模拟学生和老师的一天

学生	老师
人物出场介绍（姓名、年龄、地址）	人物出场介绍（姓名、年龄、地址）
起床（起床）	起床（起床啦）
洗漱（刷牙、洗脸）	洗漱（刷牙、洗脸）
吃饭（吃菜、扒饭）	吃饭（吃菜、扒饭）
登录账号（账号登录成功）	打卡（今日打卡成功）
学习（看视频、写代码）	工作（授课、答疑、写代码）
吃饭（吃菜、扒饭）	吃饭（吃菜、扒饭）
学习（看视频、写代码）	工作（授课、答疑、写代码）
吃饭（吃菜、扒饭）	吃饭（吃菜、扒饭）
洗漱（刷牙、洗脸）	洗漱（刷牙、洗脸）
睡觉（睡觉）	睡觉（睡觉）
人数统计（统计、公布）	人数统计（统计、公布）

面向过程（早期语言的编程）

```
count_s = 0
stu1 = "张三"
age_s = 18
adres_s = "黄土高坡"

print(f"大家好！ 我是{stu1}，今年{age_s}岁，家住在{adres_s}，欢迎大家有空来玩哦！")

count_s += 1

print(f"{stu1}起床")

print(f"{stu1}刷牙")
```

```
print(f"{stu1}洗脸")

print(f"{stu1}吃菜")
print(f"{stu1}扒饭")

print(f"{stu1}账号登录成功")

print(f"{stu1}看视频")
print(f"{stu1}写代码")

print(f"{stu1}吃菜")
print(f"{stu1}扒饭")

print(f"{stu1}看视频")
print(f"{stu1}写代码")

print(f"{stu1}吃菜")
print(f"{stu1}扒饭")

print(f"{stu1}刷牙")
print(f"{stu1}洗脸")

print(f"{stu1}睡觉")

print(f"当前统计的学生人数是: {count_s} 人")
```

```
count_t = 0
teacher1 = "老王"
age_t = 40
adres_t = "人民广场"

print(f"大家好! 我是{teacher1}, 今年{age_t}岁, 家住在{adres_t}, 欢迎
大家有空来玩哦!")

count_t += 1

print(f"{teacher1}起床")

print(f"{teacher1}刷牙")
```

```
print(f"{teacher1}洗脸")

print(f"{teacher1}吃菜")
print(f"{teacher1}扒饭")

print(f"{teacher1}今日打卡成功")

print(f"{teacher1}授课")
print(f"{teacher1}答疑")
print(f"{teacher1}写代码")

print(f"{teacher1}吃菜")
print(f"{teacher1}扒饭")

print(f"{teacher1}授课")
print(f"{teacher1}答疑")
print(f"{teacher1}写代码")

print(f"{teacher1}吃菜")
print(f"{teacher1}扒饭")

print(f"{teacher1}刷牙")
print(f"{teacher1}洗脸")

print(f"{teacher1}睡觉")

print(f"当前统计的老师人数是: {count_t} 人")
```

学生	老师
人物出场介绍（姓名、年龄、地址）	人物出场介绍（姓名、年龄、地址）
起床（起床啦）	起床（起床啦）
定义洗漱函数、吃饭函数、学习函数	定义洗漱函数、吃饭函数、工作函数
调用洗漱函数	调用洗漱函数
调用吃饭函数	调用吃饭函数
登录账号（账号登录成功）	打卡（今日打卡成功）

学生	老师
调用学习函数	调用工作函数
调用吃饭函数	调用吃饭函数
调用学习函数	调用工作函数
调用吃饭函数	调用吃饭函数
调用洗漱函数	调用洗漱函数
睡觉（睡觉）	睡觉（睡觉）
人数统计（统计、公布）	人数统计（统计、公布）

面向过程（结构化编程）

```
count_s = 0
stu1 = "张三"
age_s = 18
adres_s = "黄土高坡"

print(f"大家好！ 我是{stu1}，今年{age_s}岁，家住在{adres_s}，欢迎大家有空来玩哦！")

count_s += 1

def wash(name):
    print(f"{name}刷牙")
    print(f"{name}洗脸")


def eat(name):
    print(f"{name}吃菜")
    print(f"{name}扒饭")


def study(name):
    print(f"{stu1}看视频")
    print(f"{stu1}写代码")
```

```
print(f"{stu1}起床")

wash(stu1)

eat(stu1)

print(f"{stu1}账号登录成功")

study(stu1)

eat(stu1)

study(stu1)

eat(stu1)

wash(stu1)

print(f"{stu1}睡觉")

print(f"当前统计的学生人数是: {count_s} 人")
```

```
count_t = 0
teacher1 = "老王"
age_t = 40
adres_t = "人民广场"

print(f"大家好! 我是{teacher1}, 今年{age_t}岁, 家住在{adres_t}, 欢迎大家有空来玩哦!")

count_t += 1

def wash(name):
    print(f"{name}刷牙")
    print(f"{name}洗脸")


def eat(name):
    print(f"{name}吃菜")
```

```
    print(f"{name}扒饭")


def work(name):
    print(f"{name}授课")
    print(f"{name}答疑")
    print(f"{name}写代码")


print(f"{teacher1}起床")

wash(teacher1)

eat(teacher1)

print(f"{teacher1}今日打卡成功")

work(teacher1)

eat(teacher1)

work(teacher1)

eat(teacher1)

wash(teacher1)

print(f"{teacher1}睡觉")

print(f"当前统计的老师人数是: {count_t} 人")
```

学生	老师	人
创建一个学生 类	创建一个老师 类	创建一个人 类
针对某个学生的属性作为： 实例变量	针对某个老师的属性作为： 实例变量	让学生类和老师类 继承 人类
不针对某个学生而针对整个类的属性作为：类变量	不针对某个老师而针对整个类的属性作为：类变量	把 子类 共有的变量和方法写到 父类 中，而子类可不用写

学生	老师	人
针对某个学生的功能作为： 对象方法	针对某个老师的功能作为： 对象方法	把 子类 独有的变量和方法留在 子类 中
不针对某个学生而针对整个类的功能作为：类方法	不针对某个老师而针对整个类的功能作为：类方法	

面向对象（高内聚低耦合）

程序 = 数据 + 算法。面向过程编程，更侧重于算法；而面向对象编程更侧重于数据。

```
class Student:
    count = 0

    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address
        self.show_time()
        Student.count += 1

    def show_time(self):
        print(f"大家好！ 我是{self.name}, 今年{self.age}岁, 家住在
{self.address}, 欢迎大家有空来玩哦!")

    def get_up(self):
        print(f"{self.name}起床")

    def wash(self):
        print(f"{self.name}刷牙")
        print(f"{self.name}洗脸")

    def eat(self):
        print(f"{self.name}吃菜")
        print(f"{self.name}扒饭")

    def login_ID(self):
        print(f"{self.name}账号登录成功")
```

```
    def study(self):
        print(f"{self.name}看视频")
        print(f"{self.name}写代码")

    def sleep(self):
        print(f"{self.name}睡觉")

    @classmethod
    def counter(cls):
        print(f"当前统计的学生人数是: {cls.count} 人")


stu1 = Student("张三", 18, "黄土高坡")
stu1.get_up()
stu1.wash()
stu1.eat()
stu1.login_ID()
stu1.study()
stu1.eat()
stu1.study()
stu1.eat()
stu1.wash()
stu1.sleep()
stu1.counter()
```

```
class Teacher:
    count = 0

    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address
        self.show_time()
        Teacher.count += 1

    def show_time(self):
        print(f"大家好! 我是{self.name}, 今年{self.age}岁, 家住在
{self.address}, 欢迎大家有空来玩哦!")
```

```
    def get_up(self):
        print(f"{self.name}起床")

    def wash(self):
        print(f"{self.name}刷牙")
        print(f"{self.name}洗脸")

    def eat(self):
        print(f"{self.name}吃菜")
        print(f"{self.name}扒饭")

    def clock_in(self):
        print(f"{self.name}今日打卡成功")

    def work(self):
        print(f"{self.name}授课")
        print(f"{self.name}答疑")
        print(f"{self.name}写代码")

    def sleep(self):
        print(f"{self.name}睡觉")

    @classmethod
    def counter(cls):
        print(f"当前统计的老师人数是: {cls.count} 人")


teacher1 = Teacher("老王", 40, "人民广场")
teacher1.get_up()
teacher1.wash()
teacher1.eat()
teacher1.clock_in()
teacher1.work()
teacher1.eat()
teacher1.work()
teacher1.eat()
teacher1.wash()
teacher1.sleep()
```

```
teacher1.counter()
```

```
class Person:

    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address
        self.show_time()

    def show_time(self):
        print(f"大家好！ 我是{self.name}, 今年{self.age}岁, 家住在
{self.address}, 欢迎大家有空来玩哦!")

    def get_up(self):
        print(f"{self.name}起床")

    def wash(self):
        print(f"{self.name}刷牙")
        print(f"{self.name}洗脸")

    def eat(self):
        print(f"{self.name}吃菜")
        print(f"{self.name}扒饭")

    def sleep(self):
        print(f"{self.name}睡觉")


class Student(Person):

    count = 0

    def __init__(self, name, age, address, classes):
        super().__init__(name, age, address)
        self.classes = classes
        Student.count += 1

    def login_ID(self):
```

```
        print(f"{self.name}账号登录成功")

    def study(self):
        print(f"{self.name}看视频")
        print(f"{self.name}写代码")

    @classmethod
    def counter(cls):
        print(f"当前统计的学生人数是: {cls.count} 人")


class Teacher(Person):
    count = 0

    def __init__(self, name, age, address, department):
        super().__init__(name, age, address)
        self.department = department
        Teacher.count += 1

    def clock_in(self):
        print(f"{self.name}今日打卡成功")

    def work(self):
        print(f"{self.name}授课")
        print(f"{self.name}答疑")
        print(f"{self.name}写代码")

    @classmethod
    def counter(cls):
        print(f"当前统计的老师人数是: {cls.count} 人")


stu1 = Student("张三", 18, "黄土高坡", "高三1班")
Student.counter()

teacher1 = Teacher("老王", 38, "人民广场", "教育部")
teacher2 = Teacher("老翁", 39, "黄浦江", "教育部")
teacher3 = Teacher("老李", 40, "黄浦江", "教导处")
Teacher.counter()
```

类、对象、变量、方法

类变量、实例变量、实例化

```
# 定义一个类对象，类的名字首字母通常大写
class Student(object):  # object 是所有类的基类，通常省略不写

    school = '二中'  # 类变量

    def __init__(self, name):
        self.name = name  # 实例变量


"""
对__new__(cls)方法介绍:
构造方法。它会将请求实例化所属的类作为实参传给cls(其他实参传给__init__)，
创建并返回这个类的实例对象，通常不需要显示的声明该方法，因为父类的
object 中有定义

对__init__(self)方法介绍:
初始化方法。用来定制实例变量，返回值只能是None(因为负责返回实例对象的是构
造方法)
"""


"""
实例化时，会先调用__new__(cls)方法，在实例对象被创建之后且返回给调用者之
前，
再调用__init__(self)方法，把实例对象传给self参数，做进一步的定制
"""
stu1 = Student('小明')  # 实例化，得到实例对象stu1
stu2 = Student('小红')  # 实例化，得到实例对象stu2

""" 实例变量的调用规则 """
```

```
print(stu1.name)  # 实例对象stu1调用实例变量
print(stu2.name)  # 实例对象stu2调用实例变量
# print(Student.name)  # 类对象调用不到实例变量

""" 类变量的调用规则 """
print(Student.school)  # 类对象可以直接调用类变量(推荐)
print(stu1.school)  # 实例对象stu1也可以调用类变量
print(stu2.school)  # 实例对象stu2也可以调用类变量

""" 实例变量是每个实例对象所独有的 """
stu1.name = '小强'  # 把实例对象stu1的name变量指向新的值
print(stu1.name)  # 实例对象stu1再次调用实例变量，输出新的值
print(stu2.name)  # 而实例对象stu2再次调用实例变量，并不受到影响

""" 类变量在整个实例对象中是公用的
类对象调用类变量重新赋值，会影响所有的对象对该类变量的调用 """
Student.school = '一中'  # 把Student类的school变量指向新的值
print(Student.school)  # 类对象再次调用类变量，输出新的值
print(stu1.school)  # 实例对象stu1再次调用类变量，也输出新的值
print(stu2.school)  # 实例对象stu2再次调用类变量，也输出新的值

""" 实例对象调用类变量重新赋值，其实是在动态定义变量，根本与类变量无关
"""
stu1.school = '三中'  # 动态定义变量，定义了stu1对象的一个实例变量
print(stu1.school)  # 实例对象stu1调用上一步定义的实例变量
print(Student.school)  # 而类变量还是那个类变量
print(stu2.school)  # 而类变量还是那个类变量
```

类方法、对象方法、静态方法

- 在 Python 中，一般把定义在类中的函数叫方法（method），不在类中的叫函数（function）
- 方法通常有参数，比如对象方法隐式的接收了 self 参数，类方法隐式的接收了 cls 参数
- 函数可以没有参数

```
```

```python
class Student:

    @staticmethod  # 静态方法 装饰器
    def eat():
        print('我要开动了~')

    @classmethod  # 类方法 装饰器
    def sleep(cls):  # cls表示当前调用的类对象
        print('我要就寝了~')

    def init(self, age):  # self表示当前调用的实例对象
        self.age = age
        print(f'我{self.age}岁开始学习')


# self, cls不是关键字，可以换成自己写的其他任意名字代替，调用的时候统一就可以了
# 静态方法：类可以直接调用，实例对象也可以调用，推荐类调用
# 类方法：类可以直接调用，实例对象也可以调用，推荐类调用
# 对象方法：实例对象调用，类不可调用
stu = Student()
Student.eat()
Student.sleep()
stu.eat()
stu.sleep()

# print(stu.age)  # 报错(stu没有age变量)
stu.init(7)  # 可能会碰到另外一种写法：Student.init(stu, 7)
print(stu.age)  # 不报错(上一步调用方法之后，创建了self.age变量)
```

- 思考：静态方法、类方法有什么区别？

```python
class A:

    var1 = 123

    @classmethod
    def func1(cls):
        print(cls.var1)
```

```
    @staticmethod
    def func2():
        print(A.var1)


class B(A):

    var1 = 321


A.func1()
A.func2()
B.func1()
B.func2()
```

动态定义变量

```
class Student:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def show_info(self):
        print(f'名字:{self.name}, 年龄:{self.age}, 地址:
{self.address}')

    def study1(self):
        self.course = "语文"
        print(f'今天学习的科目是:{self.course}')

    def study2(self):
        print(f'今天学习的科目是:{self.course}')
```

```python
stu1 = Student('张三', 18)  # 实例化
stu2 = Student('小明', 28)  # 实例化

# stu1.show_info()  # 报错(因为stu1没有address变量)

stu1.address = '黄土高坡'  # stu1动态定义实例变量
print(stu1.address)  # 输出: 黄土高坡
# print(stu2.address)  # 报错(因为实例变量是每个实例对象所独有的)

stu1.show_info()  # 不报错(因为stu1动态定义了address变量)

# stu2.study2()  # 报错(因为stu2没有course变量)
stu2.study1()  # 不报错(因为study1方法中定义了course变量)
stu2.study2()  # 不报错(因为study1方法执行之后, course变量已经被定义了)

stu2.course = '数学'  # 动态定义实例变量
stu2.study1()  # study1方法中会把self.course重写赋值



class Worker:

    def __init__(self, name):
        self.name = name


wk = Worker('李四')
Worker.factory = "江南皮革厂"  # 动态定义类变量
print(worker.factory)  # 类对象调用类变量(推荐)
print(wk.factory)  # 实例对象也可以调用类变量

# print(Worker.name)  # 报错(类对象调用不到实例变量)
Worker.name = '王五'  # 动态定义类变量
print(Worker.name)  # 类对象调用类变量
print(wk.name)  # 实例对象优先调用实例变量name, 而不调用类变量name
```

与属性操作相关的内置函数

delattr(object, name)

- 删除 object 的 name 属性（name 参数为字符串）

```
class Person:
    eat = "rice"

    def __init__(self, age):
        self.age = age


p = Person(18)
print(Person.eat)
""" 等价于 del Person.eat """
delattr(Person, "eat")  # 删除类属性eat
print(Person.eat)


print(p.age)
""" 等价于 del p.eat """
delattr(p, "age")  # 删除实例属性age
print(p.age)
```

getattr(object, name[, default])

- 返回 object 对象的 name 属性值（name 参数为字符串）
- 如果 name 属性不存在，且提供了 default 值，则返回它，否则触发 AttributeError

```
class Person:
    eat = "rice"

    def __init__(self, age):
        self.age = age


p = Person(18)
```

```
""" 等价于 Person.eat """
print(getattr(Person, "eat"))

""" 等价于 p.age """
print(getattr(p, "age"))

print(getattr(p, "height", 178))
# print(getattr(p, "height"))
```

hasattr(object, name)

- 判断 object 对象是否包含 name 属性（name 参数为字符串），返回 True 或 False
- 此功能是通过调用 getattr(object, name) 看是否有 AttributeError 异常来实现的

```
class Person:
    eat = "rice"

    def __init__(self, age):
        self.age = age


p = Person(18)
print(hasattr(Person, "eat"))  # True
print(hasattr(p, "eat"))  # True
print(hasattr(p, "age"))  # True
print(hasattr(p, "height"))  # False
```

setattr(object, name, value)

- 将 object 的 name 属性设置为 value，属性不存在则新增属性（name参数为字符串）

```
class Person:
    eat = "rice"
```

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


p = Person(18)
setattr(Person, "eat", "noodles")
print(Person.eat)

setattr(Person, "drink", "water")
print(Person.drink)

setattr(p, "age", 29)
print(p.age)

setattr(p, "height", 178)
print(p.height)
```

面向对象三大特性

封装

- 在属性或者方法前面加两个下划线开头, 声明为私有属性或者私有方法
- 私有属性或者私有方法只能在类的内部调用, 不能在类的外部直接调用
- 但是可以提供非私有方法来访问私有属性, 或者调用私有方法
- 子类无法继承父类的私有属性 和 私有方法

```
class Person:

    def __init__(self, name, age):
        self.__name = name  # 私有属性
        if age <= 0:
            self.__age = "年龄必须大于0"  # 私有属性
```

```
            else:
                self.__age = age  # 私有属性

    # 利用非私有方法访问私有属性
    def show_info(self):
        print(f"姓名:{self.__name}\n年龄:{self.__age}")

    # 私有方法
    def __sleep(self):
        print("我要睡觉了, 晚安!")

    # 利用非私有方法调用私有方法
    def sleep(self):
        self.__sleep()


ps = Person("赵六", 26)
# print(ps.__name)  # 报错(私有属性不能在类的外部直接访问)
# print(ps.__age)  # 同上
ps.show_info()

# ps.__sleep()  # 报错(私有方法不能在类的外部直接调用)
ps.sleep()
```

```
class Person:

    __school = "调用__school成功"  # 私有属性
    school = "调用school成功"  # 非私有属性

    def __sleep(self):  # 私有方法
        print("调用__sleep成功")

    def sleep(self):  # 非私有方法
        print("调用sleep成功")


class Student(Person):
    pass
```

```python

stu = Student()
print(stu.school)  # 子类继承父类的非私有属性
stu.sleep()  # 子类继承父类的非私有方法

print(stu.__school)  # 子类无法继承父类的私有属性
stu.__sleep()  # 子类无法继承父类的私有方法
```

```python
"""
继承时，子类是不会继承父类的私有属性和方法
但是，如果子类是在父类中去调用父类的私有属性和方法，那么是可以的
因为私有属性和方法可以在被封装的那个类的区域直接调用
"""

class Person:

    def __init__(self):
        print(self)
        self.__func()  # 可以调用


    def __func(self):
        print(1234)


class Student(Person):
    pass


stu1 = Student()
stu1.__func()  # 调用不到
```

继承

- 所有的类都默认继承 object，只是一般不用写出来
- 子类继承父类后，会拥有父类中所有的非私有属性和方法
- 继承的作用：从子类来看，继承可以简化代码；从父类来看，子类是对父类功能的扩充

```
class A:  # class A(object) 每一个类默认继承 class object
    pass

class B(A): # class B(A, object) 每一个类默认继承 class object
    pass
```

单继承

```
class Person:

    state = "China"

    def eat(self):
        print('吃饭')

    def speak(self):
        print('说话')


class Student(Person):

    def study(self):
        print('读书')


class Worker(Person):

    def work(self):
        print('搬砖')
```

```
stu = Student()
print(Student.state)  # 子类调用父类的属性
stu.study()  # 子类调用自己的方法
stu.eat()  # 子类调用父类的方法
stu.speak()  # 子类调用父类的方法

wk = Worker()
print(Worker.state)
wk.work()  # 子类调用自己的方法
wk.eat()  # 子类调用父类的方法
wk.speak()  # 子类调用父类的方法
```

```
class Animal:
    def eat(self):
        print('吃东西')

class Cat(Animal):
    def catch_mouse(self):
        print('抓老鼠')

class Ragdoll(Cat):
    def cute(self):
        print('卖萌')


c1 = Ragdoll()
c1.cute()  # 子类调用自己的方法
c1.catch_mouse()  # 子类调用父类的方法
c1.eat()  # 子类调用父类的父类的方法
```

多重继承

```
class Animal:
    def eat(self):
        print('吃东西')
```

```
class Cat:
    def catch_mouse(self):
        print('抓老鼠')

class Ragdoll(Cat, Animal):  # 继承多个父类
    def cute(self):
        print('卖萌')


c1 = Ragdoll()
c1.cute()  # 子类调用自己的方法
c1.catch_mouse()  # 子类调用Cat父类的方法
c1.eat()  # 子类调用Animal父类的方法
```

继承顺序

- 单继承查找顺序：先找自己的，再去找父类，再去找父类的父类，依此类推
- 多重继承查找顺序：先找自己的，再按照从左往右的顺序依次找父类的
- 当继承比较复杂时，可以使用__mro__属性查看搜索顺序

```
class A:
    a = 1
    def pr(self):
        print('A')

class B(A):
    def pr(self):
        print('B')

class C(B):
    a = 3
    def pr(self):
        print('C')

class D(B):
    pass
```

```
c = C()
c.pr()
print(c.a)

d = D()
d.pr()
print(d.a)
```

```
class Biology:

    def eat(self):
        print("Biology吃东西")


class Animal:

    def sleep(self):
        print("Animal睡觉")

    def cute(self):
        print("Animal卖萌")


class Cat:

    def sleep(self):
        print("Cat睡觉")


class Ragdoll(Cat, Animal, Biology):
    pass


rd = Ragdoll()
rd.eat()
rd.sleep()
rd.cute()
```

方法重写

- 在继承中，当父类方法的功能不能满足需求时，可以在子类重写父类的方法

```
class Animal:

    def __init__(self, food):
        self.food = food

    def eat(self):
        print(f"动物吃{self.food}")


class Cat(Animal):

    # 为了实现'猫吃鱼'的功能，而不是父类的'动物吃鱼'，在子类对eat方法重写
    def eat(self):
        print(f"猫吃{self.food}")


c = Cat("鱼")  # 实例化，调用父类的初始化方法
c.eat()
```

super()

- super是内置的类，可以调用指定类的父类（超类）
- 适用场景：在子类重写父类方法后，想再使用父类的该方法

```
class Animal:

    def eat(self):
        print("吃东西")
```

```
class Cat(Animal):

    def eat(self):
        print("吃鱼")


class Ragdoll(Cat):

    def eat(self):
        print("喝咖啡")


rd = Ragdoll()
rd.eat()  # rd调用Ragdoll类中的对象方法
super(Ragdoll, rd).eat()  # rd调用Ragdoll父类的对象方法
super(Cat, rd).eat()  # rd调用Cat父类的对象方法

c = Cat()
c.eat()  # c调用Cat类中的对象方法
super(Cat, c).eat()  # c调用Cat父类的对象方法
```

继承中的**__init__**方法

```
class A:

    def E(self):
        print('E方法被调用')

    def __init__(self, name):
        self.name = name
        self.Q()

    def Q(self):
        print(self.name, 'Q方法被调用')


class B(A):
```

```
    pass


b = B('张三')  # 实例化,调用初始化方法,B没有则调用父类中的初始化方法,初始化方法中调用了Q方法
b.E()  # 调用父类的E方法
b.Q()  # 调用父类的Q方法


class C(A):

    def __init__(self, name):
        self.names = name


c = C('赵六')  # 实例化, 优先调用C中初始化方法

''' 虽然可以调用父类的Q方法, 但是因为Q方法中的self.name没有定义, 因为A的初始化方法没有被调用, 所以报错
解决方案: 先通过c调用一次A的初始化方法 或者 把C类中的self.names改为self.name '''
# c.Q()  # 报错



class D(A):
    def __init__(self, name):
        super(D, self).__init__('李四')
        self.name = name


d = D('王五')  # 实例化, 先调用D的初始化方法, super方法调用父类的初始化方法, 父类的初始化方法中调用Q方法
d.Q()  # 调用父类的Q方法
```

与继承相关的两个内置函数

isinstance(object, classinfo)

- object：实例对象
- classinfo：类名、基本类型或者由它们组成的元组
- 如果 object 是 classinfo 的实例或者是其子类的实例，则返回 True
- 如果 object 不是给定类型的对象，则返回 False
- 如果 classinfo 是类型对象元组，那么如果 object 是其中任何一个类型的实例或其子类的实例，就返回 True
- 如果 classinfo 既不是类型，也不是类型元组或类型元组的元组，则将引发 TypeError 异常

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass


a = A()
b = B()
c = C()
print(isinstance(a, A))  # True
print(type(a) == A)  # True
print(isinstance(b, A))  # True，考虑继承
print(type(b) == A)  # False，type不考虑继承
print(isinstance(c, A))  # True，考虑继承
print(type(c) == A)  # False，type不考虑继承
print(isinstance(c, (B, A)))  # True，C是A子类的实例
```

issubclass(class, classinfo)

- 如果 class 是 classinfo 的子类则返回 True
- 类会被视作其自身的子类
- classinfo 也可以是类对象的元组，只要 class 是其中任何一个类型的子类，就返回 True

```
```

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass


print(issubclass(B, A))  # True
print(issubclass(C, A))  # True
print(issubclass(A, A))  # True，类会被视作其自身的子类
print(issubclass(C, (B, A)))  # True
```

多态性

- 多态性是指具有不同内容的方法可以使用相同的方法名，则可以用一个方法名调用不同内容的方法

```
class Apple:
    def change(self):
        return '啊~ 我变成了苹果汁!'


class Banana:
    def change(self):
        return '啊~ 我变成了香蕉汁!'


class Mango:
    def change(self):
        return '啊~ 我变成了芒果汁!'


class Juicer:
```

```
    def work(self, fruit):
        print(fruit.change())

"""
三个内容不同的change方法使用相同的名字命名，
只要改变change的调用对象，就可以调用不同内容的方法
"""
a = Apple()
b = Banana()
m = Mango()
j = Juicer()
j.work(a)
j.work(b)
j.work(m)
```