过对象调用静态方法
Car.test()                               # 通过类调用静态方法
```

8.3.2 方法

Python中的方法按定义方式和用途可以分为三类：实例方法、类方法和静态方法。

3. 静态方法

- 静态方法内部不能直接访问属性或方法，但可以使用类名访问类属性或调用类方法，

示例

```
class Car:
    wheels = 3      # 类属性
    @staticmethod
    def test():
        print("我是静态方法")
        print(f"类属性的值为{Car.wheels}")  # 静态方法中访问类属性
```

8.3.3 私有成员

类的成员默认是公有成员，可以在类的外部通过类或对象随意地访问，这样显然不够安全。

为了保证类中数据的安全，Python支持将公有成员改为私有成员，在一定程度上限制在类的外部对类成员的访问。

8.3.3 私有成员

Python通过在类成员的名称前面添加双下画线（__）的方式来表示私有成员，语法格式如下：

- __属性名
- __方法名

示例

```
class Car:
    __wheels = 4                          # 私有属性
    def __drive(self):                    # 私有方法
        print("开车")
```

8.3.3 私有成员

私有成员在类的内部可以直接访问，在类的外部不能直接访问，但可以通过**调用类的公有成员方法**的方式进行访问。

示例

```
class Car:
    __wheels = 4                                        # 私有属性
    def __drive(self):                                  # 私有方法
        print("行驶")
    def test(self):
        print(f"轿车有{self.__wheels}个车轮")           # 公有方法中访问私有属性
        self.__drive()                                  # 公有方法中调用私有方法
```

目录页

8.4 特殊方法

除了8.3节介绍的方法之外，类中还包括两个特殊的方法：构造方法和析构方法，这两个方法都是系统内置方法。

8.4.1 构造方法

- 构造方法指的是__init__()方法。
- 创建对象时系统自动调用，从而实现对象的初始化。
- 每个类默认都有一个__init__()方法，可以在类中显式定义__init__()方法。
- __init__()方法可以分为无参构造方法和有参构造方法。

a. 当使用无参构造方法创建对象时，所有对象的属性都有相同的初始值。

b. 当使用有参构造方法创建对象时，对象的属性可以有不同的初始值。

8.4.1 构造方法

示例：无参构造方法

```
class Car:
    def __init__(self):                                    # 无参构造方法
        self.color = "红色"
    def drive(self):
        print(f"车的颜色为：{self.color}")
car_one = Car()                                            # 创建对象并初始化
car_one.drive()
car_two = Car()                                            # 创建对象并初始化
car_two.drive()
```

8.4.1 构造方法

示例：有参构造方法

```
class Car:
    def __init__(self, color):                    # 有参构造方法
        self.color = color                        # 将形参赋值给属性
    def drive(self):
        print(f"车的颜色为：{self.color}")
car_one = Car("红色")                             # 创建对象，并根据实参初始化属性
car_one.drive()
car_two = Car("蓝色")                             # 创建对象，并根据实参初始化属性
car_two.drive()
```

8.4.2 析构方法

- 析构方法（即__del__()方法）是销毁对象时系统自动调用的方法。
- 每个类默认都有一个__del__()方法，可以显式定义析构方法。

析构方法示例

```
class Car:
    def __init__(self):
        self.color = "蓝色"
        print("对象被创建")
    def __del__(self):
        print("对象被销毁")
car = Car()
print(car.color)
del car
print(car.color)
```

结果

```
对象被创建
蓝色
对象被销毁
----------------------------------------------------------
NameError                    Traceback (most recent call last)
...
---> 10 print(car.color)
NameError: name 'car' is not defined
```

多学一招：销毁对象

与文件类似，每个对象都会占用系统的一块内存，使用之后若不及时销毁，会浪费系统资源。那么对象什么时候销毁呢？

多学一招：销毁对象

Python通过**引用计数器**记录所有对象的引用（可以理解为对象所占内存的别名）数量，一旦某个对象的引用计数器的**值为0**，系统就会**销毁**这个**对象**，收回对象所占用的内存空间。

目录页

8.5.1 好友管理系统

本实例要求编写代码，实现一个基于面向对象思想的、具有添加好友、删除好友、备注好友、展示好友、好友分组、退出功能的好友管理系统。

8.5.2 生词本

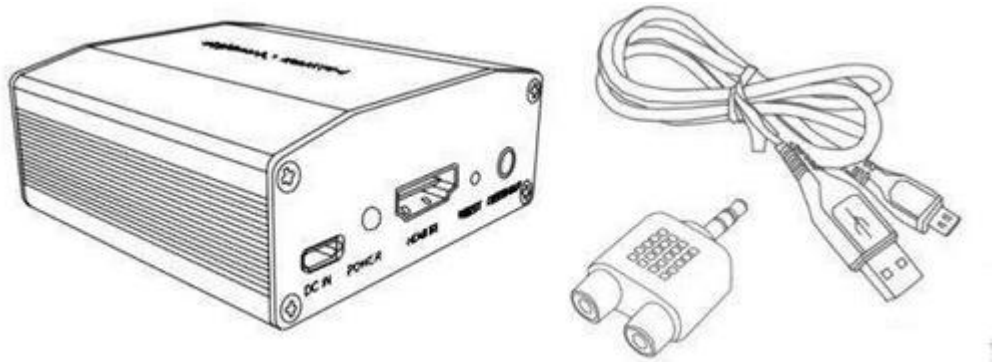
本实例要求编写代码，实现一个基于面向对象思想的、具有背单词、添加新单词、删除单词、查找单词以及清空、退出生词本功能的生词本程序。

目录页

8.6 封装

封装是面向对象的重要特性之一，它的基本思想是对外隐藏类的细节，提供用于访问类成员的公开接口。

如此，类的外部无需知道类的实现细节，只需要使用公开接口便可访问类的内容，这在一定程度上保证了类内数据的安全。



8.6 封装

为了契合封装思想，我们在定义类时需要满足以下两点要求。

1．将类属性声明为私有属性。

2．添加两类供外界调用的公有方法，分别用于设置或获取私有属性的值。

8.6 封装

示例

```
class Person:
    def __init__(self, name):
        self.name = name            # 姓名
        self.__age = 1              # 年龄，默认为1岁，私有属性
    # 设置私有属性值的方法
    def set_age(self, new_age):
        if 0 < new_age <= 120:      # 判断年龄是否合法
            self.__age = new_age
    # 获取私有属性值的方法
    def get_age(self):
        return self.__age
```

调用

```
person = Person("小明")
person.set_age(20)
print(f"年龄为{person.get_age()}岁")
```

目录页

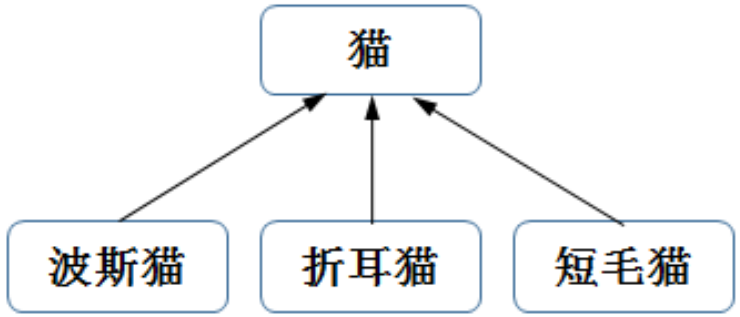
8.7 继承

继承是面向对象的重要特性之一，它主要用于描述类与类之间的关系，在不改变原有类的基础上扩展原有类的功能。

若类与类之间具有继承关系，被继承的类称为父类或基类，继承其他类的类称为子类或派生类，子类会自动拥有父类的公有成员。

8.7.1 单继承

单继承即子类只继承一个父类。现实生活中，波斯猫、折耳猫、短毛猫都属于猫类，它们之间存在的继承关系即为单继承，如图所示。



Python中单继承的语法格式如下所示：

class 子类名(父类名)：

- 子类继承父类的同时会自动拥有父类的公有成员。
- 自定义类默认继承基类object。

8.7.1 单继承

示例

```
class Cat(object):
    def __init__(self, color):
        self.color = color
    def walk(self):
        print("走猫步~")
# 定义继承Cat的ScottishFold类
class ScottishFold(Cat):
    pass
fold = ScottishFold("灰色")          # 创建子类的对象
print(f"{fold.color}的折耳猫")       # 子类访问从父类继承的属性
fold.walk()                          # 子类调用从父类继承的方法
```

8.7 继承

子类不会拥有父类的私有成员，也不能访问父类的私有成员。

8.7.1 单继承

示例

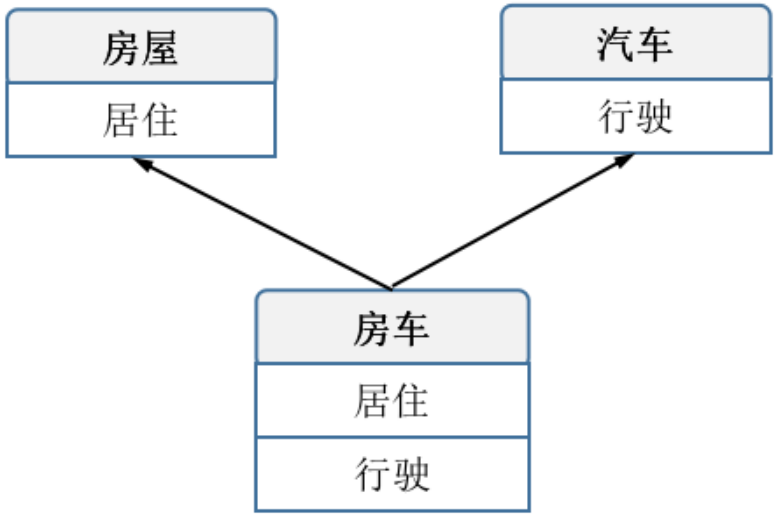
```
class Cat(object):
    def __init__(self, color):
        self.color = color    # 增加私有属性
        self.__age = 1
    def walk(self):
        print("走猫步~")
def __test(self):          # 增加私有方法
        print("父类的私有方法")
print(fold.__age)                    # 子类访问父类的私有属性
fold.__test()                        # 子类调用父类的私有方法
```

AttributeError: 'ScottishFold' object has no attribute '__age'

AttributeError: 'ScottishFold' object has no attribute '__test'

8.7.2 多继承

程序中的一个类也可以继承多个类，如此子类具有多个父类，也自动拥有所有父类的公有成员。



Python中多继承的语法格式如下所示：

class 子类名(父类名1, 父类名2, ...):

8.7.2 多继承

示例

```
# 定义一个表示房屋的类
class House(object):
    def live(self):                                    # 居住
        print("供人居住")
# 定义一个表示汽车的类
class Car(object):
    def drive(self):                                   # 行驶
        print("行驶")
# 定义一个表示房车的类
class TouringCar(House, Car):
    pass
tour_car = TouringCar()
tour_car.live()                                        # 子类对象调用父类House的方法
tour_car.drive()                                       # 子类对象调用父类Car的方法
```

8.7.2 多继承

如果House类和Car类中有一个同名的方法，那么子类会调用哪个父类的同名方法呢?

8.7.2 多继承

如果子类继承的多个父类是平行关系的类，那么子类先继承哪个类，便会先调用哪个类的方法。

8.7.3 重写

子类会原封不动地继承父类的方法，但子类有时需要按照自己的需求对继承来的方法进行调整，也就是在子类中重写从父类继承来的方法。

8.7.3 重写

在子类中定义与父类方法同名的方法，在方法中按照子类需求重新编写功能代码即可。

示例

```
# 定义一个表示人的类
class Person(object):
    def say_hello(self):
        print("打招呼！")
# 定义一个表示中国人的类
class Chinese(Person):
    def say_hello(self):                    # 重写的方法
        print("吃了吗？")
chinese = Chinese()
chinese.say_hello()                         # 子类调用重写的方法
```

8.7.3 重写

子类重写了父类的方法之后，无法直接访问父类的同名方法，但可以使用super()函数间接调用父类中被重写的方法。

示例

```
# 定义一个表示中国人的子类
class Chinese(Person):
    def say_hello(self):
        super().say_hello()              # 调用父类被重写的方法
        print("吃了吗？")
```

目录页

8.8 多态

多态是面向对象的重要特性之一，它的直接表现即让不同类的同一功能可以通过同一个接口调用，表现出不同的行为。

8.8 多态

定义

```
class Cat:
    def shout(self):
        print("喵喵喵~")
class Dog:
    def shout(self):
        print("汪汪汪！")
```

调用

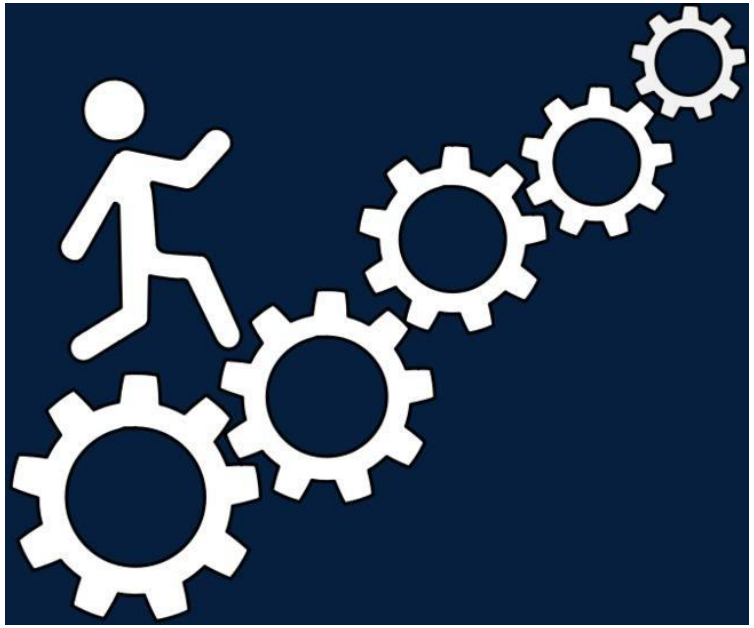
```
def shout(obj):
    obj.shout()
cat = Cat()
dog = Dog()
shout(cat)
shout(dog)
```

结果

```
喵喵喵~
汪汪汪!
```

目录页

8.9 运算符重载



运算符重载是指赋予内置运算符新的功能，使内置运算符能适应更多的数据类型。

8.9 运算符重载

基类object中提供的一些特殊方法及其对应的运算符如表所示。

特殊方法	运算符
__add__()	+
__sub__()	-
__mul__()	*
__truediv__()	/
__mod__()	%
__pow__()	**
__contains__()	in
__eq__()、__ne__()、__lt__()、__le__()、__、gt__()、__ge__()	==、!=、<、<=、>、>=
__and__()、__or__()、__invert__()、__xor__()	&、|、~、^
__iadd__()、__isub__()、__imul__()、__itruediv__()	+=、-=、*=、/=

8.9 运算符重载

如果类中重写了Python基类object内置的有关运算符的特殊方法，那么该特殊方法对应的运算符将支持对该类的实例进行运算。

使用

```
...
calculator = Calculator(10)
print(calculator + 5)
print(calculator - 5)
print(calculator * 5)
print(calculator / 5)
```

示例

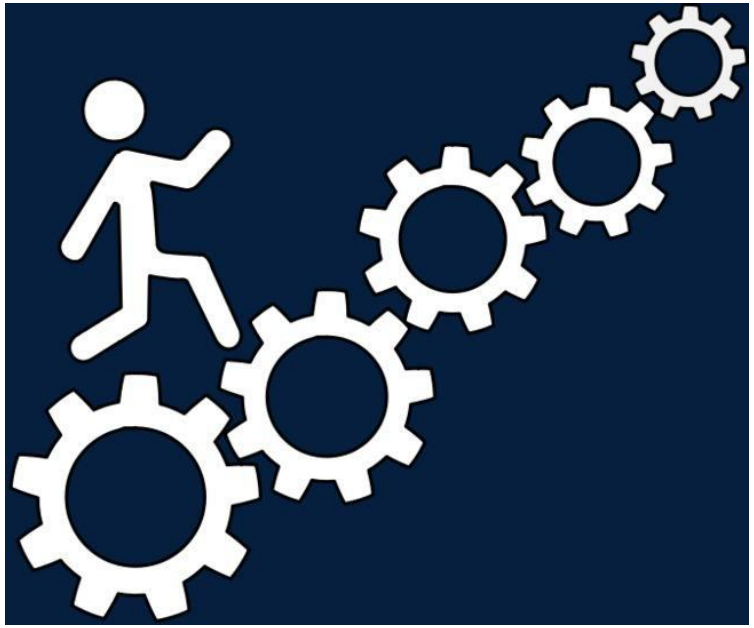
```
class Calculator(object):
    def __init__(self, number):    # 记录数值
        self.number = number
    def __add__(self, other):      # 重载运算符+
        self.number = self.number + other
        return self.number
    def __sub__(self, other):      # 重载运算符-
        self.number = self.number - other
        return self.number
    def __mul__(self, other):      # 重载运算符*
        self.number = self.number * other
        return self.number
    def __truediv__(self, other):  # 重载运算符/
        self.number = self.number / other
        return self.number
```

目录页

8.10.1 人机猜拳游戏

猜拳游戏一般包含三种手势：石头、剪刀、布，判定规则为石头胜剪刀，剪刀胜布，布胜石头。本实例要求编写代码，实现基于面向对象思想的人机猜拳游戏。

8.10.2 自定义列表



为使列表支持四则运算，我们可以自定义一个列表类，在其中重载运算符，列表中各元素分别与数值相加、相减、相乘或相除后所得的结果组成该列表的新元素。本实例要求编写代码，重载运算符，使列表支持四则运算。

目录页

8.11 阶段案例——银行管理系统

本案例要求编写程序，实现一个基于面向思想的、具有开户、查询、取款、存款、转账、锁定、解锁和退出功能的银行管理系统。

8.12 本章小结

本章主要讲解了面向对象的相关知识，包括面向对象概述、类的定义和使用、类的成员、特殊方法、封装、继承、多态、运算符重载，并结合众多实训案例演示了面向对象的编程技巧。通过本章的学习，希望读者能理解面向对象的思想与特性，掌握面向对象的编程技巧，为以后的开发奠定扎实的面向对象编程基础。