《Python程序设计》

Python类与对象

刘潇
机械科学与工程学院

2023年10月30日

本节要点

- □ 了解面向对象编程的思维方式
- □ 掌握Python中类与对象的属性和方法的创建
- □ 了解Python面向对象编程的继承、多态和封装的特点

主要内容

1. 面向对象编程

2. 类和对象

3. 特殊方法

4. 继承、多态和封装

面向对象编程

面向过程编程是首先分析解决问题所需要的步骤，然后用函数或者模块把这些步骤一步一步实现， 通过依次调用达到目的

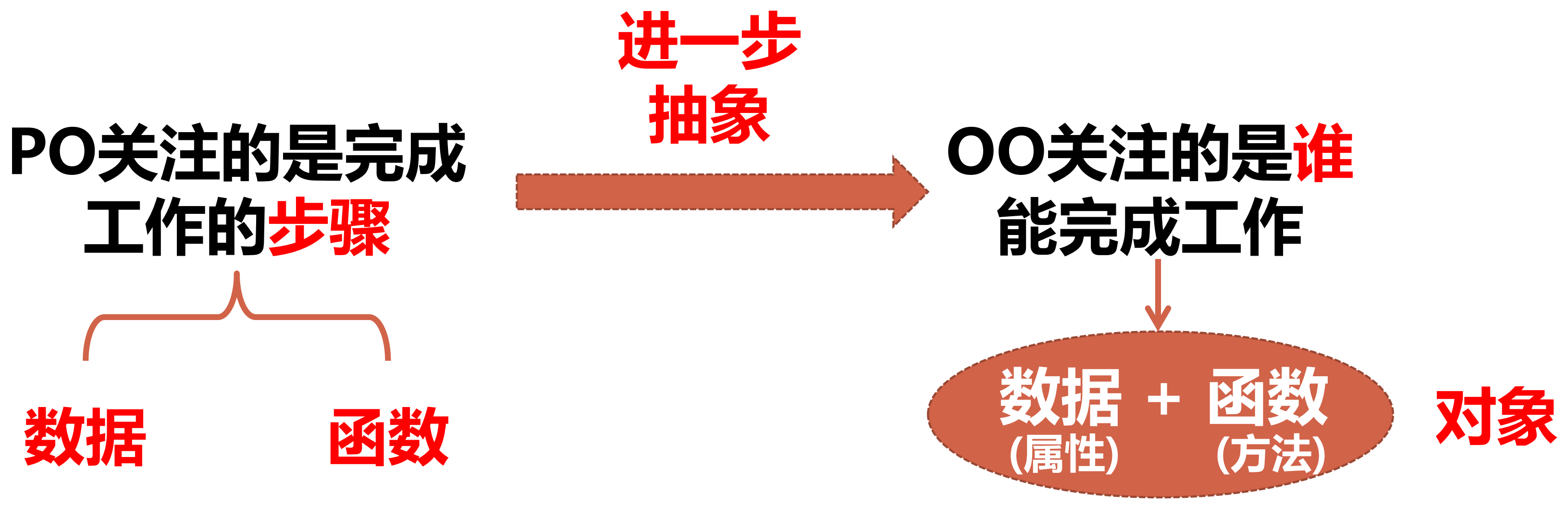
Process Oriented（PO）程序设计步骤：

- **分析程序从输入到输出的各步骤**
- **按照执行过程从前到后编写程序**
- **将高耦合部分封装成模块或函数**
- **输入参数，按照程序执行过程调试**

1. **番茄**切块；
2. **鸡蛋**磕入碗搅匀；
3. 淀粉加水调开；
4. 蒜瓣切片；
5. 炒菜锅放少量的油，然后放蒜片；
6. 待出香味，**放番茄翻炒**到皮软出汁，放适量的水、鸡精、盐炒匀；
7. 水开了慢慢把湿淀粉倒进锅中，并不停搅动；
8. 锅再次开后，用筷子搅拌着**鸡蛋慢慢转着倒进去**；
9. 再次开锅即可关火，装汤盆，撒葱花。

面向对象编程（Object Oriented, OO）

OO是把构成问题的事物分解成对象，建立对象的目的不是为了完成一个步骤，而是描述事物在解决问题的步骤中的行为



OO将同类型对象抽象出其共性，形成类。类通过简单的外部接口与外界发生关系，通过继承与多态性提高程序的可重用性

OO是一种思维方式

求斐波拉契数列

打印出2到20内的斐波那契数列 F(1) = 1, F(2) = 1, F(n) = F(n-1) + F(n-2)（n>=2，n∈N*）

函数编程

```python
from functools import reduce
reduce(lambda  list1, number: number == list1[-1] + list1[-2] and list1 + [number] or list1, \
        range(2,20), [1,1])
```

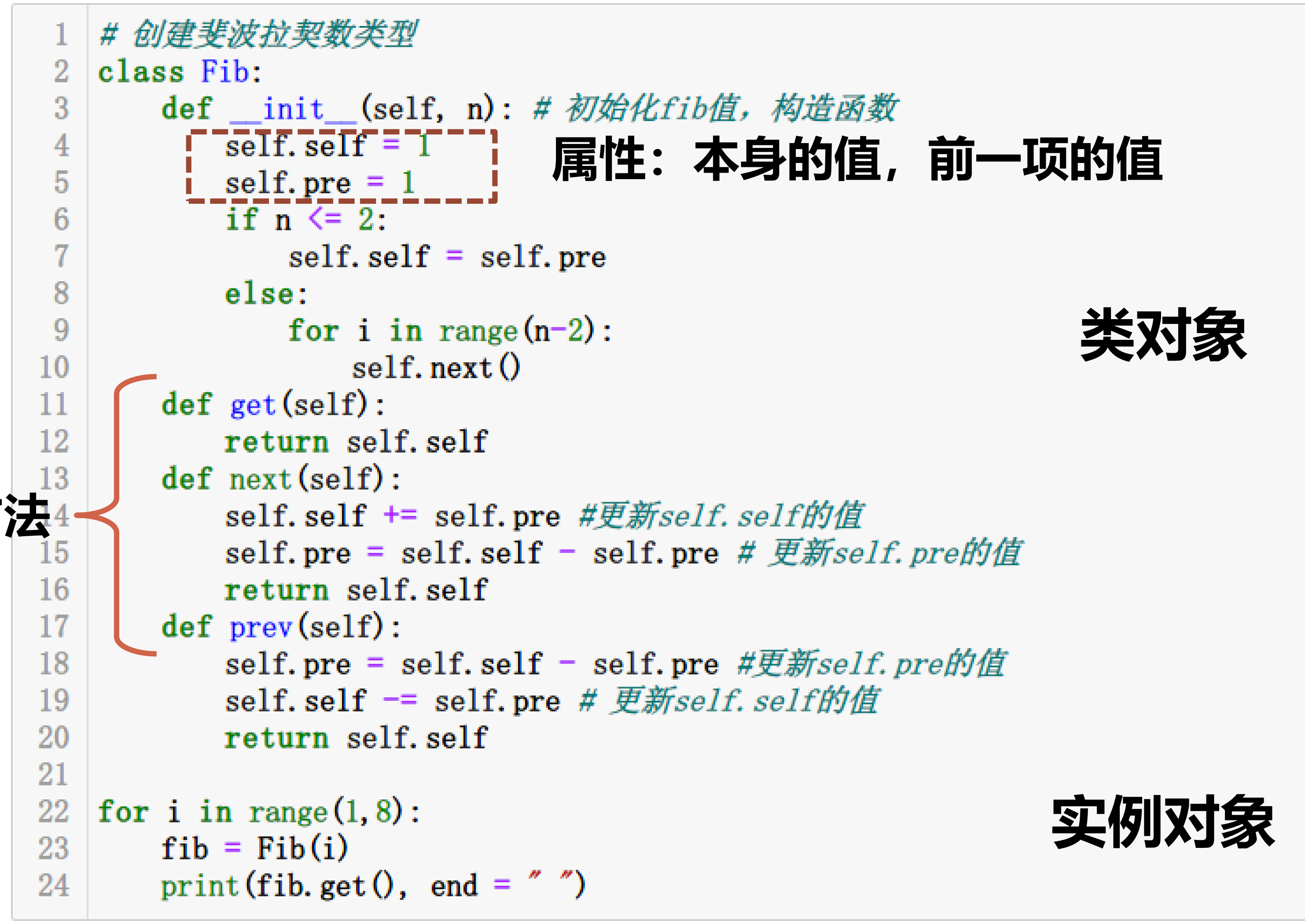
```
[1, 1, 2, 3, 5, 8, 13]
```

面向过程编程

```python
list1 = [1, 1]
i = len(list1)
for number in range(2, 20):
    if number == list1[i-1] + list1[i-2]:
        list1.append(number)
        i += 1
print(list1)
```

```
[1, 1, 2, 3, 5, 8, 13]
```

面向对象编程



```
# 创建斐波拉契数类型
class Fib:
    def __init__(self, n): # 初始化fib值，构造函数
        self.self = 1
        self.pre = 1
        if n <= 2:
            self.self = self.pre
        else:
            for i in range(n-2):
                self.next()
    def get(self):
        return self.self
    def next(self):
        self.self += self.pre #更新self.self的值
        self.pre = self.self - self.pre # 更新self.pre的值
        return self.self
    def prev(self):
        self.pre = self.self - self.pre #更新self.pre的值
        self.self -= self.pre # 更新self.self的值
        return self.self

for i in range(1,8):
    fib = Fib(i)
    print(fib.get(), end = " ")
```

属性：本身的值，前一项的值

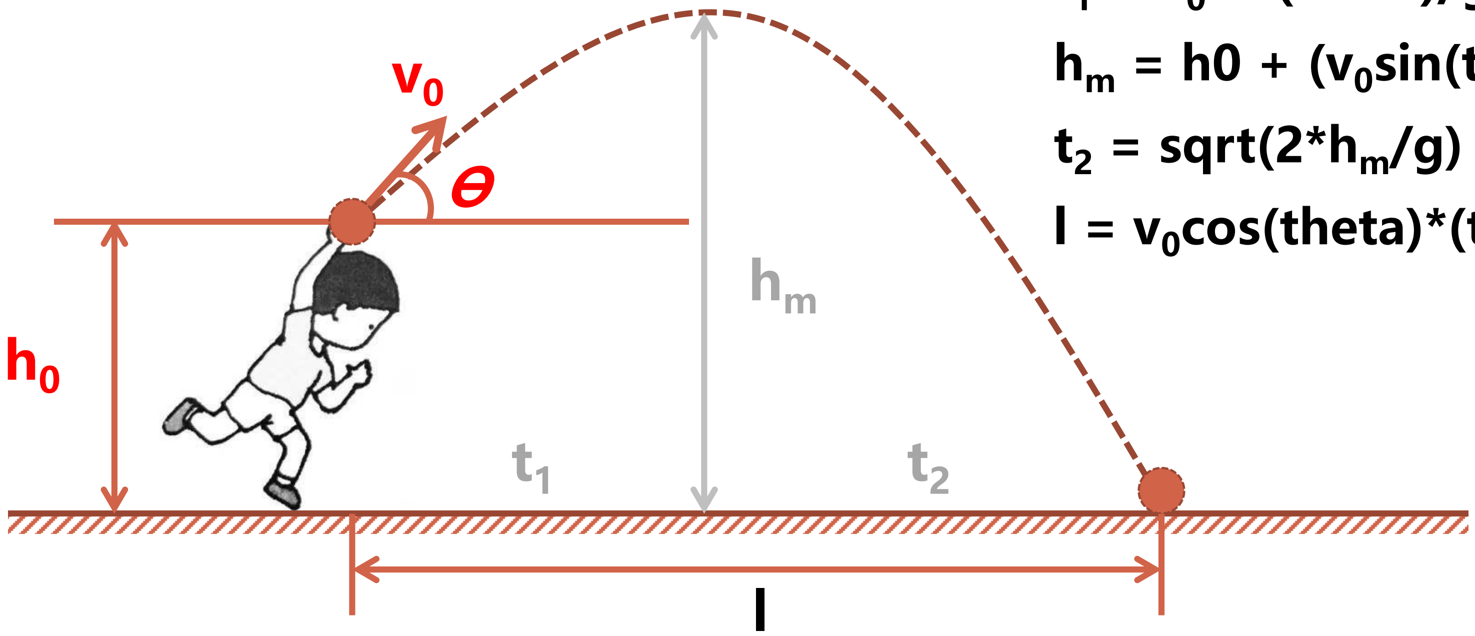
类对象

方法

实例对象

```
1 1 2 3 5 8 13
```

计算铅球飞行距离l



t_1 = $v_0$sin(theta)/g

h_m = h0 + ($v_0$sin(theta))^2/2g

t_2 = sqrt(2*h_m/g)

l = $v_0$cos(theta)*(t_1+t_2)

输入：

初始高度：h_0 （米）

初始速度：v_0 （米）

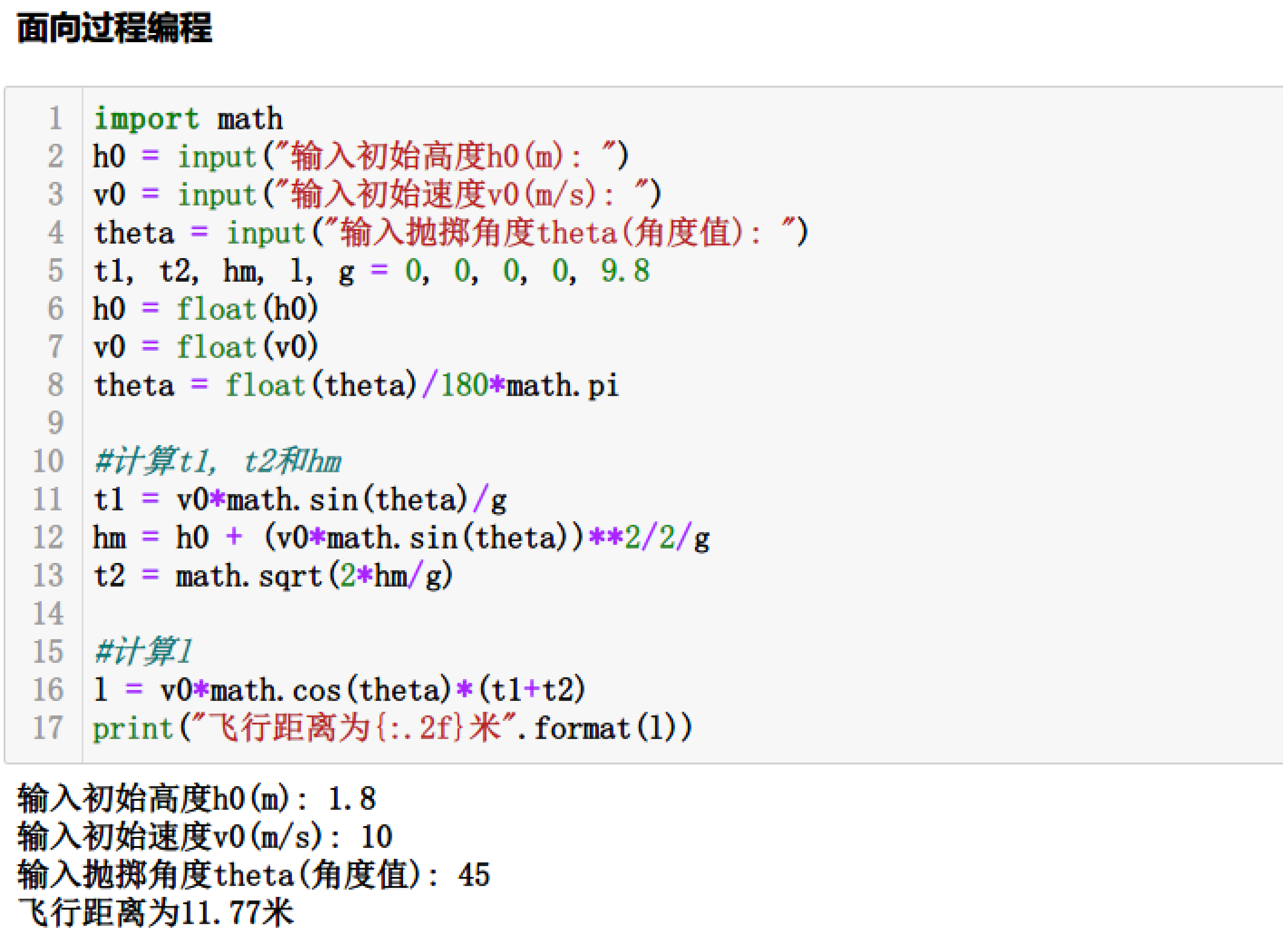
抛掷角度：theta （角度）

输出：

飞行距离：l （米）

假设：忽略空气阻力，重力加速度g为9.8 m/s^2

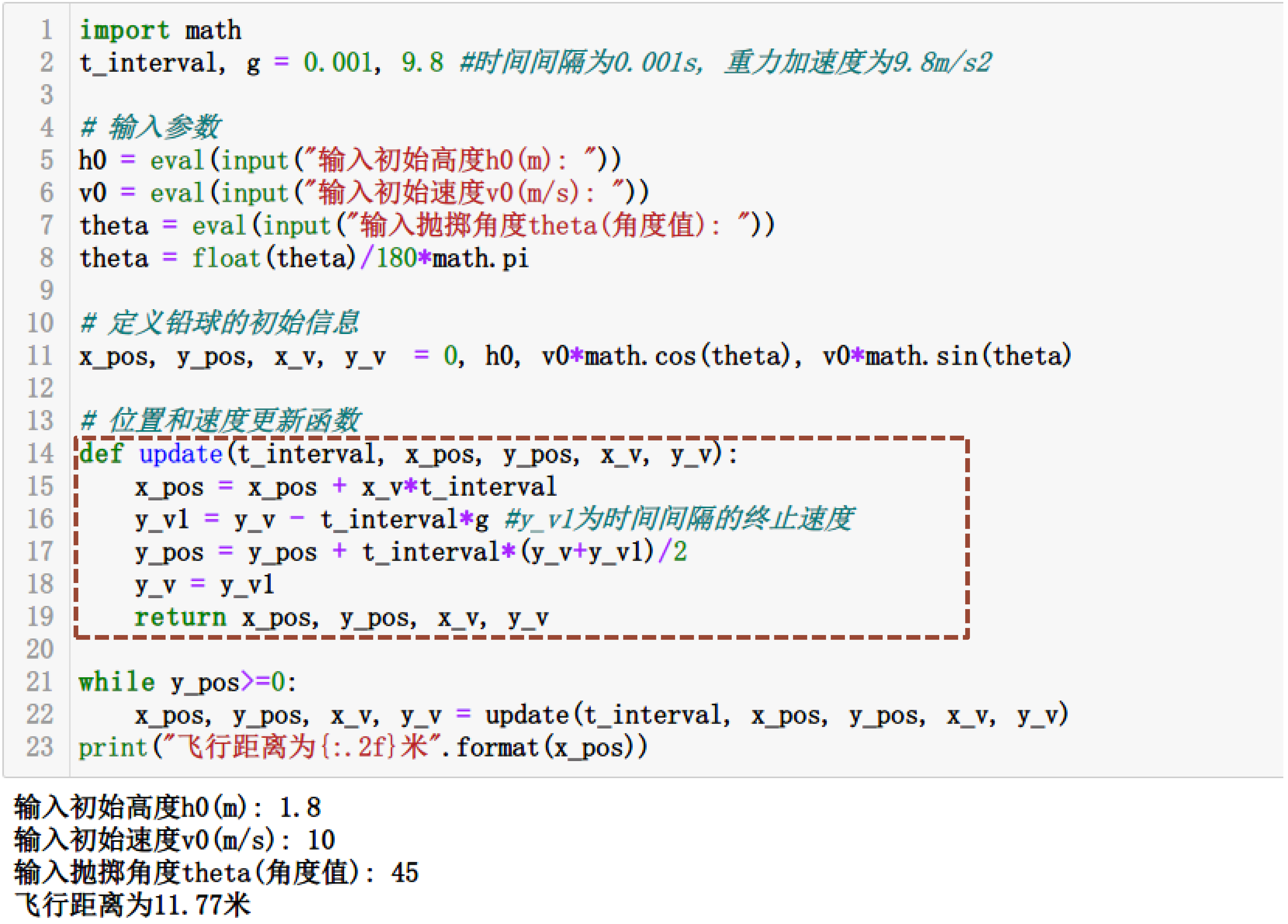
面向过程编程



面向过程编程

```python
import math
h0 = input("输入初始高度h0(m)：")
v0 = input("输入初始速度v0(m/s)：")
theta = input("输入抛掷角度theta(角度值)：")
t1, t2, hm, l, g = 0, 0, 0, 0, 9.8
h0 = float(h0)
v0 = float(v0)
theta = float(theta)/180*math.pi

#计算t1, t2和hm
t1 = v0*math.sin(theta)/g
hm = h0 + (v0*math.sin(theta))**2/2/g
t2 = math.sqrt(2*hm/g)

#计算l
l = v0*math.cos(theta)*(t1+t2)
print("飞行距离为{:.2f}米".format(l))
```

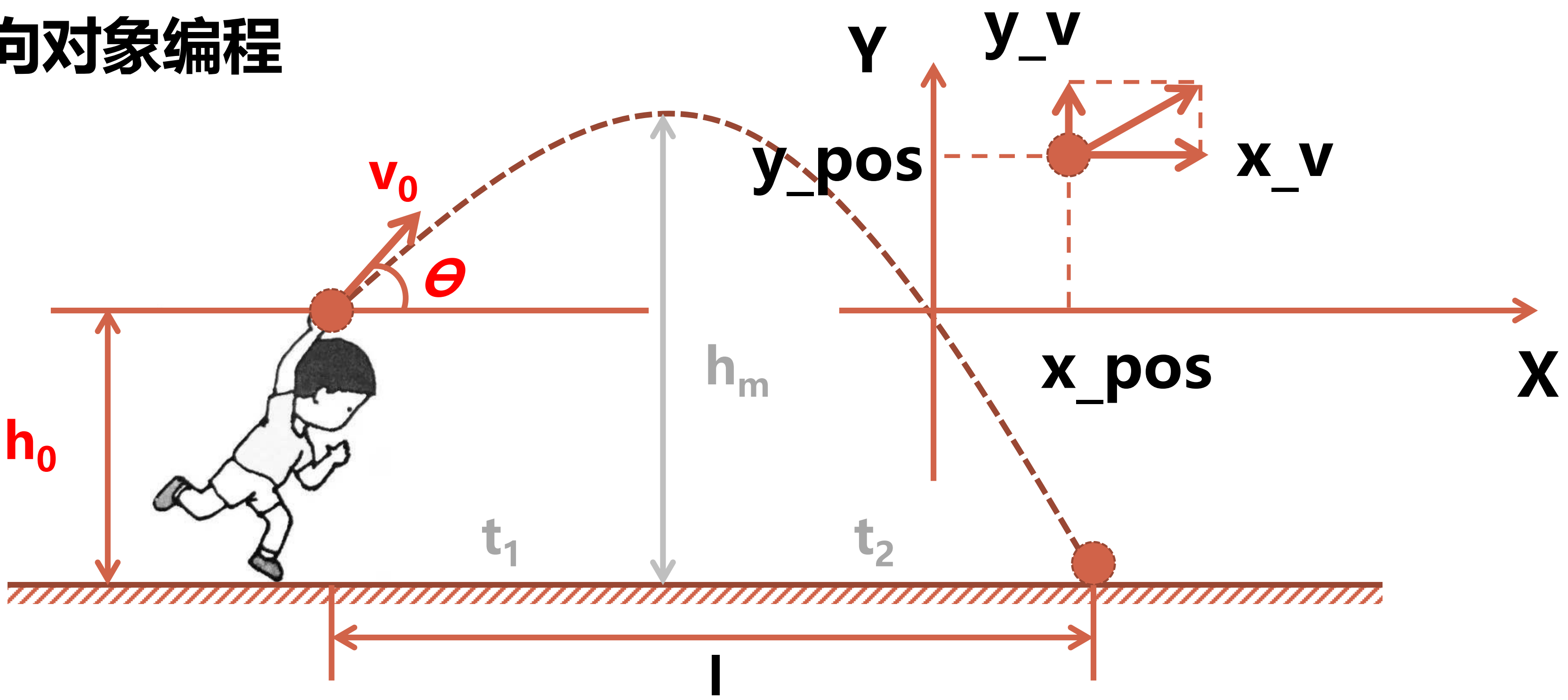
```
输入初始高度h0(m)：1.8
输入初始速度v0(m/s)：10
输入抛掷角度theta(角度值)：45
飞行距离为11.77米
```



```python
import math
t_interval, g = 0.001, 9.8 #时间间隔为0.001s, 重力加速度为9.8m/s2

# 输入参数
h0 = eval(input("输入初始高度h0(m): "))
v0 = eval(input("输入初始速度v0(m/s): "))
theta = eval(input("输入抛掷角度theta(角度值): "))
theta = float(theta)/180*math.pi

# 定义铅球的初始信息
x_pos, y_pos, x_v, y_v  = 0, h0, v0*math.cos(theta), v0*math.sin(theta)

# 位置和速度更新函数
def update(t_interval, x_pos, y_pos, x_v, y_v):
    x_pos = x_pos + x_v*t_interval
    y_v1 = y_v - t_interval*g #y_v1为时间间隔的终止速度
    y_pos = y_pos + t_interval*(y_v+y_v1)/2
    y_v = y_v1
    return x_pos, y_pos, x_v, y_v

while y_pos>=0:
    x_pos, y_pos, x_v, y_v = update(t_interval, x_pos, y_pos, x_v, y_v)
print("飞行距离为{:.2f}米".format(x_pos))
```

```
输入初始高度h0(m): 1.8
输入初始速度v0(m/s): 10
输入抛掷角度theta(角度值): 45
飞行距离为11.77米
```

面向对象编程



输入:

初始高度：h_0 （米）

初始速度：v_0 （米）

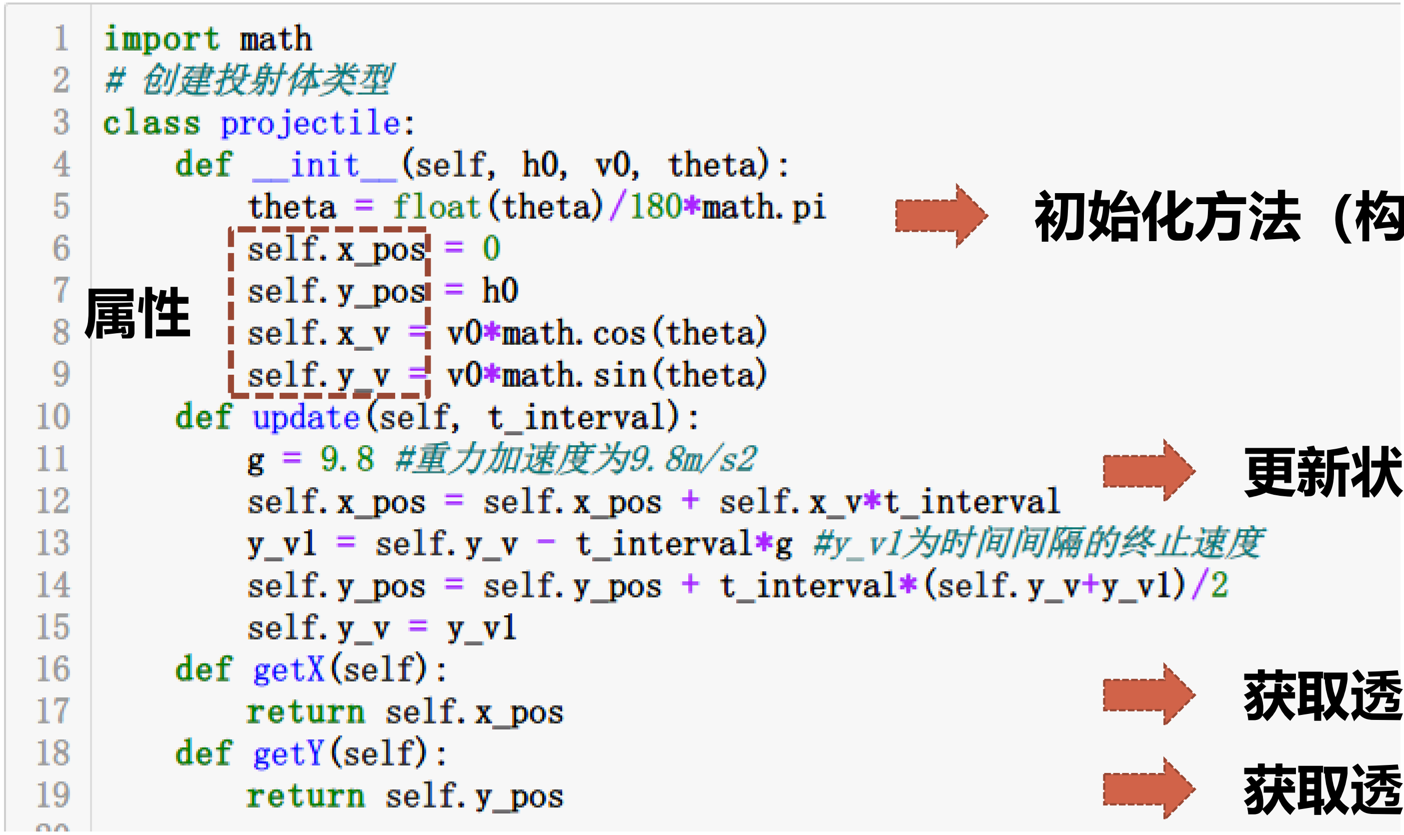
抛掷角度：theta （角度）

创建投射体类型

属性：x_pos, y_pos, x_v, y_v

方法：更新投射体状态，获取投射体高度，获取投射体距离

面向对象编程



```python
import math
# 创建投射体类型
class projectile:
    def __init__(self, h0, v0, theta):
        theta = float(theta)/180*math.pi
        self.x_pos = 0
        self.y_pos = h0
        self.x_v = v0*math.cos(theta)
        self.y_v = v0*math.sin(theta)
    def update(self, t_interval):
        g = 9.8 #重力加速度为9.8m/s2
        self.x_pos = self.x_pos + self.x_v*t_interval
        y_v1 = self.y_v - t_interval*g #y_v1为时间间隔的终止速度
        self.y_pos = self.y_pos + t_interval*(self.y_v+y_v1)/2
        self.y_v = y_v1
    def getX(self):
        return self.x_pos
    def getY(self):
        return self.y_pos
```

属性

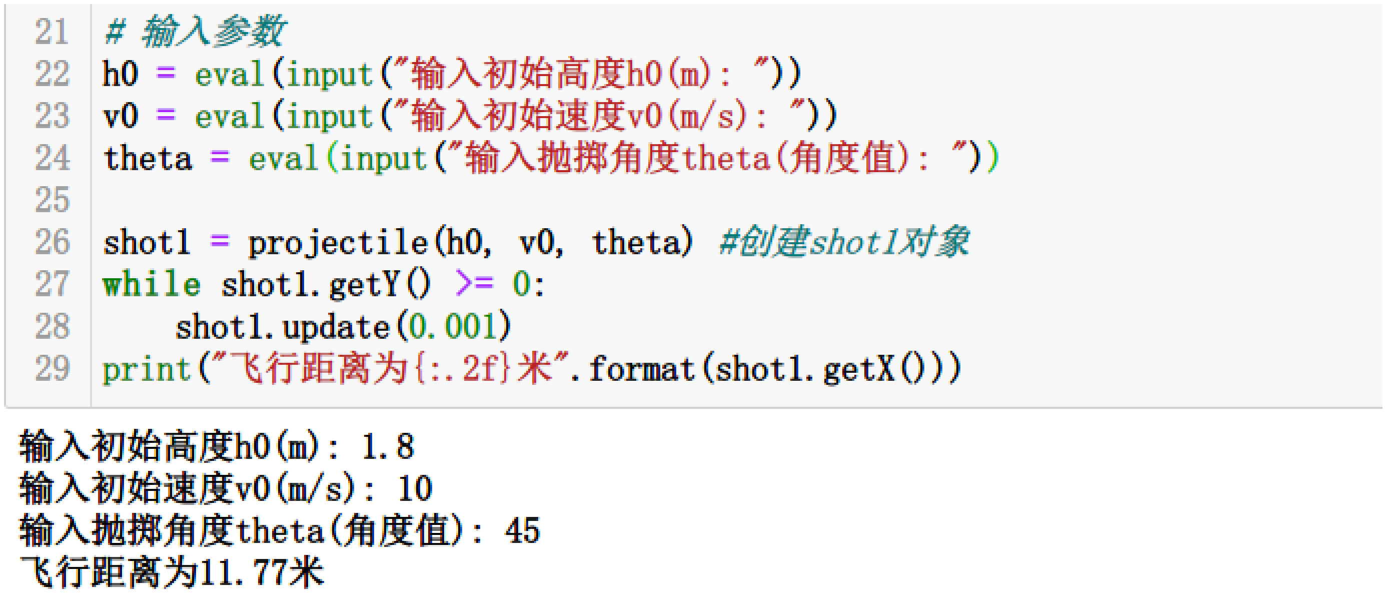
初始化方法（构造函数）

更新状态方法

获取透射体高度

获取透射体距离

面向对象编程



```python
# 输入参数
h0 = eval(input("输入初始高度h0(m): "))
v0 = eval(input("输入初始速度v0(m/s): "))
theta = eval(input("输入抛掷角度theta(角度值): "))

shot1 = projectile(h0, v0, theta) #创建shot1对象
while shot1.getY() >= 0:
    shot1.update(0.001)
print("飞行距离为{:.2f}米".format(shot1.getX()))
```

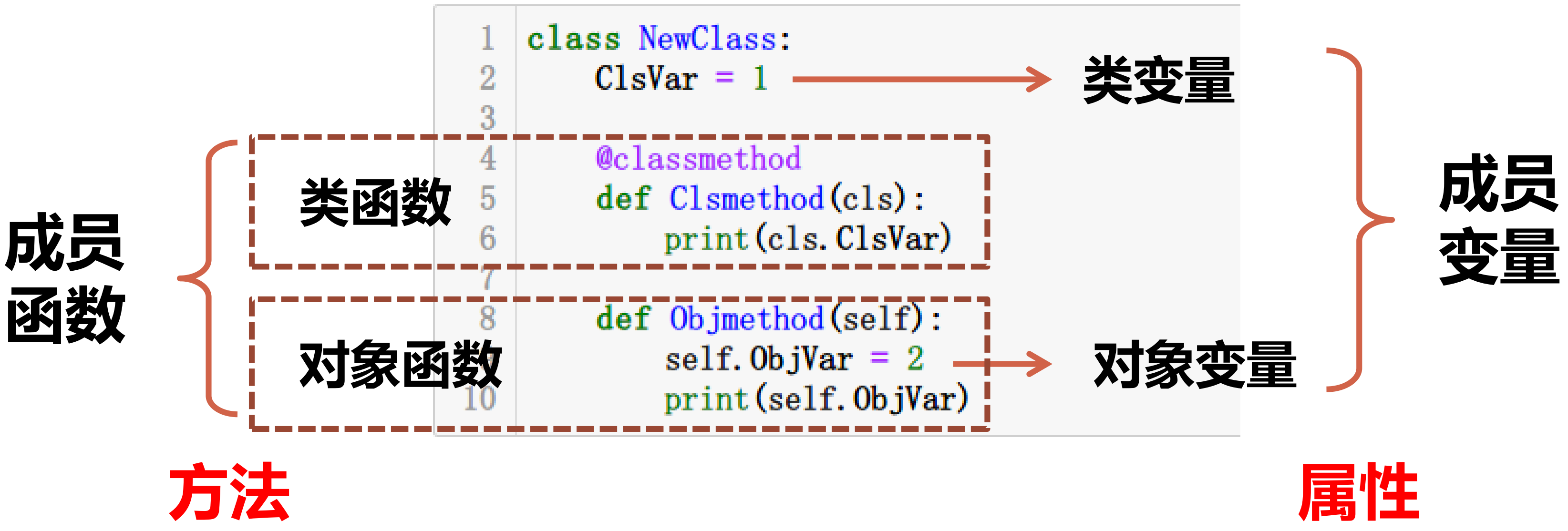
```
输入初始高度h0(m): 1.8
输入初始速度v0(m/s): 10
输入抛掷角度theta(角度值): 45
飞行距离为11.77米
```

相比于直接定义函数，更加抽象，只需要关注对象提供的方法

类和对象

类是用来描述具有相同属性和方法的对象的集合，它定义该集合中每个对象所共有的属性和方法，对象是类的实例

利用关键字class创建类，class与冒号之间为类的名字，注意缩进

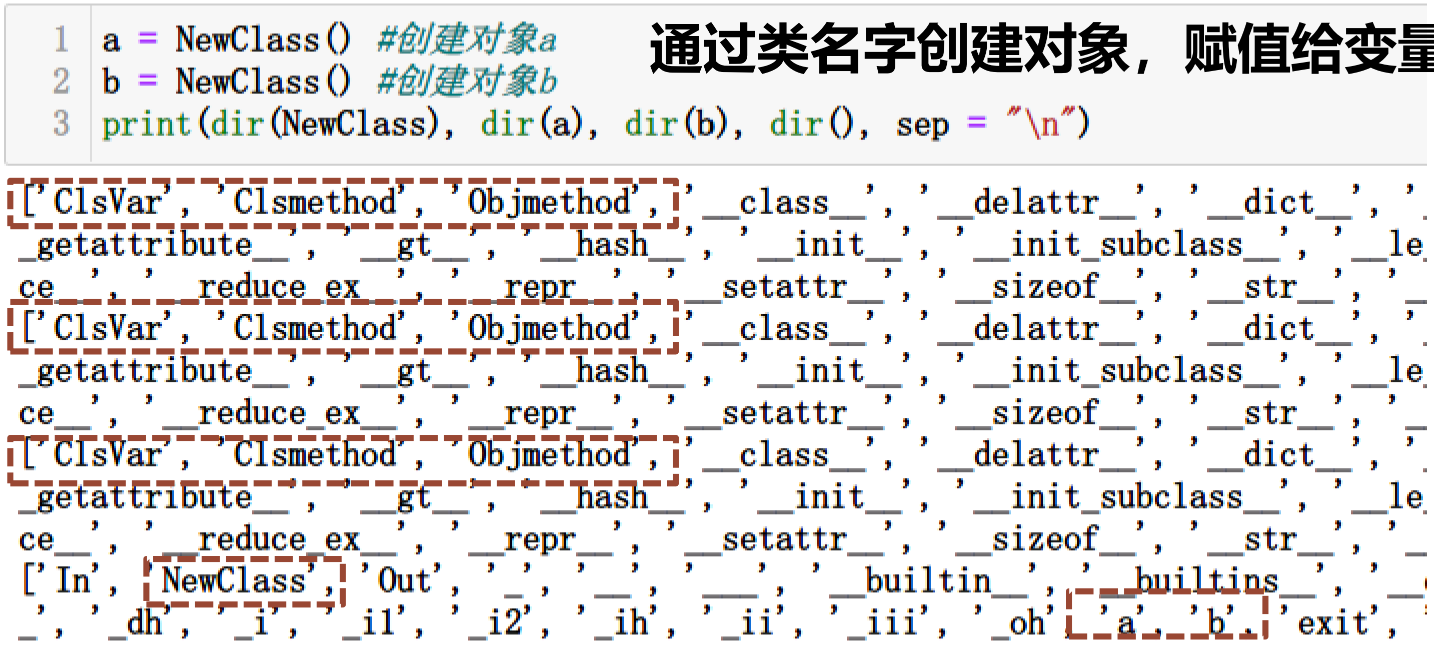


```
class NewClass:
    ClsVar = 1

    @classmethod
    def Clsmethod(cls):
        print(cls.ClsVar)

    def Objmethod(self):
        self.ObjVar = 2
        print(self.ObjVar)
```

用属性和方法与面向过程中的变量和函数区分开

类属性与对象属性

类属性定义在类中，是不属于某个具体对象的特征，被所有对象共同使用。对象属性定义在对象方法中，是以self为前缀的变量，没有该前缀的变量是普通的局部变量。



```
a = NewClass() #创建对象a
b = NewClass() #创建对象b
print(dir(NewClass), dir(a), dir(b), dir(), sep = "\n")
```

通过类名字创建对象，赋值给变量a和b

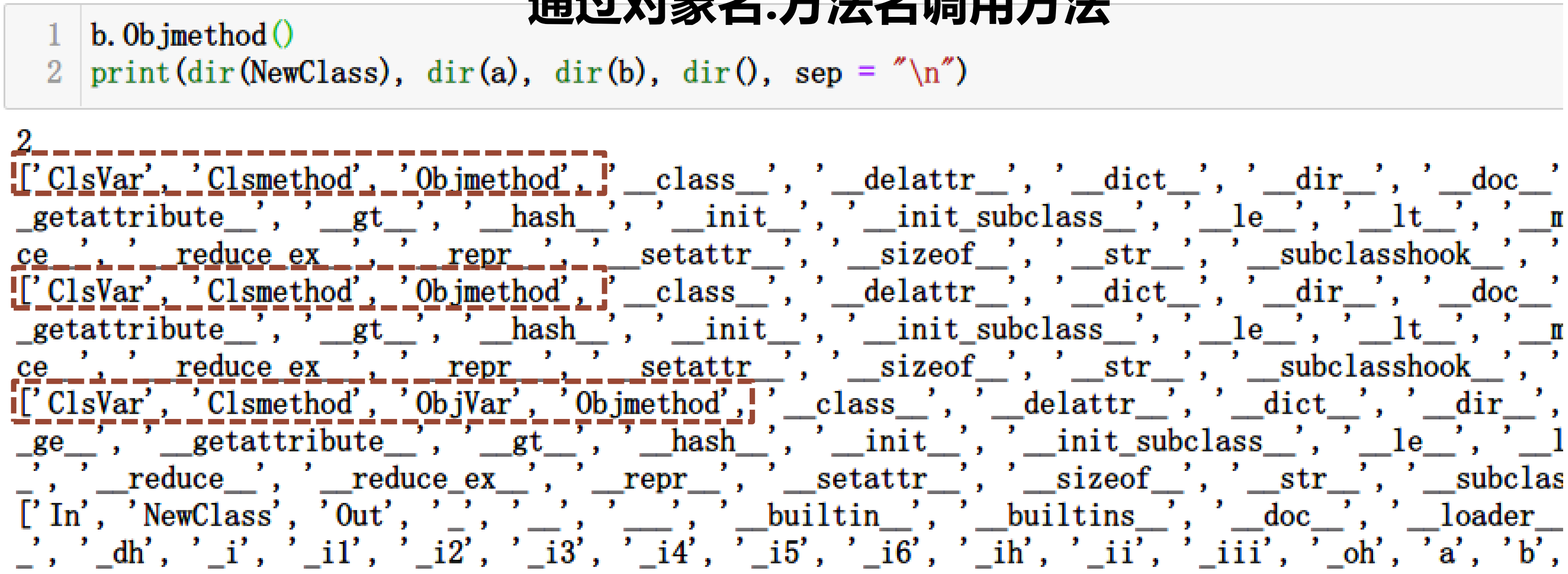
```
['ClsVar', 'Clsmethod', 'Objmethod', '__class__', '__delattr__', '__dict__', '_
_getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le
ce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '_
['ClsVar', 'Clsmethod', 'Objmethod', '__class__', '__delattr__', '__dict__', '_
_getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le
ce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '_
['ClsVar', 'Clsmethod', 'Objmethod', '__class__', '__delattr__', '__dict__', '_
_getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le
ce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '_
['In', 'NewClass', 'Out', '_', '__', '___', '__builtin__', '__builtins__', '__
_', '_dh', '_i', '_i1', '_i2', '_ih', '_ii', '_iii', '_oh', 'a', 'b', 'exit',
```

在顶层命名空间中产生NewClass，a和b对象名，三个对象命名空间中都存在ClsVar属性（类属性为缺省属性）

类属性与对象属性

调用对象b的Objmethod()方法后在b的命名空间中产生ObjVar属性，说明对象a和b的命名空间是相互独立的

通过对象名.方法名调用方法

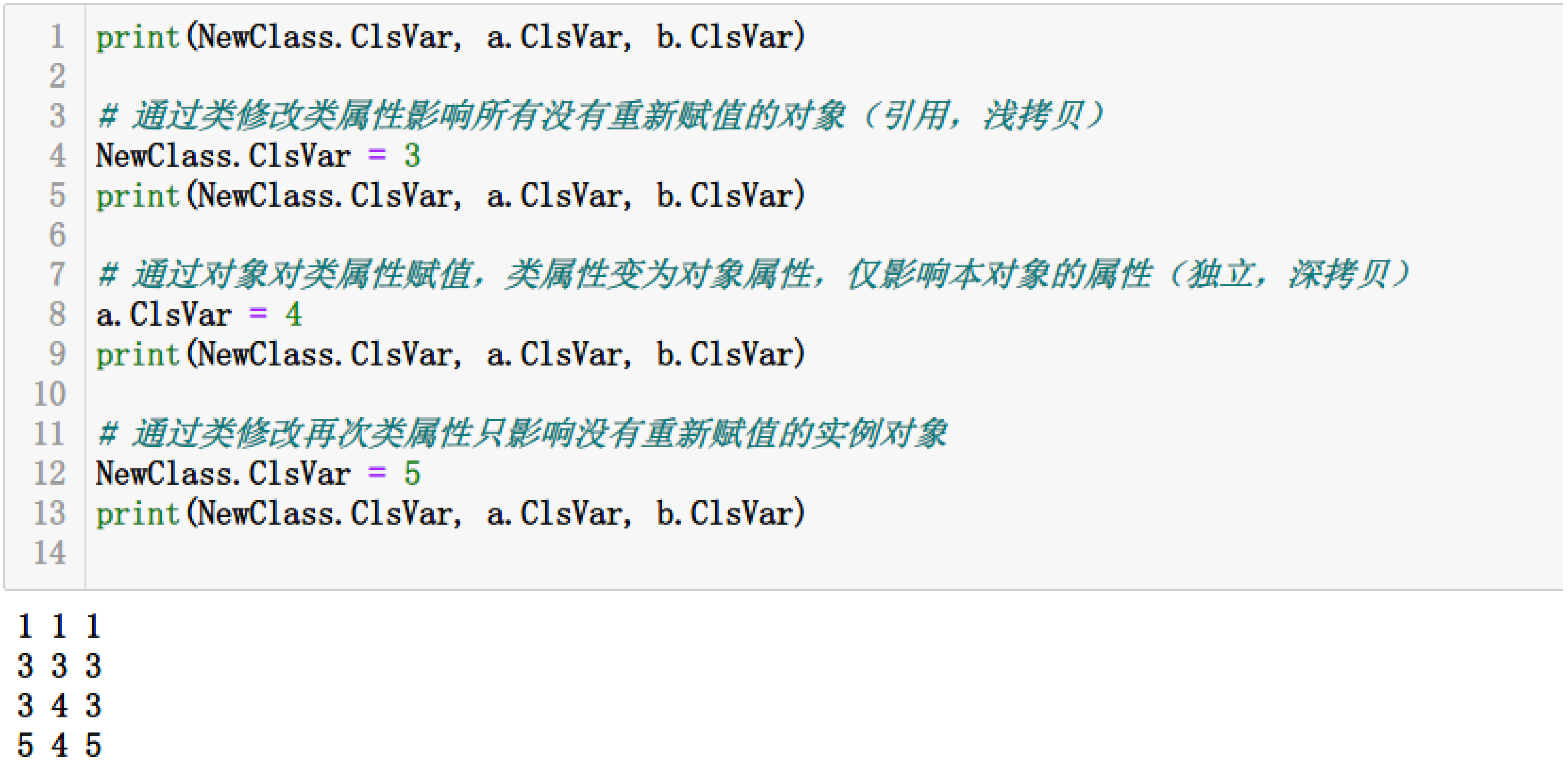


```
b.Objmethod()
print(dir(NewClass), dir(a), dir(b), dir(), sep = "\n")
```

```
2
['ClsVar', 'Clsmethod', 'Objmethod', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__'
_getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__m
ce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '
['ClsVar', 'Clsmethod', 'Objmethod', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__'
_getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__m
ce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '
['ClsVar', 'Clsmethod', 'ObjVar', 'Objmethod', '__class__', '__delattr__', '__dict__', '__dir__',
_ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__l
_', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclas
['In', 'NewClass', 'Out', '_', '__', '___', '__builtin__', '__builtins__', '__doc__', '__loader__
_', '_dh', '_i', '_i1', '_i2', '_i3', '_i4', '_i5', '_i6', '_ih', '_ii', '_iii', '_oh', 'a', 'b',
```

类属性与对象属性

通过类修改类属性影响所有没有重新赋值的对象，通过对象对类属性赋值，类属性变为对象属性，仅影响对象本身



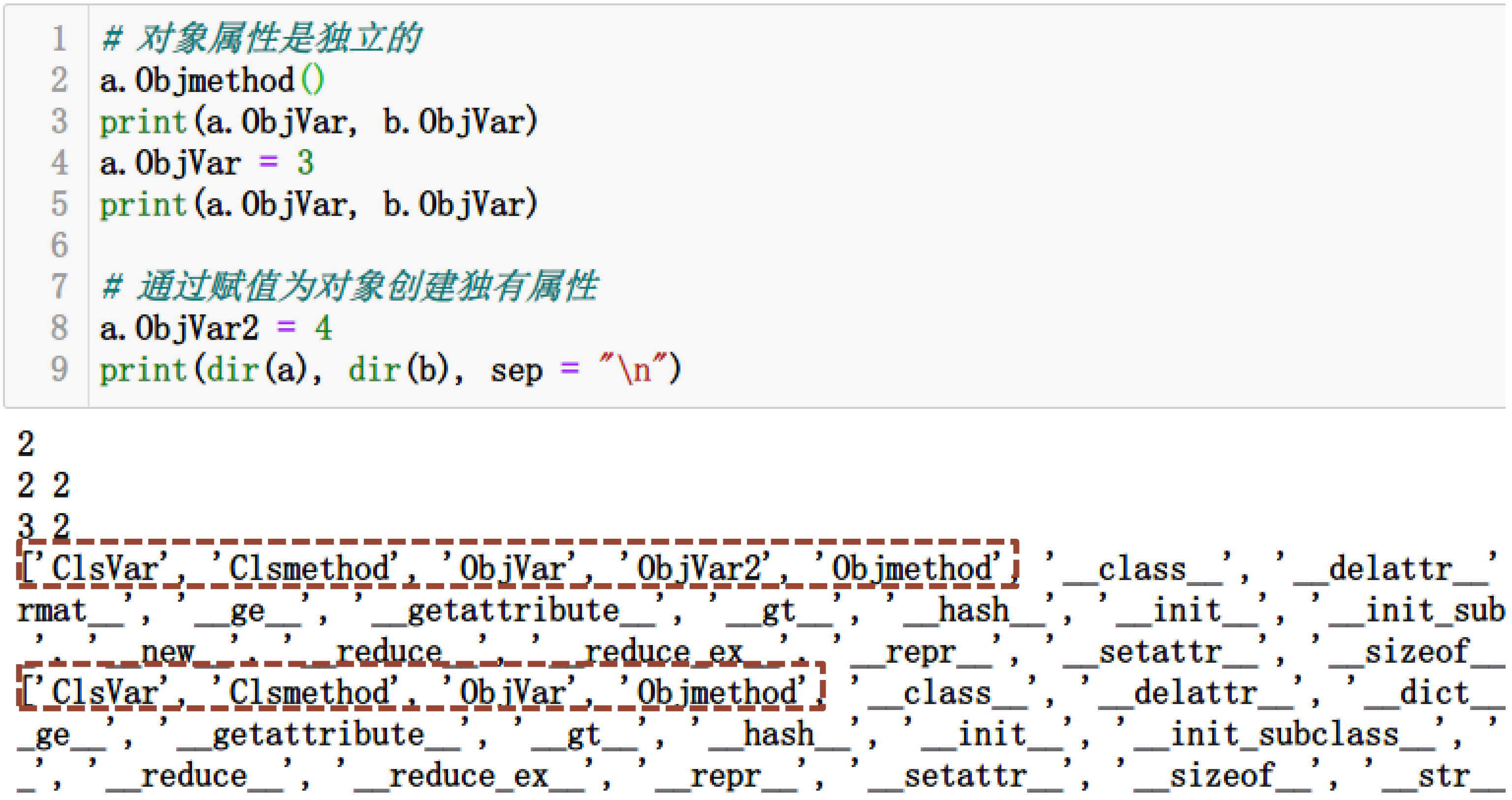
```
print(NewClass.ClsVar, a.ClsVar, b.ClsVar)

# 通过类修改类属性影响所有没有重新赋值的对象（引用，浅拷贝）
NewClass.ClsVar = 3
print(NewClass.ClsVar, a.ClsVar, b.ClsVar)

# 通过对象对类属性赋值，类属性变为对象属性，仅影响本对象的属性（独立，深拷贝）
a.ClsVar = 4
print(NewClass.ClsVar, a.ClsVar, b.ClsVar)

# 通过类修改再次类属性只影响没有重新赋值的实例对象
NewClass.ClsVar = 5
print(NewClass.ClsVar, a.ClsVar, b.ClsVar)

```

```
1 1 1
3 3 3
3 4 3
5 4 5
```

推荐使用类名取值访问和修改类属性！！！

类属性与对象属性

通过对象对对象属性赋值，如果变量名相同直接修改原有属性，如果变量名不同创建对象独有属性

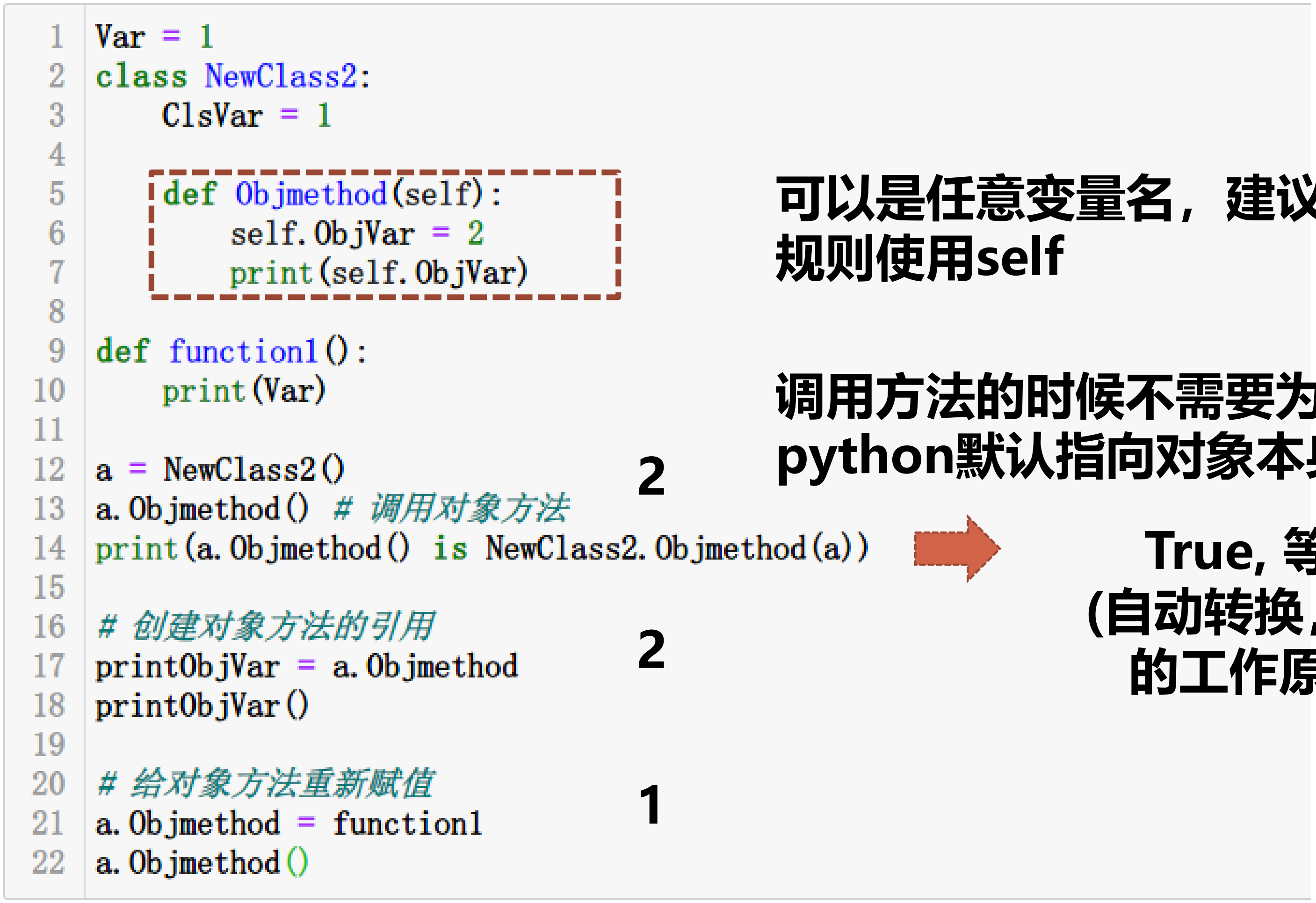


```python
# 对象属性是独立的
a.Objmethod()
print(a.ObjVar, b.ObjVar)
a.ObjVar = 3
print(a.ObjVar, b.ObjVar)

# 通过赋值为对象创建独有属性
a.ObjVar2 = 4
print(dir(a), dir(b), sep = "\n")
```

```
2
2 2
3 2
['ClsVar', 'Clsmethod', 'ObjVar', 'ObjVar2', 'Objmethod', '__class__', '__delattr__'
rmat__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_sub
', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__
['ClsVar', 'Clsmethod', 'ObjVar', 'Objmethod', '__class__', '__delattr__', '__dict__
_ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '
_', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__
```

方法与self

方法一般指对象（实例）方法，与普通的函数只有一个区别，必须有一个额外的第一个参数（self）



```python
Var = 1
class NewClass2:
    ClsVar = 1

    def Objmethod(self):
        self.ObjVar = 2
        print(self.ObjVar)

def function1():
    print(Var)

a = NewClass2()
a.Objmethod()  # 调用对象方法
print(a.Objmethod() is NewClass2.Objmethod(a))

# 创建对象方法的引用
printObjVar = a.Objmethod
printObjVar()

# 给对象方法重新赋值
a.Objmethod = function1
a.Objmethod()
```

2

2

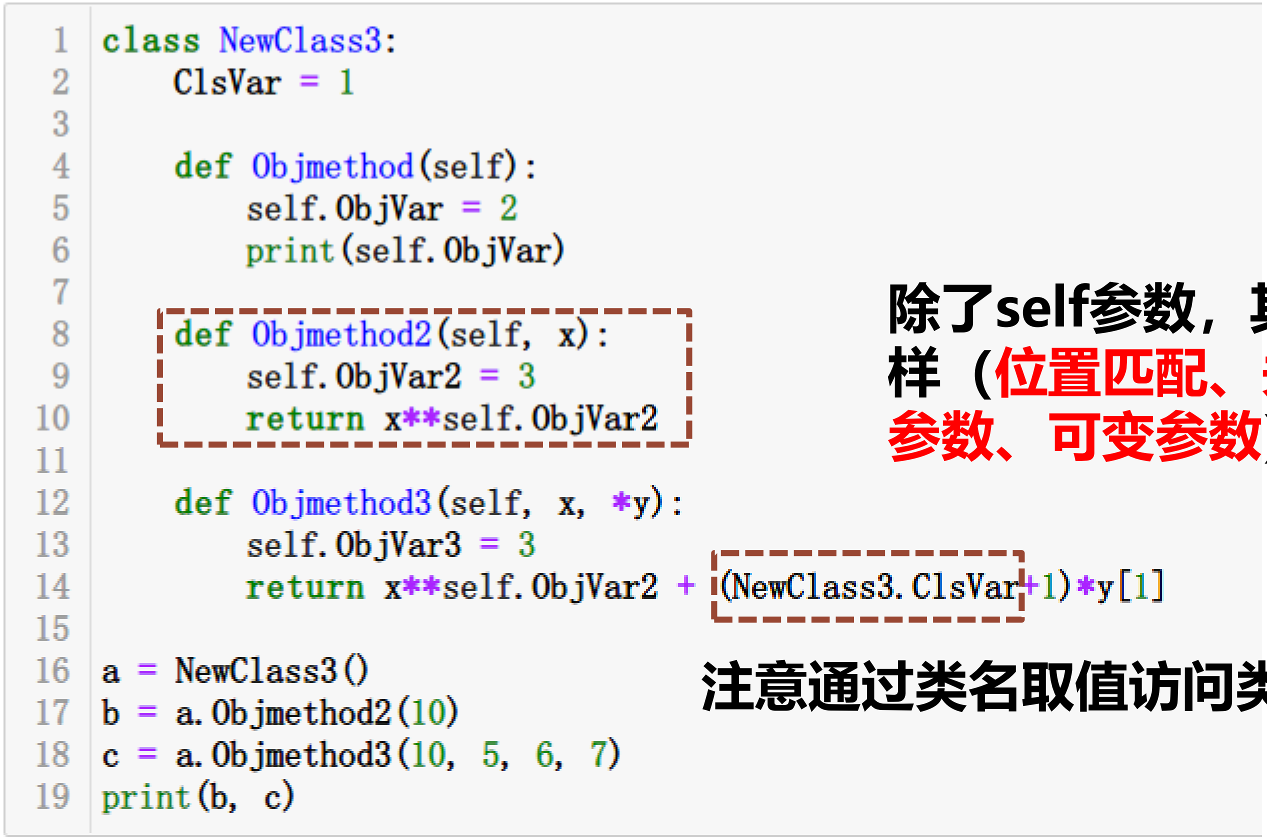
1

可以是任意变量名，建议遵循通用规则使用self

调用方法的时候不需要为self赋值，python默认指向对象本身

True, 等价
(自动转换，self的工作原理)

多参数方法



```
class NewClass3:
    ClsVar = 1

    def Objmethod(self):
        self.ObjVar = 2
        print(self.ObjVar)

    def Objmethod2(self, x):
        self.ObjVar2 = 3
        return x**self.ObjVar2

    def Objmethod3(self, x, *y):
        self.ObjVar3 = 3
        return x**self.ObjVar2 + (NewClass3.ClsVar+1)*y[1]

a = NewClass3()
b = a.Objmethod2(10)
c = a.Objmethod3(10, 5, 6, 7)
print(b, c)
```

```
1000 1012
```

除了self参数，其他参数与函数一样（位置匹配、关键字匹配、缺省参数、可变参数）

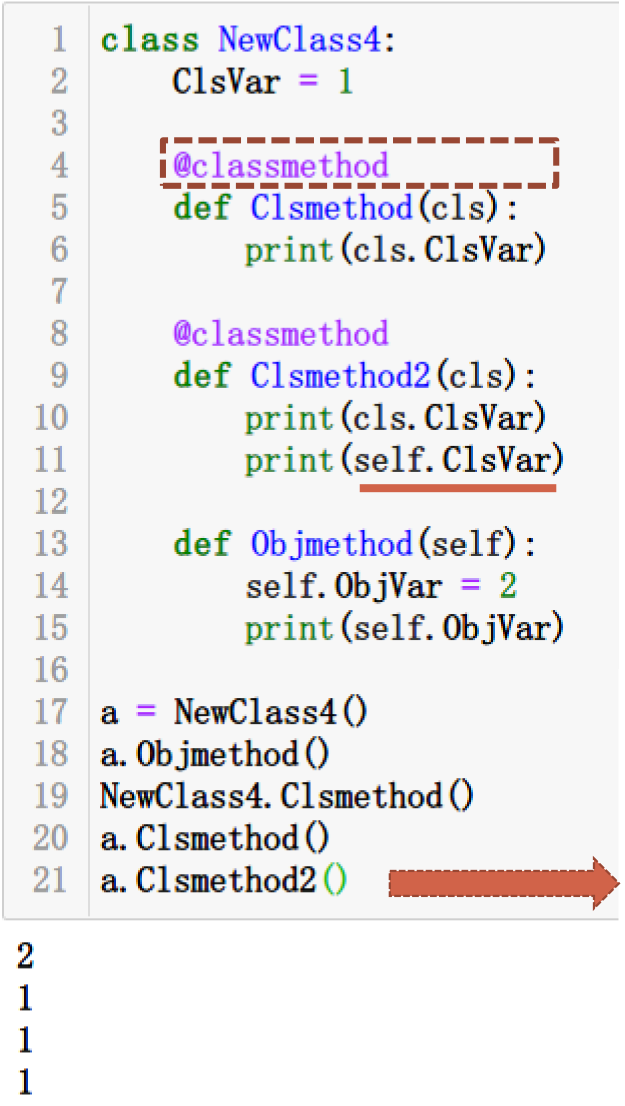
注意通过类名取值访问类属性

```
def Objmethod2(self, x):
    self.ObjVar2 = 3
    return x**self.ObjVar
```

调用a.Objmethod2(10) ???

类方法

类方法是不属于某个具体对象的行为，被所有对象共同使用



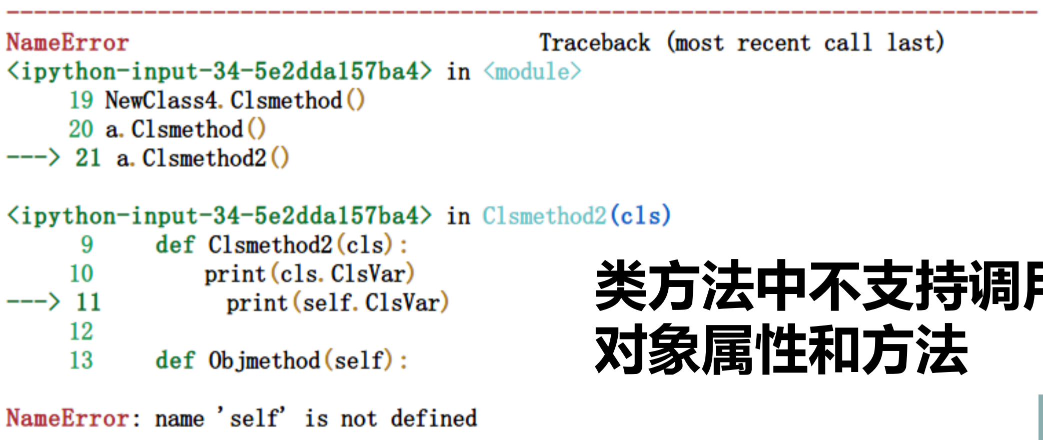
```
class NewClass4:
    ClsVar = 1

    @classmethod
    def Clsmethod(cls):
        print(cls.ClsVar)

    @classmethod
    def Clsmethod2(cls):
        print(cls.ClsVar)
        print(self.ClsVar)

    def Objmethod(self):
        self.ObjVar = 2
        print(self.ObjVar)

a = NewClass4()
a.Objmethod()
NewClass4.Clsmethod()
a.Clsmethod()
a.Clsmethod2()
```

```
2
1
1
1
```

装饰器@classmethod声明为类方法
第一个参数为cls，调用方式为：

类名.类方法名()

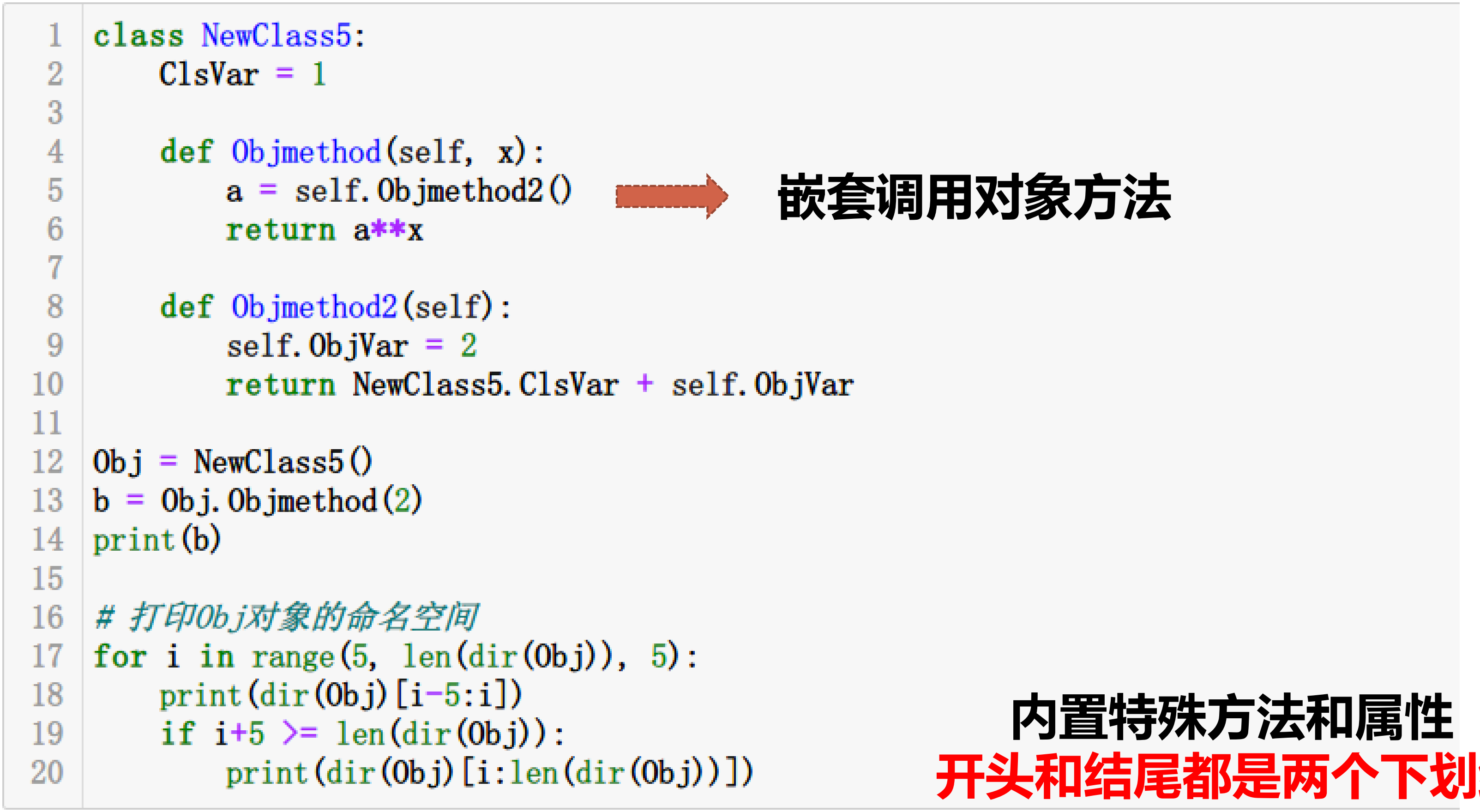
对象名.类方法名()



```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-34-5e2dda157ba4> in <module>
     19 NewClass4.Clsmethod()
     20 a.Clsmethod()
---> 21 a.Clsmethod2()

<ipython-input-34-5e2dda157ba4> in Clsmethod2(cls)
      9     def Clsmethod2(cls):
     10         print(cls.ClsVar)
---> 11         print(self.ClsVar)
     12
     13     def Objmethod(self):

NameError: name 'self' is not defined
```

类方法中不支持调用对象属性和方法

类与对象示例

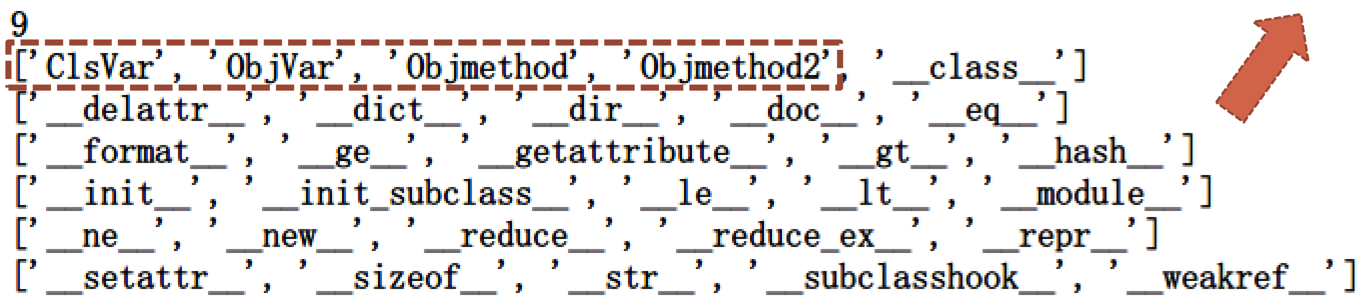


```python
class NewClass5:
    ClsVar = 1

    def Objmethod(self, x):
        a = self.Objmethod2()
        return a**x

    def Objmethod2(self):
        self.ObjVar = 2
        return NewClass5.ClsVar + self.ObjVar

Obj = NewClass5()
b = Obj.Objmethod(2)
print(b)

# 打印Obj对象的命名空间
for i in range(5, len(dir(Obj)), 5):
    print(dir(Obj)[i-5:i])
    if i+5 >= len(dir(Obj)):
        print(dir(Obj)[i:len(dir(Obj))])
```

嵌套调用对象方法

内置特殊方法和属性

开头和结尾都是两个下划线



```
9
['ClsVar', 'ObjVar', 'Objmethod', 'Objmethod2', '__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__']
```

特殊方法

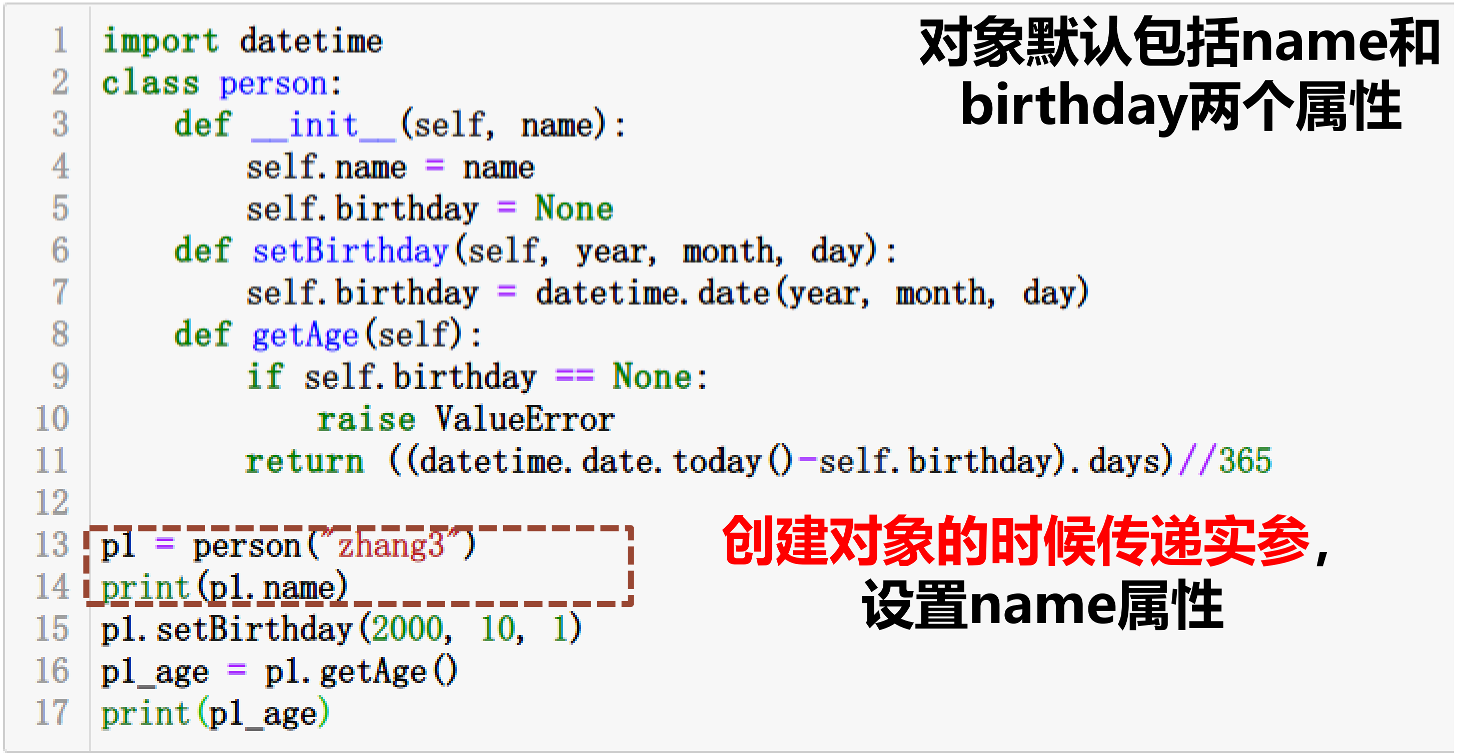
Python类定义了一些专用的方法，这些专用方法丰富了程序设计的功能，用于不同的应用场合

内置方法	描述
__init__(self, ...)	初始化对象，在创建对象是调用
__del__(self)	释放对象，在对象被删除时调用
__str__(self)	生成对象的字符串表示，使用print时被调用
__repr__(self)	生成对象的官方表示，使用print时被调用
__getattr__(self, name)	获取属性的值
__setattr__(self, name, val)	设置属性的值(val)
__delattr__(self, name)	删除name属性
__gt__(self, other)	判断self对象是否大于other对象
__lt__(self, other)	判断self对象是否小于other对象
__ge__(self, other)	判断self对象是否大于或等于other对象
__le__(self, other)	判断self对象是否小于或等于other对象
__eq__(self, other)	判断self对象是否等于other对象

__init__方法（构造函数）

创建对象时执行，用于初始化对象属性，第一个参数为self。如果有形参，需要在创建对象时传递实参

创建一个人的信息的类



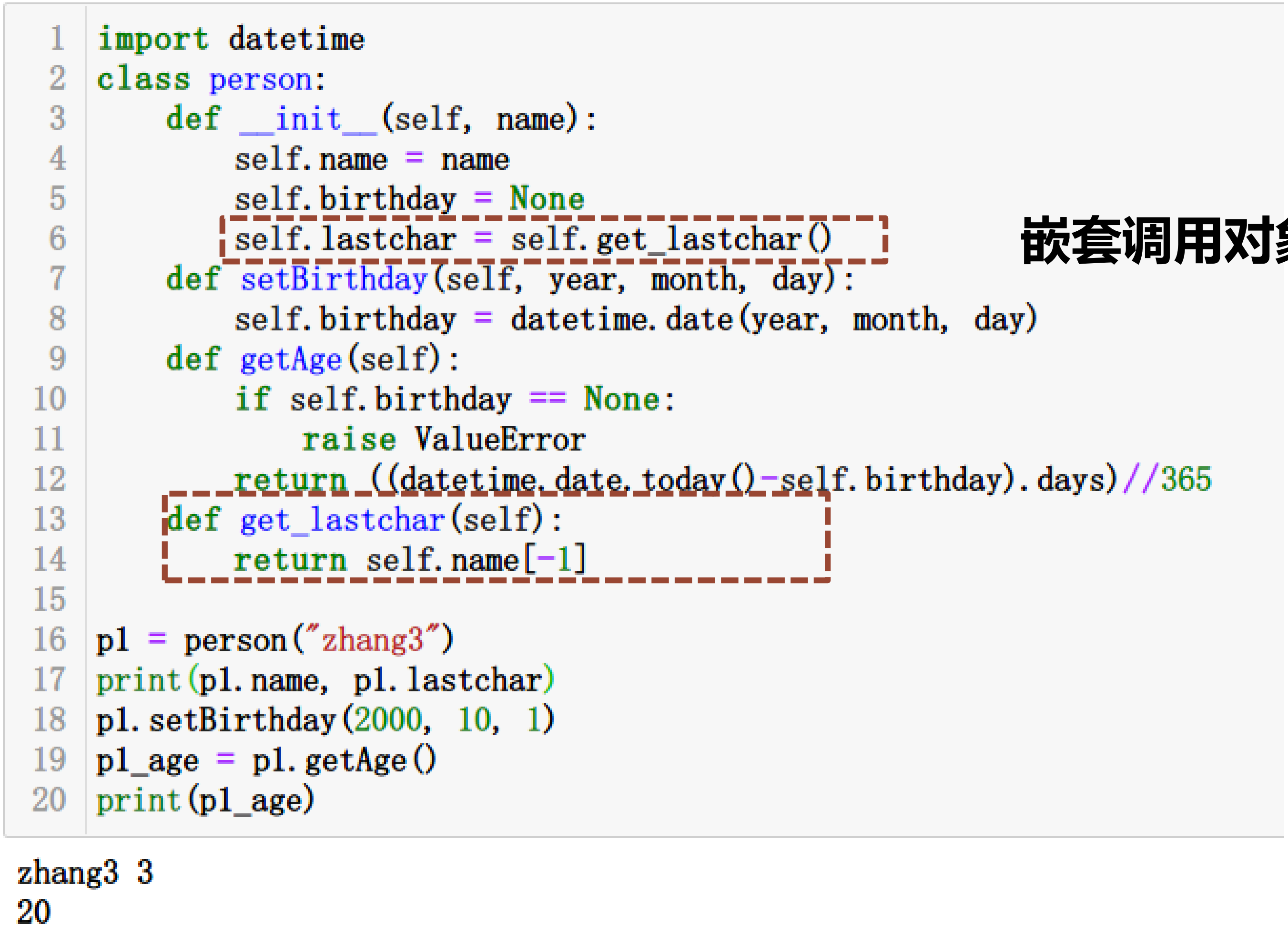
```python
import datetime
class person:
    def __init__(self, name):
        self.name = name
        self.birthday = None
    def setBirthday(self, year, month, day):
        self.birthday = datetime.date(year, month, day)
    def getAge(self):
        if self.birthday == None:
            raise ValueError
        return ((datetime.date.today()-self.birthday).days)//365

p1 = person("zhang3")
print(p1.name)
p1.setBirthday(2000, 10, 1)
p1_age = p1.getAge()
print(p1_age)
```

对象默认包括name和birthday两个属性

创建对象的时候传递实参，设置name属性

```
zhang3
20
```

__init__方法（构造函数）

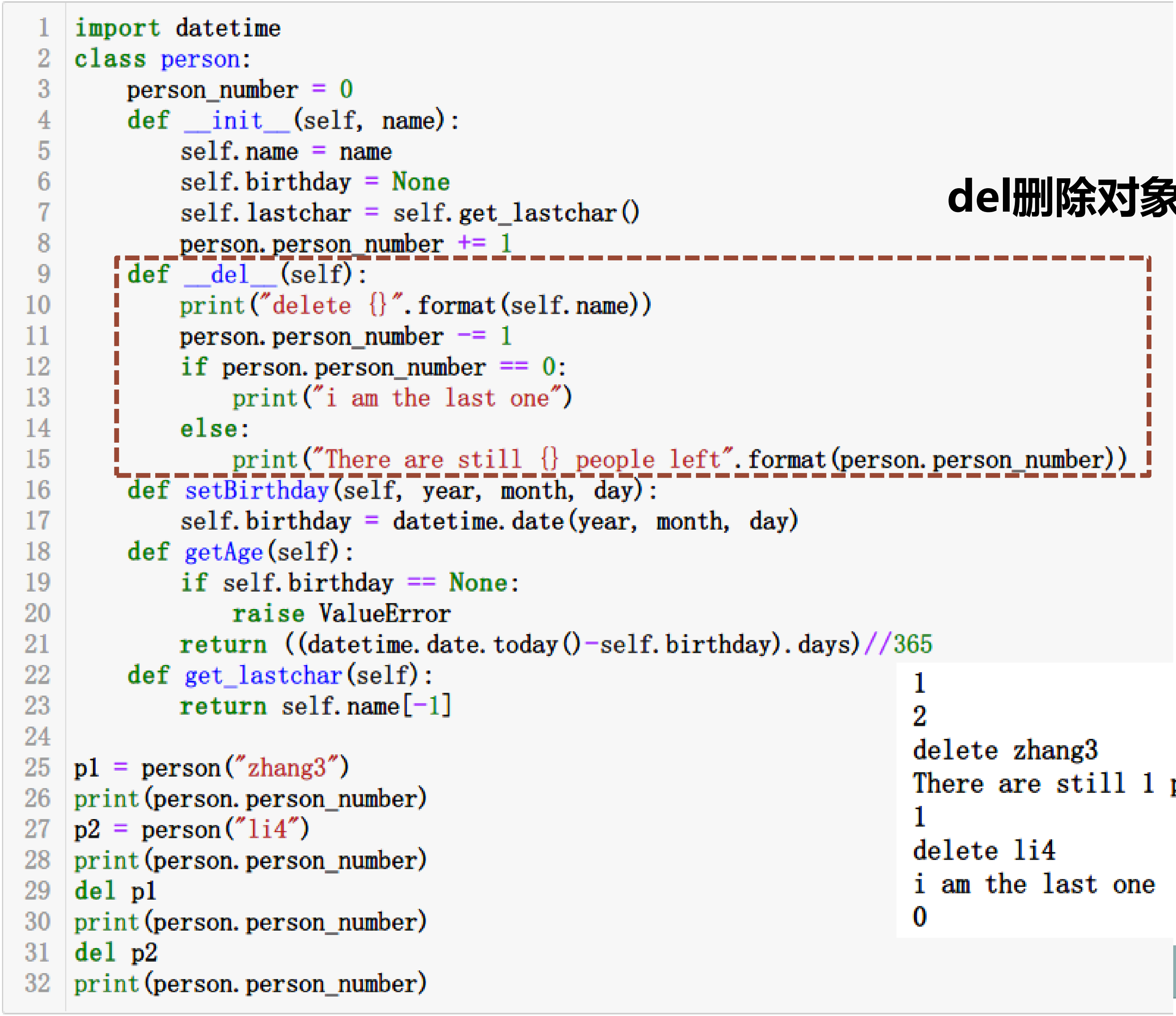


```python
import datetime
class person:
    def __init__(self, name):
        self.name = name
        self.birthday = None
        self.lastchar = self.get_lastchar()
    def setBirthday(self, year, month, day):
        self.birthday = datetime.date(year, month, day)
    def getAge(self):
        if self.birthday == None:
            raise ValueError
        return ((datetime.date.today()-self.birthday).days)//365
    def get_lastchar(self):
        return self.name[-1]

p1 = person("zhang3")
print(p1.name, p1.lastchar)
p1.setBirthday(2000, 10, 1)
p1_age = p1.getAge()
print(p1_age)
```

```
zhang3 3
20
```

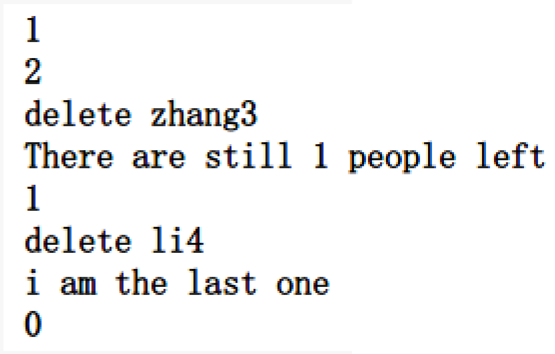
嵌套调用对象方法

__del__方法（析构函数）



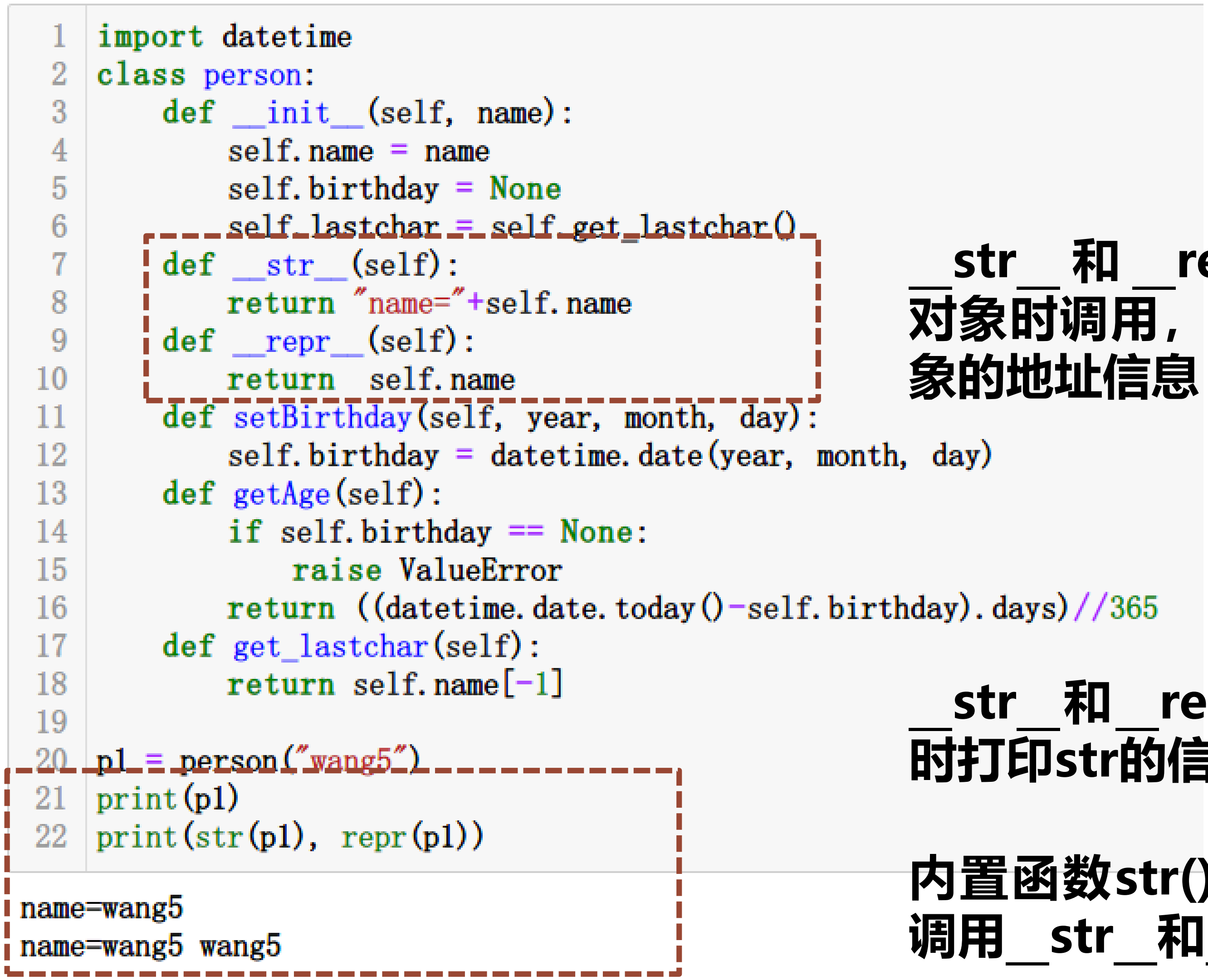
```python
import datetime
class person:
    person_number = 0
    def __init__(self, name):
        self.name = name
        self.birthday = None
        self.lastchar = self.get_lastchar()
        person.person_number += 1
    def __del__(self):
        print("delete {}".format(self.name))
        person.person_number -= 1
        if person.person_number == 0:
            print("i am the last one")
        else:
            print("There are still {} people left".format(person.person_number))
    def setBirthday(self, year, month, day):
        self.birthday = datetime.date(year, month, day)
    def getAge(self):
        if self.birthday == None:
            raise ValueError
        return ((datetime.date.today()-self.birthday).days)//365
    def get_lastchar(self):
        return self.name[-1]

p1 = person("zhang3")
print(person.person_number)
p2 = person("li4")
print(person.person_number)
del p1
print(person.person_number)
del p2
print(person.person_number)
```

del删除对象时调用



```
1
2
delete zhang3
There are still 1 people left
1
delete li4
i am the last one
0
```

__str__和__repr__方法



```
import datetime
class person:
    def __init__(self, name):
        self.name = name
        self.birthday = None
        self.lastchar = self.get_lastchar()
    def __str__(self):
        return "name="+self.name
    def __repr__(self):
        return  self.name
    def setBirthday(self, year, month, day):
        self.birthday = datetime.date(year, month, day)
    def getAge(self):
        if self.birthday == None:
            raise ValueError
        return ((datetime.date.today()-self.birthday).days)//365
    def get_lastchar(self):
        return self.name[-1]

p1 = person("wang5")
print(p1)
print(str(p1), repr(p1))
```

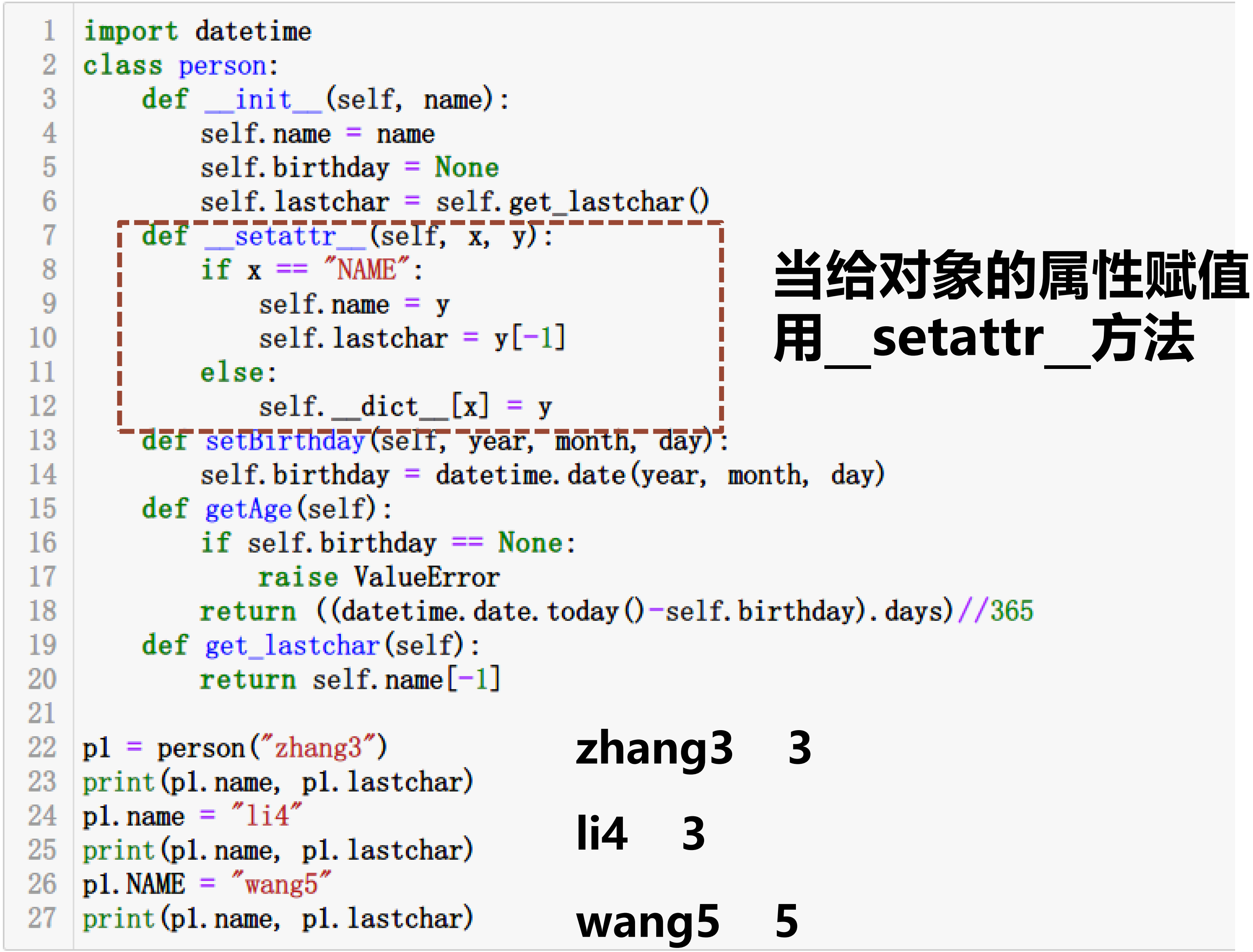
```
name=wang5
name=wang5 wang5
```

__str__和__repr__在print对象时调用，默认是打印对象的地址信息

__str__和__repr__同时存在时打印str的信息

内置函数str()和repr()分别调用__str__和__repr__

__setattr__方法

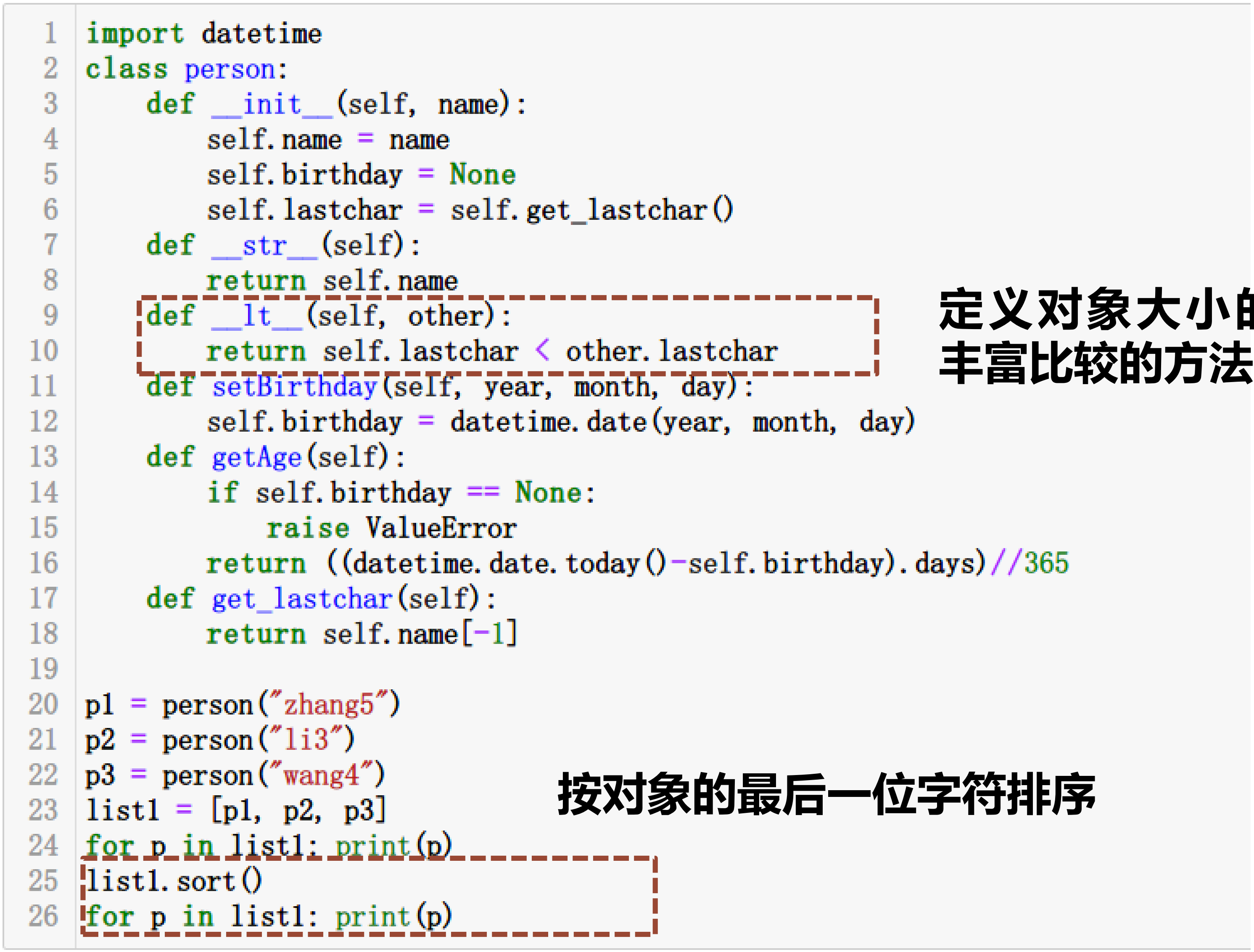


```python
import datetime
class person:
    def __init__(self, name):
        self.name = name
        self.birthday = None
        self.lastchar = self.get_lastchar()
    def __setattr__(self, x, y):
        if x == "NAME":
            self.name = y
            self.lastchar = y[-1]
        else:
            self.__dict__[x] = y
    def setBirthday(self, year, month, day):
        self.birthday = datetime.date(year, month, day)
    def getAge(self):
        if self.birthday == None:
            raise ValueError
        return ((datetime.date.today()-self.birthday).days)//365
    def get_lastchar(self):
        return self.name[-1]

p1 = person("zhang3")
print(p1.name, p1.lastchar)
p1.name = "li4"
print(p1.name, p1.lastchar)
p1.NAME = "wang5"
print(p1.name, p1.lastchar)
```

```
zhang3   3
li4   3
wang5   5
```

当给对象的属性赋值时，调用__setattr__方法

__lt__方法



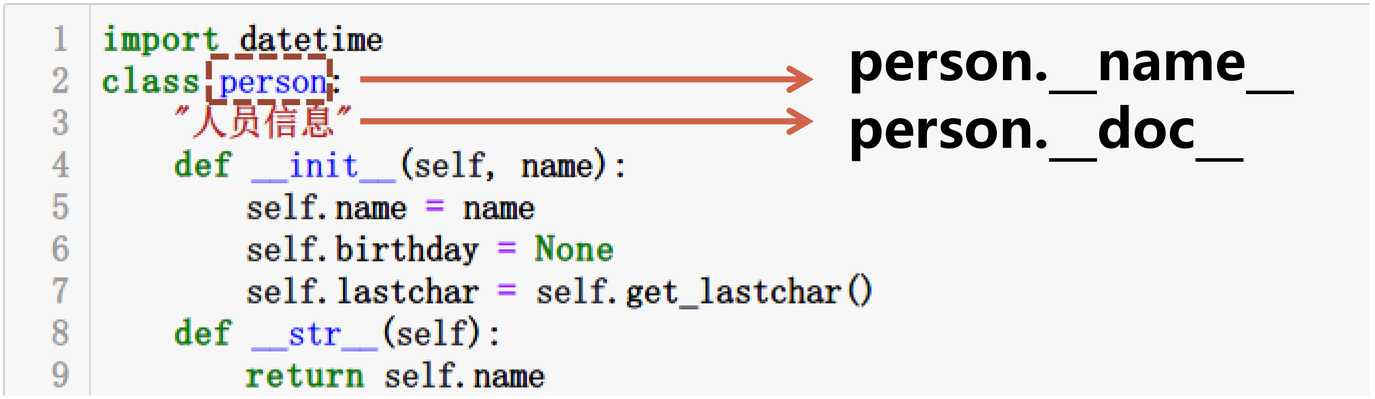
```python
import datetime
class person:
    def __init__(self, name):
        self.name = name
        self.birthday = None
        self.lastchar = self.get_lastchar()
    def __str__(self):
        return self.name
    def __lt__(self, other):
        return self.lastchar < other.lastchar
    def setBirthday(self, year, month, day):
        self.birthday = datetime.date(year, month, day)
    def getAge(self):
        if self.birthday == None:
            raise ValueError
        return ((datetime.date.today()-self.birthday).days)//365
    def get_lastchar(self):
        return self.name[-1]

p1 = person("zhang5")
p2 = person("li3")
p3 = person("wang4")
list1 = [p1, p2, p3]
for p in list1: print(p)
list1.sort()
for p in list1: print(p)
```

定义对象大小的规则，丰富比较的方法

按对象的最后一位字符排序

特殊类属性

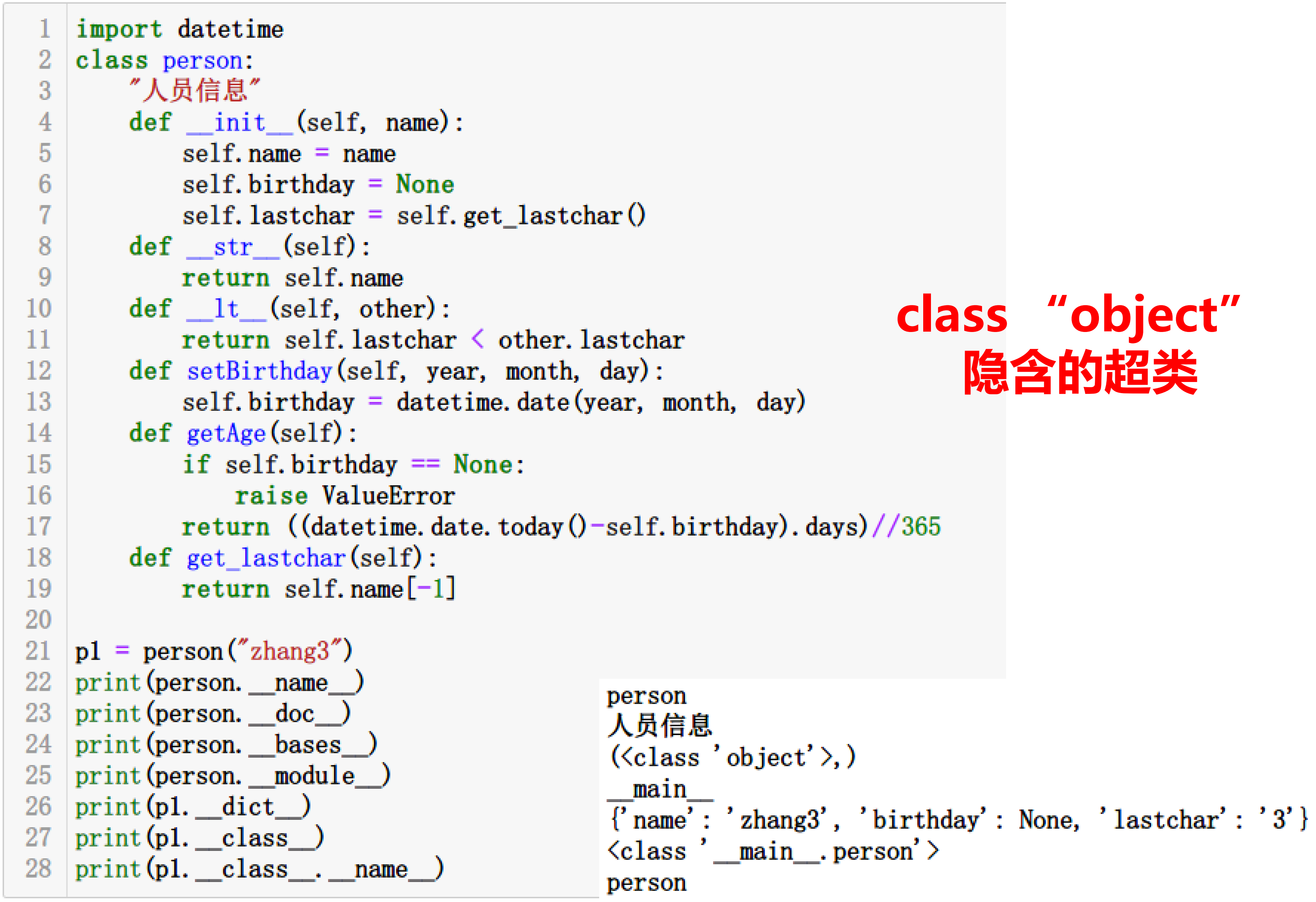
特殊类属性	描述
__name__	类的名字（字符串）
__doc__	类的文档字符串
__bases__	类的所有父类构成的元组
__module__	类定义所在的模块
__dict__	属性构成的字典
__class__	实例对应的类



```
import datetime
class person:
    ”人员信息”
    def __init__(self, name):
        self.name = name
        self.birthday = None
        self.lastchar = self.get_lastchar()
    def __str__(self):
        return self.name
```

person.__name__

person.__doc__



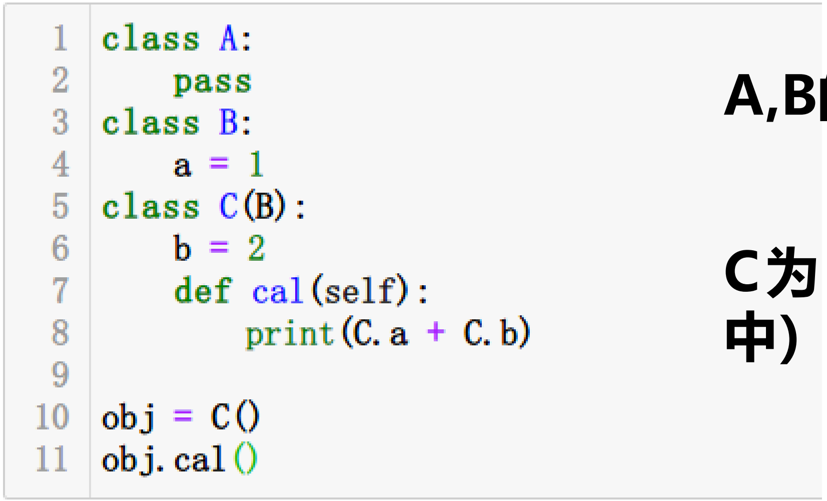
```python
import datetime
class person:
    "人员信息"
    def __init__(self, name):
        self.name = name
        self.birthday = None
        self.lastchar = self.get_lastchar()
    def __str__(self):
        return self.name
    def __lt__(self, other):
        return self.lastchar < other.lastchar
    def setBirthday(self, year, month, day):
        self.birthday = datetime.date(year, month, day)
    def getAge(self):
        if self.birthday == None:
            raise ValueError
        return ((datetime.date.today()-self.birthday).days)//365
    def get_lastchar(self):
        return self.name[-1]

p1 = person("zhang3")
print(person.__name__)
print(person.__doc__)
print(person.__bases__)
print(person.__module__)
print(p1.__dict__)
print(p1.__class__)
print(p1.__class__.__name__)
```

```
person
人员信息
(<class 'object'>,)
__main__
{'name': 'zhang3', 'birthday': None, 'lastchar': '3'}
<class '__main__.person'>
person
```

class “object”
隐含的超类

继承

通过继承创建新类，称为子类（派生类），原始的类称为父类（超类）。在子类中添加或修改变量和方法，实现代码重用



```python
class A:
    pass
class B:
    a = 1
class C(B):
    b = 2
    def cal(self):
        print(C.a + C.b)

obj = C()
obj.cal()
```

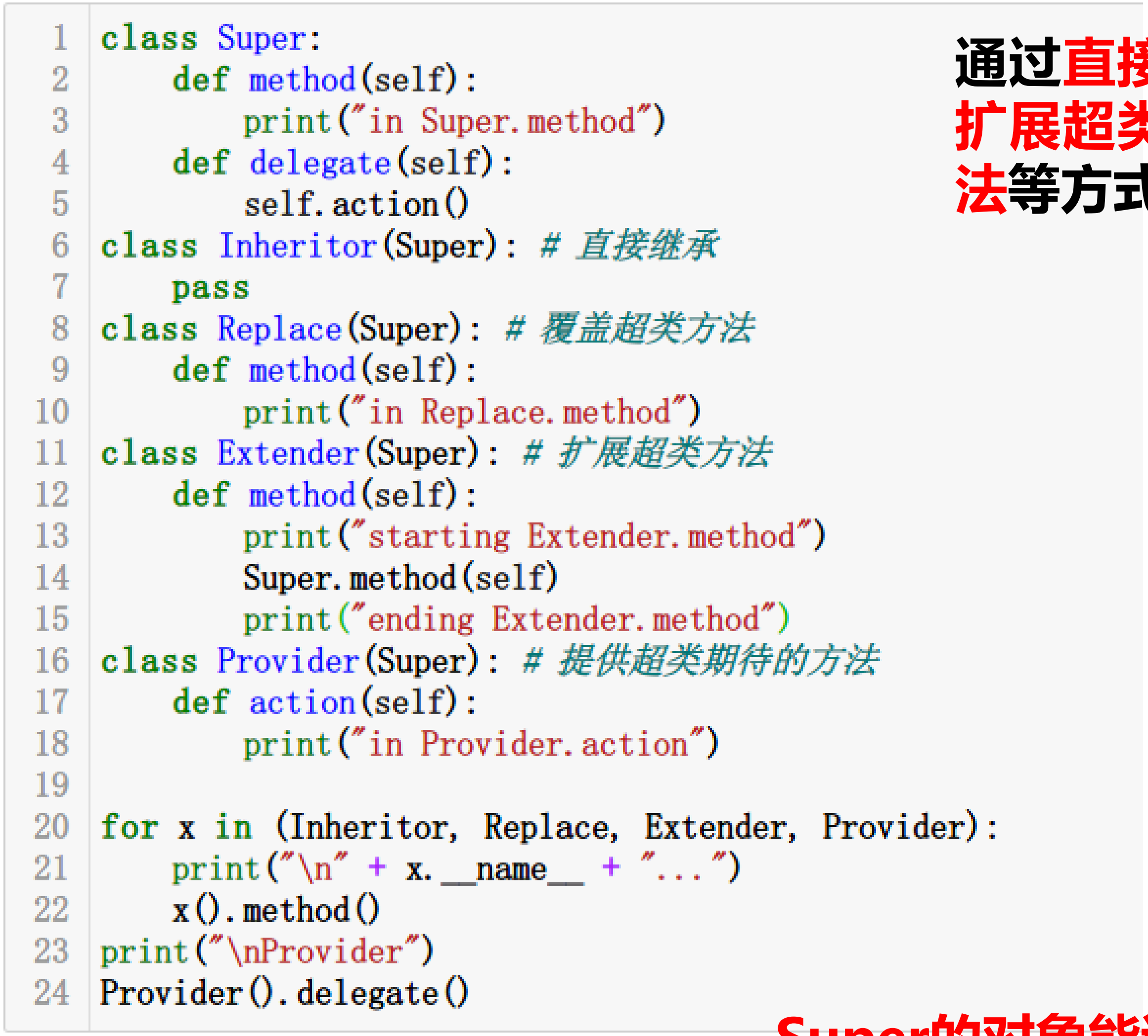
```
3
```

A,B的超类为object

C为B的子类（超类列在子类括号中），继承了类属性b

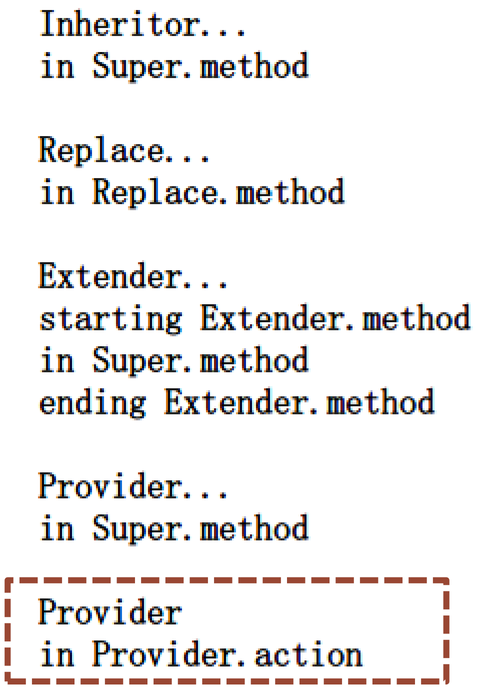
修改C.a的值，B对象的属性a会改变吗？ (独立命名空间)

继承超类的方法（重载）



```python
class Super:
    def method(self):
        print("in Super.method")
    def delegate(self):
        self.action()
class Inheritor(Super): # 直接继承
    pass
class Replace(Super): # 覆盖超类方法
    def method(self):
        print("in Replace.method")
class Extender(Super): # 扩展超类方法
    def method(self):
        print("starting Extender.method")
        Super.method(self)
        print("ending Extender.method")
class Provider(Super): # 提供超类期待的方法
    def action(self):
        print("in Provider.action")

for x in (Inheritor, Replace, Extender, Provider):
    print("\n" + x.__name__ + "...")
    x().method()
print("\nProvider")
Provider().delegate()
```

通过直接继承、覆盖超类方法、扩展超类方法、提供超类期待方法等方式重载

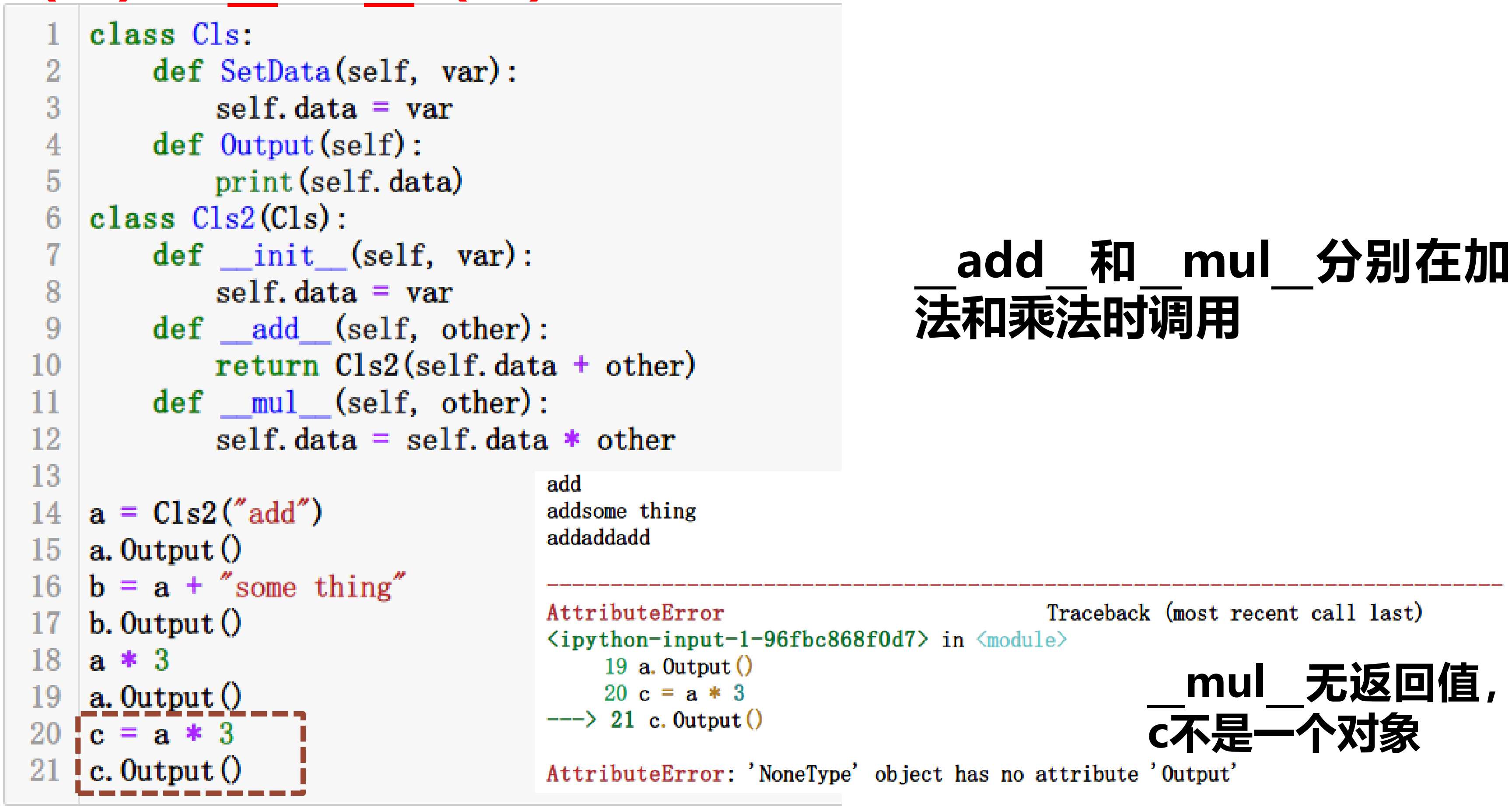


```
Inheritor...
in Super.method

Replace...
in Replace.method

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Super.method

Provider
in Provider.action
```

Super的对象能调用delegate方法吗?

运算符重载

除了特殊方法（__init__等）和自定义方法，一些运算符方法也可以重载，如__add__（加）、__sub__（减）、__mul__（乘）、__div__（除）等



```
class Cls:
    def SetData(self, var):
        self.data = var
    def Output(self):
        print(self.data)
class Cls2(Cls):
    def __init__(self, var):
        self.data = var
    def __add__(self, other):
        return Cls2(self.data + other)
    def __mul__(self, other):
        self.data = self.data * other

a = Cls2("add")
a.Output()
b = a + "some thing"
b.Output()
a * 3
a.Output()
c = a * 3
c.Output()
```

```
add
addsome thing
addaddadd

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-1-96fbc868f0d7> in <module>
     19 a.Output()
     20 c = a * 3
---> 21 c.Output()

AttributeError: 'NoneType' object has no attribute 'Output'
```

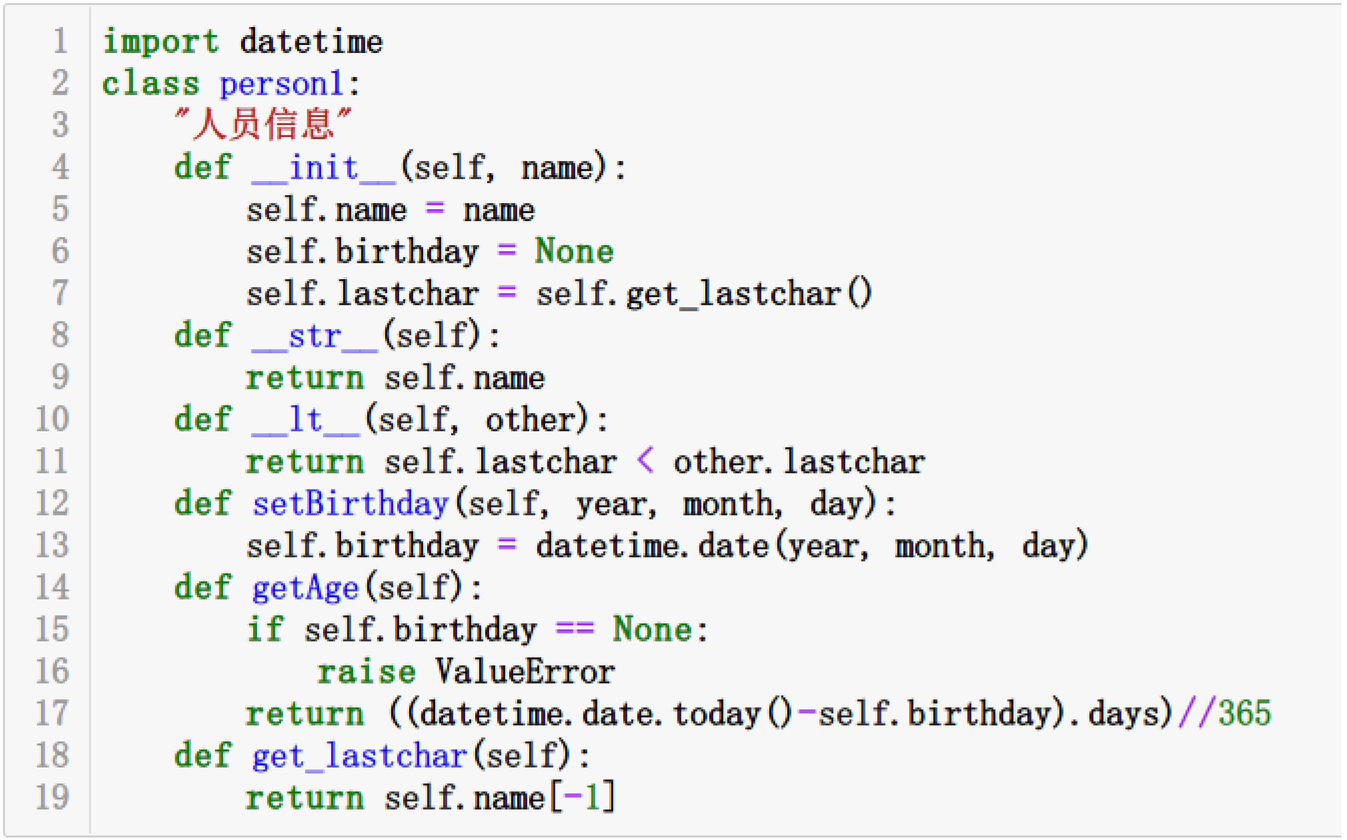
__add__和__mul__分别在加法和乘法时调用

__mul__无返回值，c不是一个对象

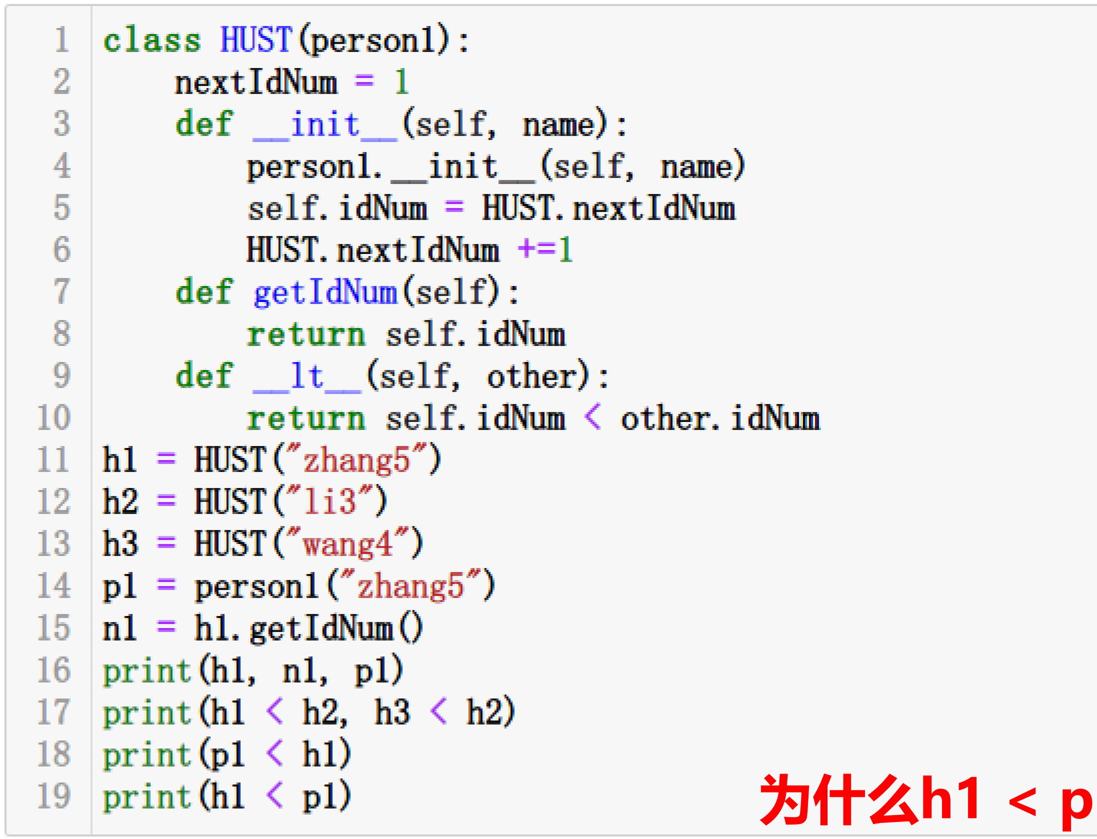
继承示例

Person1

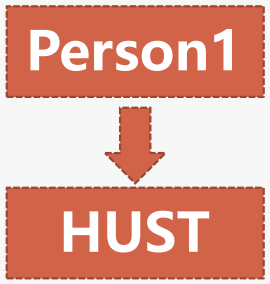
person1类



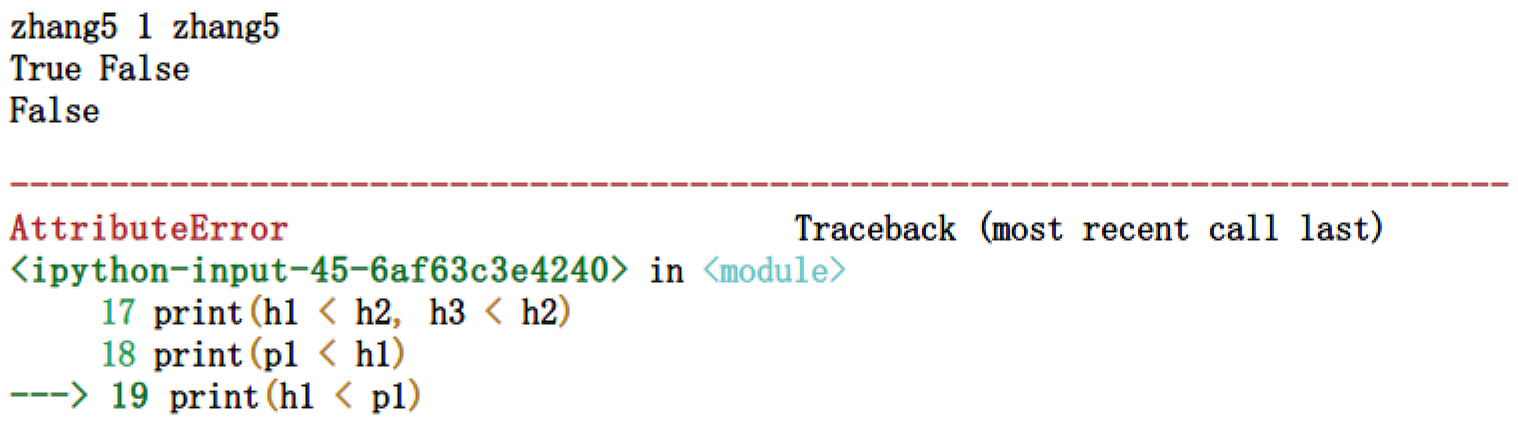
```
import datetime
class person1:
    "人员信息"
    def __init__(self, name):
        self.name = name
        self.birthday = None
        self.lastchar = self.get_lastchar()
    def __str__(self):
        return self.name
    def __lt__(self, other):
        return self.lastchar < other.lastchar
    def setBirthday(self, year, month, day):
        self.birthday = datetime.date(year, month, day)
    def getAge(self):
        if self.birthday == None:
            raise ValueError
        return ((datetime.date.today()-self.birthday).days)//365
    def get_lastchar(self):
        return self.name[-1]
```



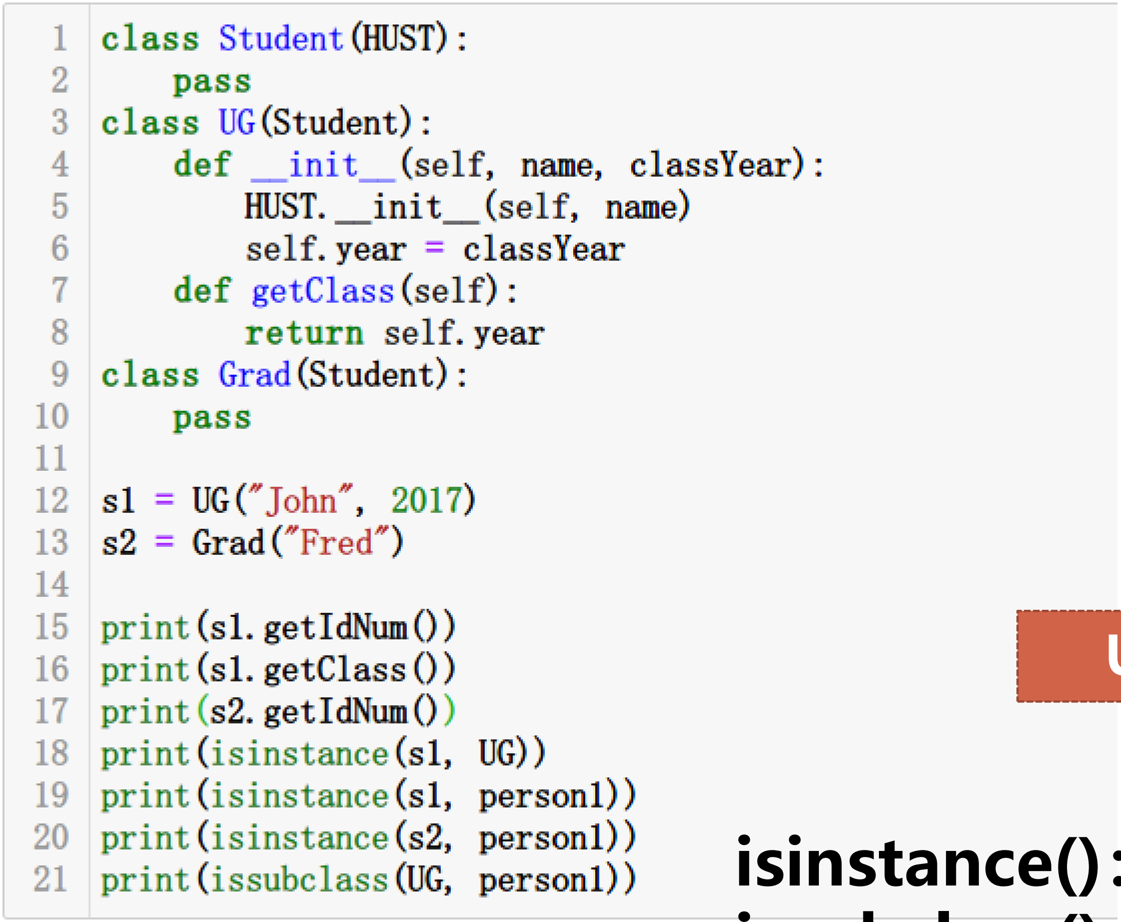
```
 1  class HUST(person1):
 2      nextIdNum = 1
 3      def __init__(self, name):
 4          person1.__init__(self, name)
 5          self.idNum = HUST.nextIdNum
 6          HUST.nextIdNum +=1
 7      def getIdNum(self):
 8          return self.idNum
 9      def __lt__(self, other):
10          return self.idNum < other.idNum
11  h1 = HUST("zhang5")
12  h2 = HUST("li3")
13  h3 = HUST("wang4")
14  p1 = person1("zhang5")
15  n1 = h1.getIdNum()
16  print(h1, n1, p1)
17  print(h1 < h2, h3 < h2)
18  print(p1 < h1)
19  print(h1 < p1)
```



为什么h1 < p1无法执行?

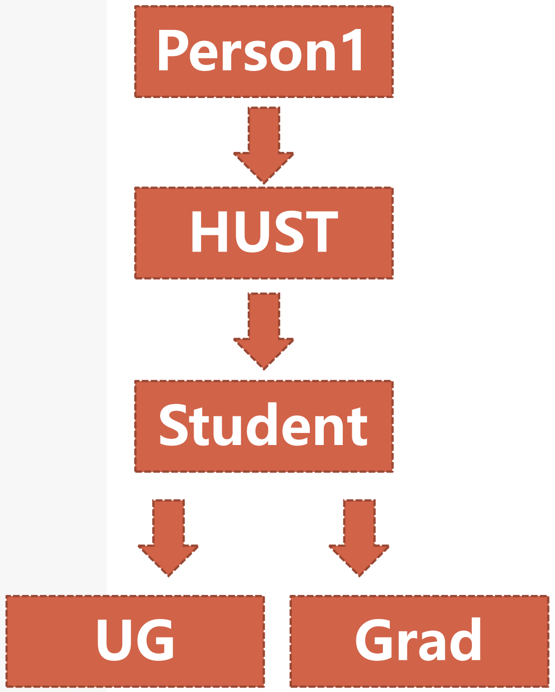


```
zhang5 1 zhang5
True False
False

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-45-6af63c3e4240> in <module>
     17 print(h1 < h2, h3 < h2)
     18 print(p1 < h1)
---> 19 print(h1 < p1)
```



```
class Student(HUST):
    pass
class UG(Student):
    def __init__(self, name, classYear):
        HUST.__init__(self, name)
        self.year = classYear
    def getClass(self):
        return self.year
class Grad(Student):
    pass

s1 = UG("John", 2017)
s2 = Grad("Fred")

print(s1.getIdNum())
print(s1.getClass())
print(s2.getIdNum())
print(isinstance(s1, UG))
print(isinstance(s1, person1))
print(isinstance(s2, person1))
print(issubclass(UG, person1))
```

```
4
2017
5
True
True
True
True
```



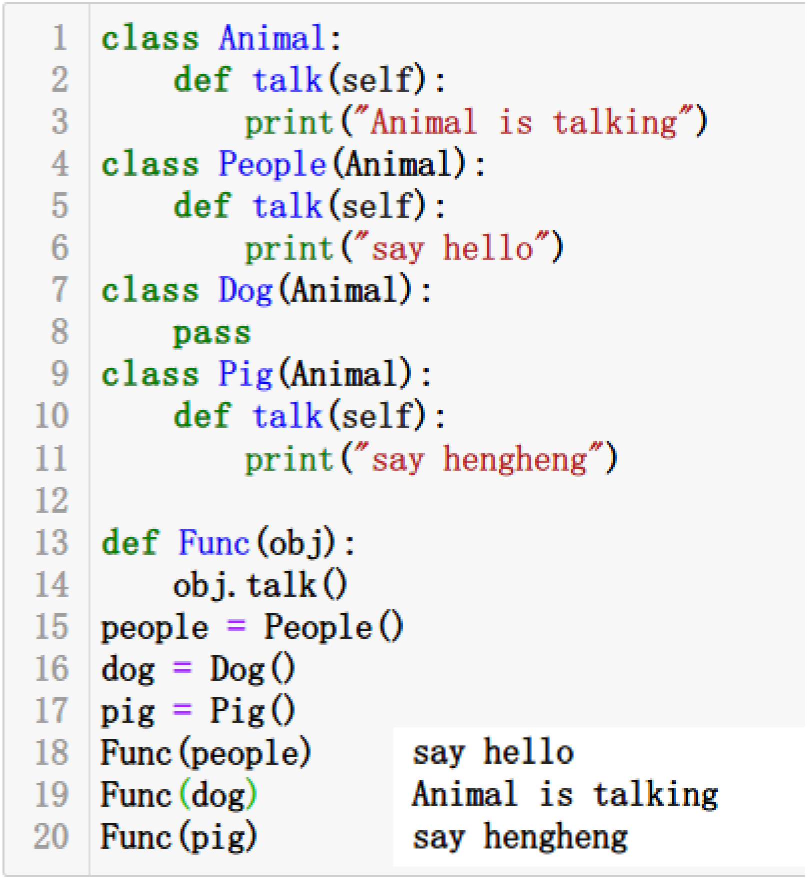
isinstance()：判断是否为对象
issubclass()：判断是否对子类

s1既是UG的对象，又是person1的对象

多态

多态性是指同一类事务具有多种形态

- **相同的类成员在不同子类有不同的重载**



```python
class Animal:
    def talk(self):
        print("Animal is talking")
class People(Animal):
    def talk(self):
        print("say hello")
class Dog(Animal):
    pass
class Pig(Animal):
    def talk(self):
        print("say hengheng")

def Func(obj):
    obj.talk()
people = People()
dog = Dog()
pig = Pig()
Func(people)
Func(dog)
Func(pig)
```

```
say hello
Animal is talking
say hengheng
```

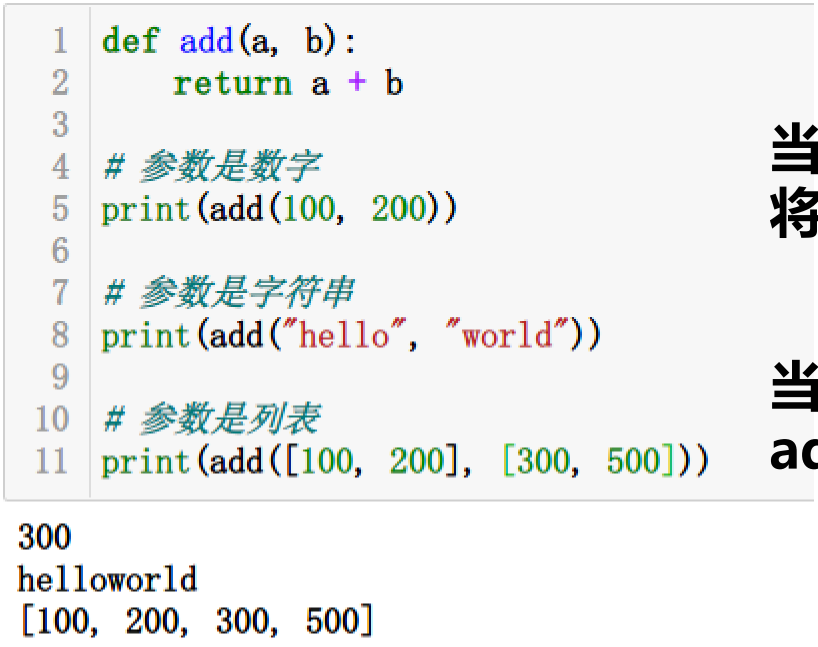
四个类都有一个talk()方法

Func()函数为同一个操作，但调用了不同的实例作为参数

“龙生九子，各有不同”

Python是一种多态语言

Python中有许多函数和运算符都是多态的，会根据接收的数据类型做出相应的运算



```
def add(a, b):
    return a + b

# 参数是数字
print(add(100, 200))

# 参数是字符串
print(add("hello", "world"))

# 参数是列表
print(add([100, 200], [300, 500]))
```

```
300
helloworld
[100, 200, 300, 500]
```

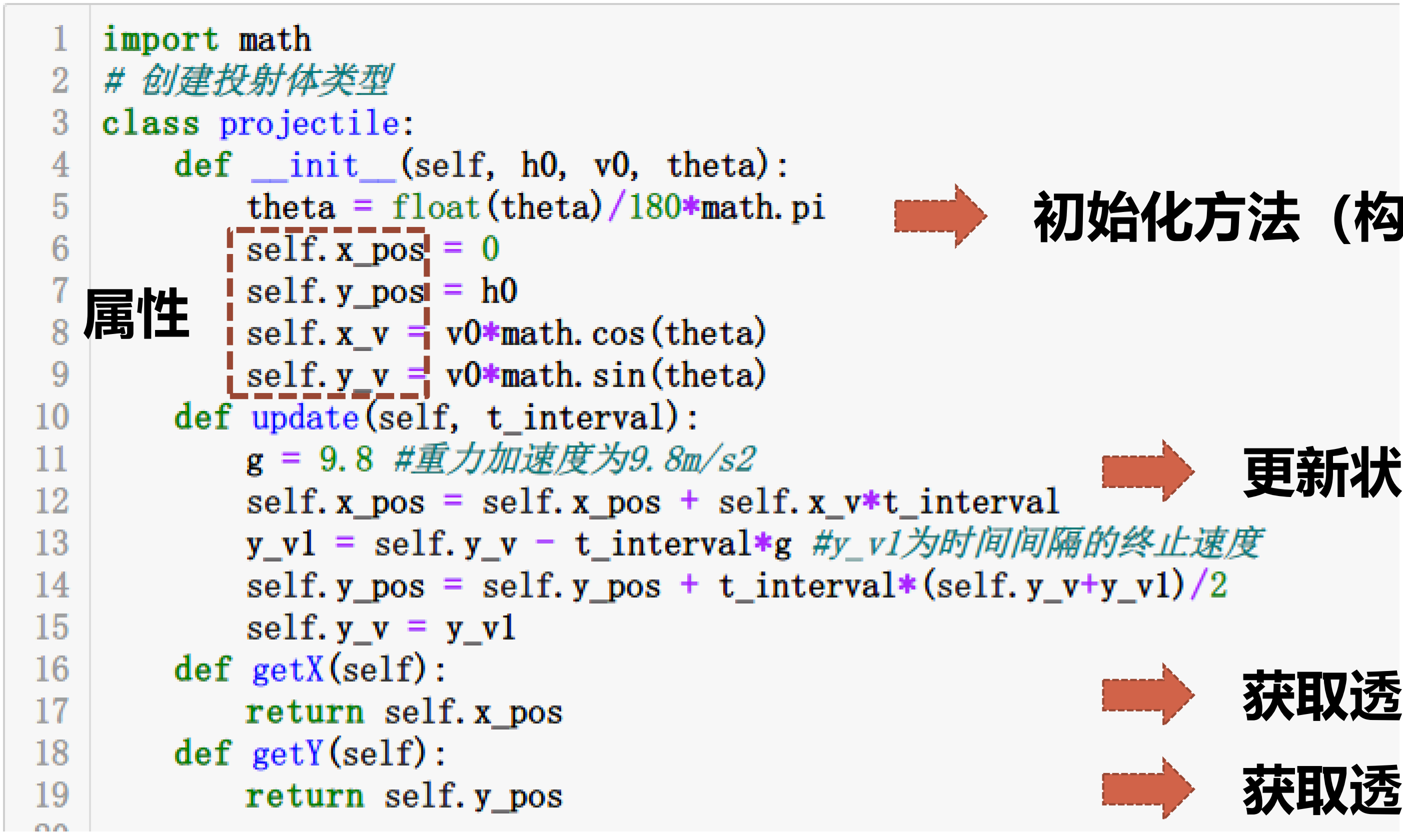
当a和b同时是数字类型时，add()函数将进行加法运算

当a和b同时是字符串类型或列表类型时，add()函数将进行拼接运算。

不需要知道对象的类型就能调用其方法

封装

封装是对外部隐藏对象内部细节，可以不用关心对象是如何构建的而直接进行使用



```python
import math
# 创建投射体类型
class projectile:
    def __init__(self, h0, v0, theta):
        theta = float(theta)/180*math.pi
        self.x_pos = 0
        self.y_pos = h0
        self.x_v = v0*math.cos(theta)
        self.y_v = v0*math.sin(theta)
    def update(self, t_interval):
        g = 9.8 #重力加速度为9.8m/s2
        self.x_pos = self.x_pos + self.x_v*t_interval
        y_v1 = self.y_v - t_interval*g #y_v1为时间间隔的终止速度
        self.y_pos = self.y_pos + t_interval*(self.y_v+y_v1)/2
        self.y_v = y_v1
    def getX(self):
        return self.x_pos
    def getY(self):
        return self.y_pos
```

属性

初始化方法（构造函数）

更新状态方法

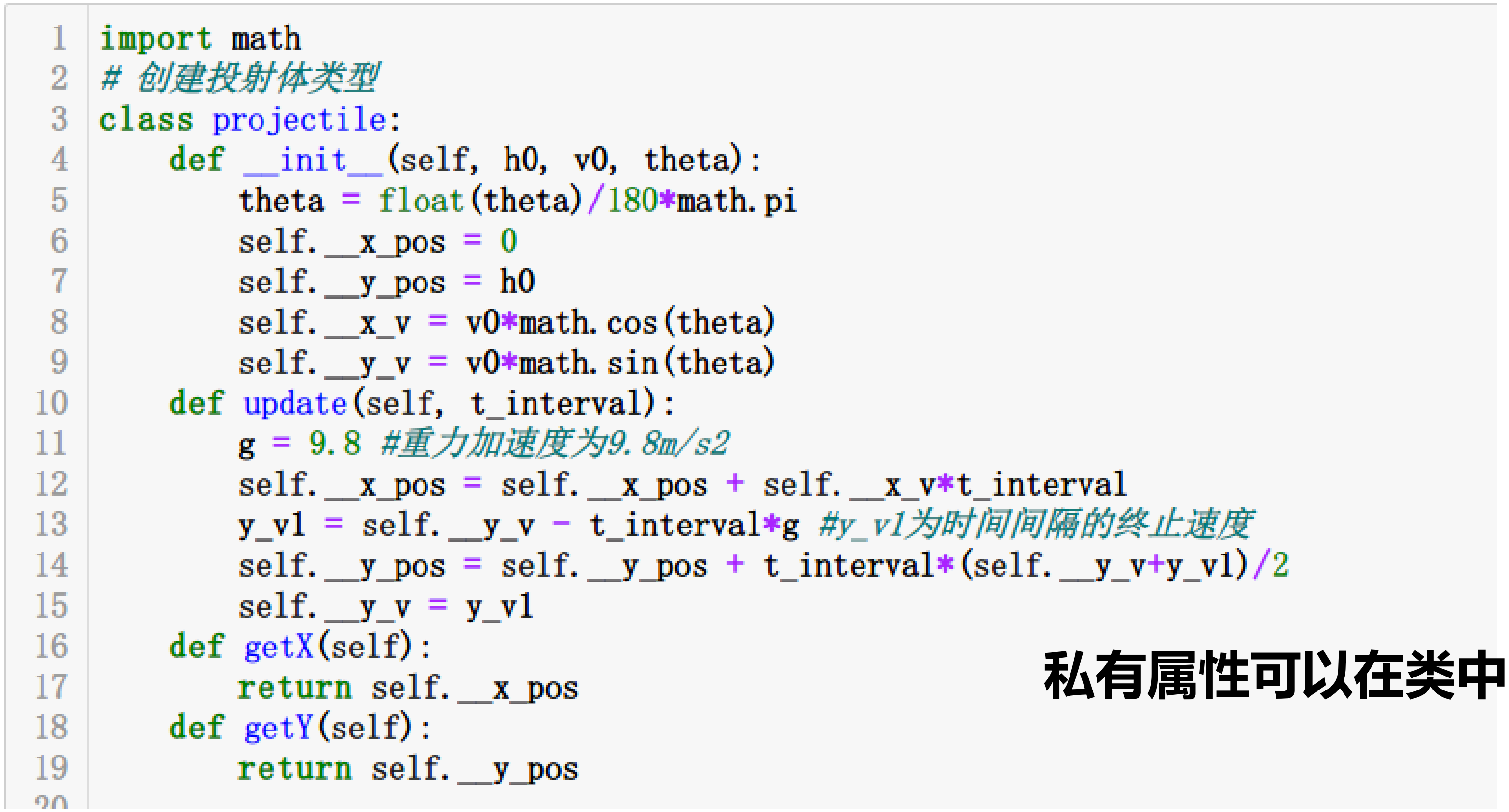
获取透射体高度

获取透射体距离

不需要关心投射体方法实现的具体细节

共有属性与私有属性

属性名前加两个下划线（__），定义私有属性

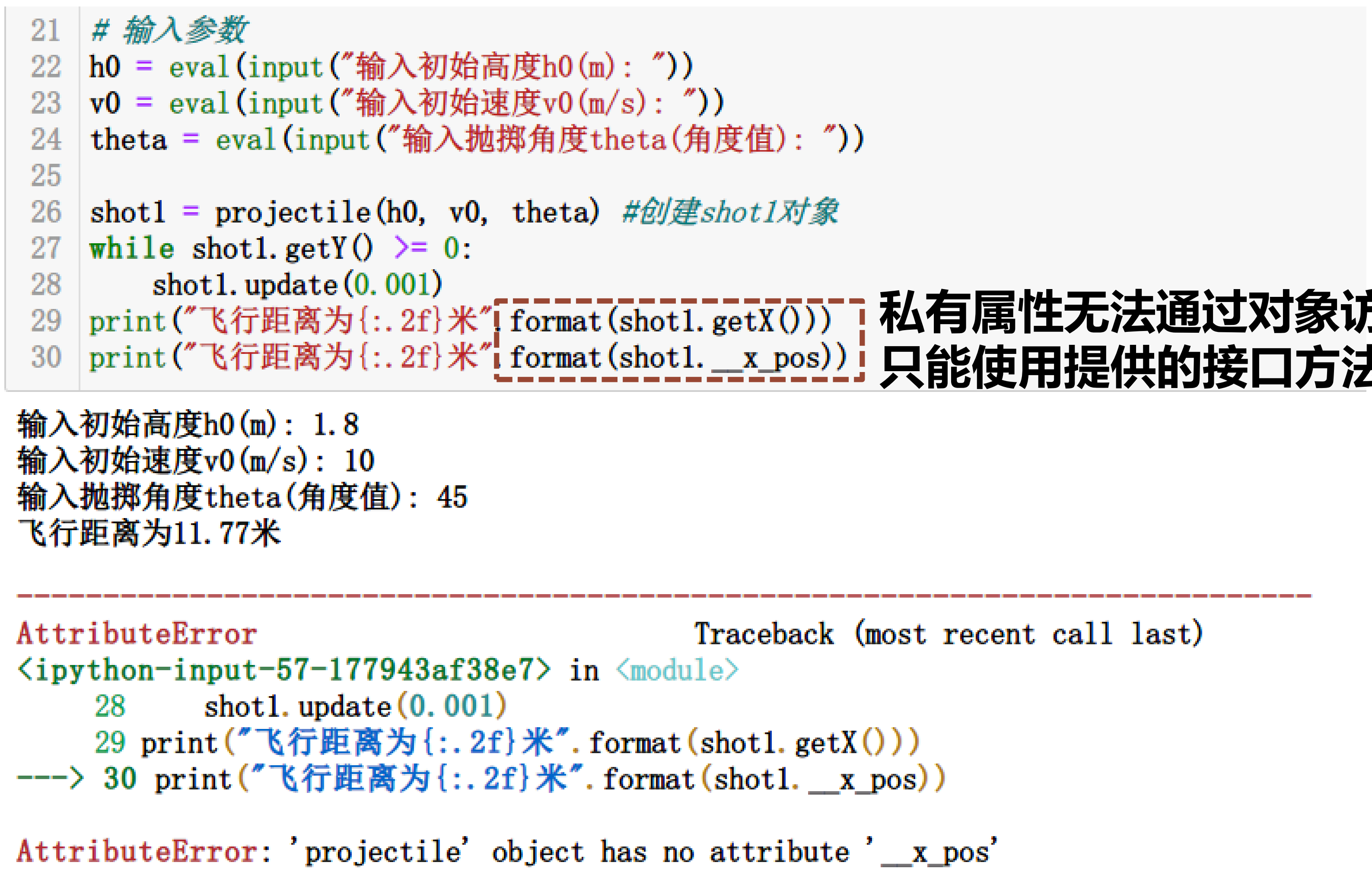


```python
import math
# 创建投射体类型
class projectile:
    def __init__(self, h0, v0, theta):
        theta = float(theta)/180*math.pi
        self.__x_pos = 0
        self.__y_pos = h0
        self.__x_v = v0*math.cos(theta)
        self.__y_v = v0*math.sin(theta)
    def update(self, t_interval):
        g = 9.8 #重力加速度为9.8m/s2
        self.__x_pos = self.__x_pos + self.__x_v*t_interval
        y_v1 = self.__y_v - t_interval*g #y_v1为时间间隔的终止速度
        self.__y_pos = self.__y_pos + t_interval*(self.__y_v+y_v1)/2
        self.__y_v = y_v1
    def getX(self):
        return self.__x_pos
    def getY(self):
        return self.__y_pos
```

私有属性可以在类中使用

相同的方法可以定义私有方法

共有属性与私有属性



```
21 # 输入参数
22 h0 = eval(input("输入初始高度h0(m): "))
23 v0 = eval(input("输入初始速度v0(m/s): "))
24 theta = eval(input("输入抛掷角度theta(角度值): "))
25
26 shot1 = projectile(h0, v0, theta) #创建shot1对象
27 while shot1.getY() >= 0:
28     shot1.update(0.001)
29 print("飞行距离为{:.2f}米".format(shot1.getX()))
30 print("飞行距离为{:.2f}米".format(shot1.__x_pos))
```

```
输入初始高度h0(m): 1.8
输入初始速度v0(m/s): 10
输入抛掷角度theta(角度值): 45
飞行距离为11.77米

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-57-177943af38e7> in <module>
     28     shot1.update(0.001)
     29 print("飞行距离为{:.2f}米".format(shot1.getX()))
---> 30 print("飞行距离为{:.2f}米".format(shot1.__x_pos))

AttributeError: 'projectile' object has no attribute '__x_pos'
```

私有属性无法通过对象访问，只能使用提供的接口方法

小结

- 面向对象的思想，类与对象的定义和使用
- 特殊的方法和属性，如__init__，__str__等
- 类的继承与重载，Python多态和封装特性

下一节：Python模块