数据结构与算法
(Python描述)

郭 炜

微信公众号

微博：http://weibo.com/guoweiofpku

学会程序和算法，走遍天下都不怕!

讲义照片均为郭炜拍摄

Python 基础知识

Python变量的指针本质

瑞士马特洪峰

一道题目

下面程序的输出结果是:

```
def swap(x,y):
        x,y = y,x
        x[0],y[0] = y[0],x[0]
a = [1,2,3]
b = [4,5,6]
swap(a,b)
print(a)
```

A) [1,2,3]
B) [4,5,6]
C) [1,5,6]
D) [4,2,3]

一道题目

下面程序的输出结果是：

```
def swap(x,y):
        x,y = y,x
        x[0],y[0] = y[0],x[0]
a = [1,2,3]
b = [4,5,6]
swap(a,b)
print(a)
```

A) [1,2,3]
B) [4,5,6]
C) [1,5,6]
D) [4,2,3]

又一道题目

下面程序的输出结果是：

```
a = [[0]] * 2 + [[0]] * 2
print(a)                    #输出[[0], [0], [0], [0]]
a[0][0] = 5
print(a)
```

A) [[5], [0], [0], [0]]
B) [[5], [5], [5], [5]]
C) [[5], [5], [0], [0]]
D) [[5], [0], [5], [0]]

又一道题目

下面程序的输出结果是：

```
a = [[0]] * 2 + [[0]] * 2
print(a)                    #输出[[0], [0], [0], [0]]
a[0][0] = 5
print(a)
```

A) [[5], [0], [0], [0]]
B) [[5], [5], [5], [5]]
C) [[5], [5], [0], [0]]
D) [[5], [0], [5], [0]]

Python中的变量都是指针

- Python中所有可赋值的东西，即可以出现在赋值号"="左边的东西，都是指针

- 指针即代表内存单元的地址

- 将指针称作 “箭头",更容易理解。所有变量都是箭头，指向内存某处

- 对变量进行赋值的本质，就是让该变量（箭头）指向某个地方

Python中的变量都是指针

- 对变量进行赋值，意味着将变量指向某处

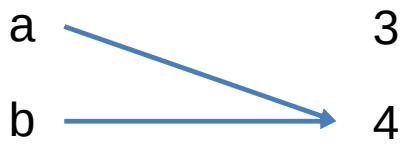
```
a = 3
b = 4
```

Python中的变量都是指针

- 用一个变量对另一个变量赋值意味着让

  两个变量指向同一个地方

```
a = b
```



is 运算符和 == 的区别

- a is b　为True 说a和b 指向同一个地方

- a == b　为True 说明a和b指向的地方放的的东西相同，但是a和b不一定指向相同的地方

- a = b 会使得a和b指向同一个地方

3

is 运算符和 == 的区别

x is y 表示x和y是否指向同一个地方

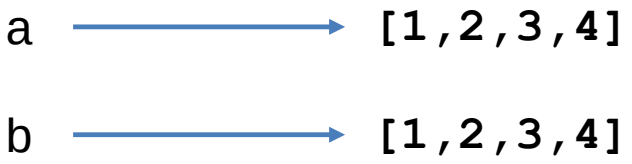
x == y 表示x和y的内容是否相同

```
a = [1,2,3,4]

b = [1,2,3,4]

print( a == b) #>>True

print( a is b) #>>False
```

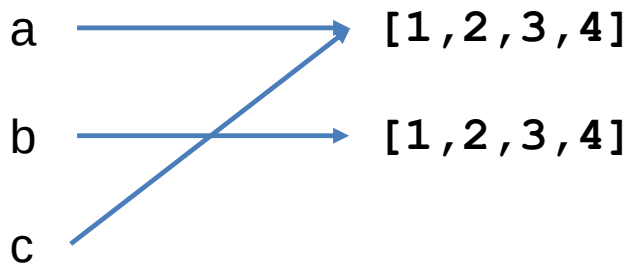


is 运算符和 == 的区别

x is y 表示x和y是否指向同一个地方

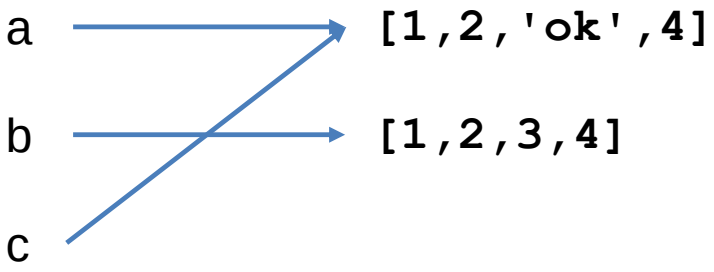
x == y 表示x和y的内容是否相同

```
a = [1,2,3,4]
b = [1,2,3,4]
print( a == b) #>>True
print( a is b) #>>False
c = a
print( a == c) #>>True
print( a is c) #>>True
```



is 运算符和 == 的区别

```
a[2] = "ok"
print(c)   #>>[1, 2, 'ok', 4]
```



因为a和c指向同一个地方，所以修改a[2]，c[2]也变。

a[2]和c[2]是同一个东西

is 运算符和 == 的区别

- 对int ,float,complex, str,tuple类型的变量a和b，只需关注 a == b是否成立，关注 a is b是否成立无意义。因这些数据本身都不会更改，不会产生a指向的东西改了b指向的东西也跟着变的情况

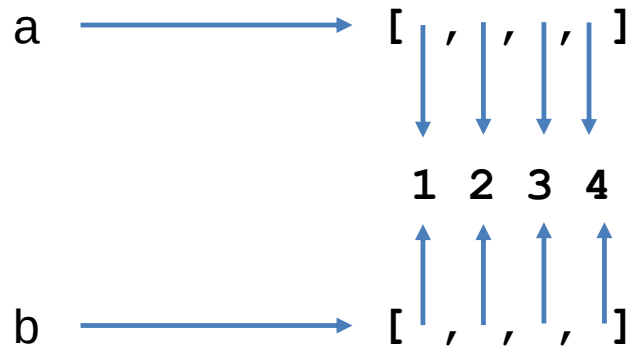
- 对 list,dict,set类型的变量a和b， a == b和 a is b的结果都需要关注。因这些数据本身会改变。改变了a指向的内容，说不定b指向的内容也变了。

列表元素的指针本质

- 列表的元素也可以赋值，因此也是指针

```
a = [1,2,3,4]
b = [1,2,3,4]
```

准确的效果：



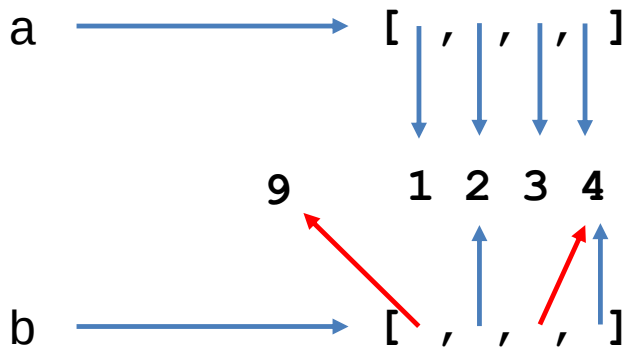
a[0],a[1]....b[0],b[1] ..... 都是指针

列表元素的指针本质

- 列表的元素也可以赋值，因此也是指针

```
a = [1,2,3,4]
b = [1,2,3,4]
```

准确的效果：



执行 b[0],b[2] = 9,4　后

元组元素的指针本质

- 元组的元素虽然不可赋值，但也是指针

```
a = [1,2,3,4]
b = (100,a)
a[0] = 1000
print(b)      #>>(100, [1000, 2, 3, 4])
```

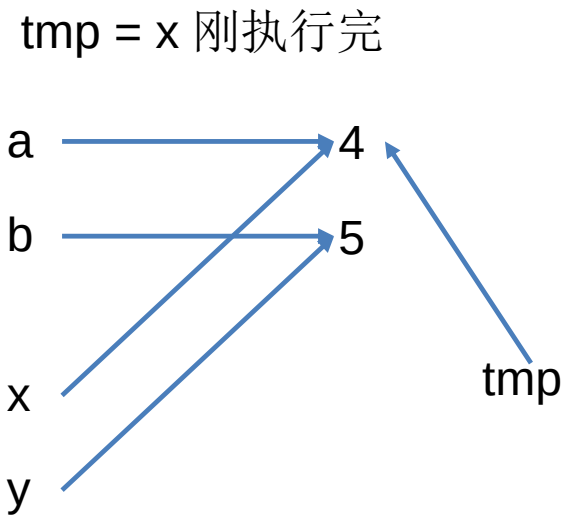
函数参数的传递

京都金阁寺

函数参数的传递

- 函数参数传递方式都是传值，即形参是实际参数的一个拷贝。函数参数也是指针。形参和实参指向同一个地方。对形参赋值（让其指向别处）不会影响实参。

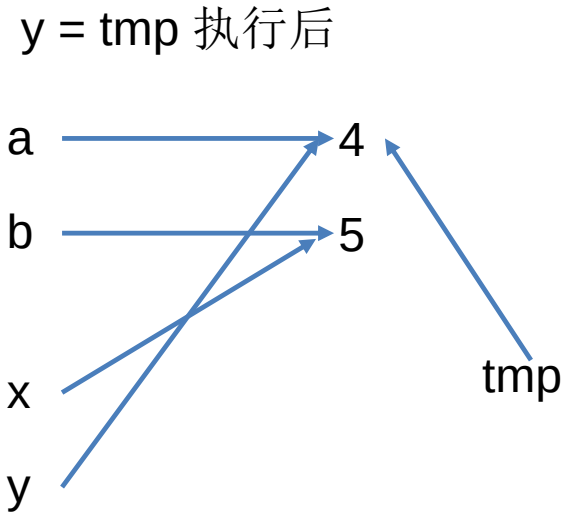
```
def Swap(x,y):
    tmp = x
    x = y
    y = tmp
a = 4
b = 5
Swap(a,b)
print(a,b)      #>>4, 5
```



函数参数的传递

- 函数参数传递方式都是传值，即形参是实际参数的一个拷贝。函数参数也是指针。形参和实参指向同一个地方。对形参赋值（让其指向别处）不会影响实参。

```
def Swap(x,y):
    tmp = x
    x = y
    y = tmp
a = 4
b = 5
Swap(a,b)
print(a,b)      #>>4, 5
```



函数参数的传递

- 但是如果函数执行过程中，改变了形参所指向的地方的内容，则实参所指向的地方内容也会被改变。

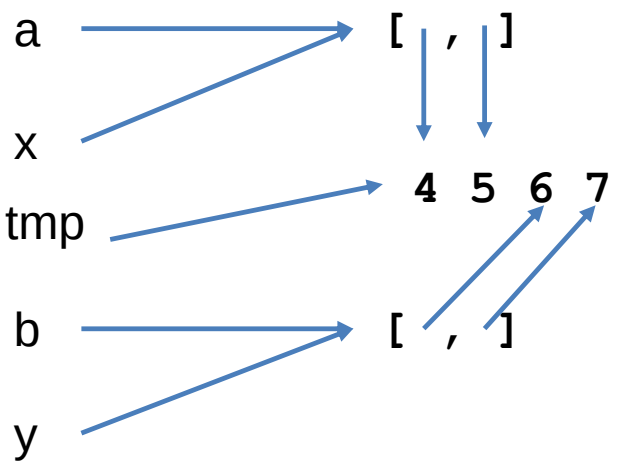
```
def Swap(x,y):
    tmp = x[0]
    x[0] = y[0]     #若x,y是列表，则x[0],y[0],tmp都是指针
    y[0] = tmp


a = [4,5]
b = [6,7]
Swap(a,b)     #进入函数后，x和a指向相同地方，y和b指向相同地方
print(a,b)      #>>[6, 5] [4, 7]
```

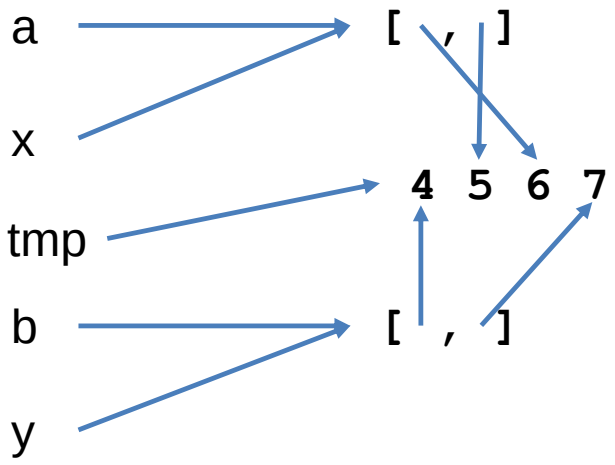
函数参数的传递

进入Swap函数执行完

tmp = x[0] 时



Swap函数执行完时：



程序或算法的时间复杂度

美国加州1号公路

程序或算法的时间复杂度

- 一个程序或算法的时间效率，也称“时间复杂度”，有时简称“复杂度”

程序或算法的时间复杂度

- 一个程序或算法的时间效率，也称“时间复杂度”，有时简称“复杂度”

- 复杂度常用大的字母O和小写字母n来表示，比如O(n)，O(n^2)等。n代表问题的规模

程序或算法的时间复杂度

●一个程序或算法的时间效率，也称“时间复杂度”，有时简称“复杂度”

●复杂度常用大的字母O和小写字母n来表示，比如O(n)，O(n^2)等。n代表问题的规模。O(X）可大致认为表示解决问题的时间和X成正比关系

●时间复杂度是用算法运行过程中，某种时间固定的操作需要被执行的次数和n的关系来度量的。在无序数列中查找某个数，复杂度是O(n)

程序或算法的时间复杂度

●一个程序或算法的时间效率，也称“时间复杂度”，有时简称“复杂度”

●复杂度常用大的字母O和小写字母n来表示，比如O(n)，O(n^2)等。n代表问题的规模，O(X）可大致认为表示解决问题的时间和X成正比关系

●时间复杂度是用算法运行过程中，某种时间固定的操作需要被执行的次数和n的关系来度量的。在无序数列中查找某个数，复杂度是O(n)

●计算复杂度的时候，只统计执行次数最多的(n足够大时)那种固定操作的次数。比如某个算法需要执行加法n^2次，除法10000n次，那么就记其复杂度是O(n^2)的。

程序或算法的时间复杂度

- 如果复杂度是多个n的函数之和，则只关心随n的增长增长得最快的那个函数

$O(n^3+n^2) \Rightarrow O(n^3)$
$O(2^n+n^3) \Rightarrow O(2^n)$
$O(n! + 3^n) \Rightarrow O(n!)$

程序或算法的时间复杂度

- 如果复杂度是多个n的函数之和，则只关心随n的增长增长得最快的那个函数

$O(n^3+n^2) \Rightarrow O(n^3)$
$O(2^n+n^3) \Rightarrow O(2^n)$
$O(n! + 3^n) \Rightarrow O(n!)$

- 常数复杂度：O(1)
- 对数复杂度：O(log(n))
- 线性复杂度：O(n)
- 多项式复杂度：$O(n^k)$
- 指数复杂度：$O(a^n)$
- 阶乘复杂度：O(n!)

时间(操作次数)和问题的规模无关

程序或算法的时间复杂度

- 在无序数列中查找某个数(顺序查找)　　O(n)
- 插入排序、选择排序等笨排序方法　O(n^2)
- 快速排序　O(n*log(n))
- 二分查找　O(log(n))

in用于列表和用于字典、集合的区别

`a in b`

若b是列表，字符串或元组，则该操作时间复杂度O(n)，即时间和b的元素个数成正比

若b是字典或集合，则该操作时间复杂度O(1)，即时间基本就是常数，和b里元素个数无关

因此集合用于需要经常判断某个东西是不是在一堆东西里的情况

此种场合用列表替代集合，容易导致超时!!!!

一些操作的时间复杂度总结

O(1)：集合、字典增删元素，查找元素，以关键字作为下标访问字典元素的值，列表添加元素到末尾（append），列表、字符串、元组根据下标访问元素

O(n)：列表、元组查找元素(in,index)，列表插入元素(insert)、删除元素(remove)

计算出现次数(count)，求列表最大值最小值(min,max)

O(nlog(n))：python自带排序 sort,sorted

O(log(n))：在排好序的列表或元组上进行二分查找（初始的查找区间是整个元组或列表，每次和查找区间中点比较大小，并缩小查找区间到原来的一半。类似于查英语词典）有序就会找得快！

常见错误：没有时间观念

- 要意识到某些操作是需要较多时间的（不是常数时间）

```
if int(s[1])>=60:
        m.append((s[0],int(s[1])))
        m.sort(key=lambda x: (-x[1]))
    else:
        a.append(s[0])
```

应该添加完后，最后排序一次

非计算机专业人士，不求想出精妙的节约时间的算法，起码不要犯无谓浪费时间的错误（花费时间有数量级上的增加）

常见错误：没有时间观念

- 要意识到某些操作是需要较多时间的（不是常数时间）

```
s = [10, 2, 7.....]
a = max(s)*max(s)            #max      是O(n)的!
```

应该写：

```
a = max(s)
a *= a
```

耗时操作的结果如果要重复使用，应该只算一遍就存起来以后用，而不是重复计算

常见错误：没有时间观念

- 来自**oj**实交代码：

```
while i<=c:
        if judge(alist,alist[pos]+t):
            pos=judge(alist,alist[pos]+t)   #重复调用judge
            i+=1
        else:
            break
    if i==c:
        return True
```

常见错误：没有时间观念

- 来自oj实交代码：

```
   if judge(x,mid):
        temp=mid
        l=mid+1
 #如果4、5，4、5都满足条件，mid=4，l=4陷入死循环；如果5不满足，mid=4，
l=5，mid=5，r=mid-1=4
        #做如下调整：用temp记录目前最优解
    elif not judge(x,mid):
        r=mid-1
```

注意：尽量用库函数

- **写法1**

```
string = "....."
n = string.count('a')  #count也要从头到尾把string看一遍，复杂度O(n)
```

写法2

```
n = 0
for c in string:
        if c == 'a':
                n += 1
```

两种写法虽然复杂度一样，但是写法1直解释执行一条语句，count内部实现是机器指令速度很快，而写法2要反复解释执行每条语句，所以明显慢

数结构的概念

- 数据经过合理组织，就能查找、添加、删除、修改都快

- 排序以后二分查找
- 字典和集合是“哈希表”这种数据结构

程序或算法的空间复杂度

- 一般就看开了多大的数组

- 有递归的时候，要计算递归函数的局部变量的栈空间

Python中的类

内蒙古浑善达克沙地

类的定义

```
class 类名:
    def __init__(self,参数1,参数2......):   #构造函数
        self.属性1 = 参数1
        self.属性2 = 参数2
        .......
    def 成员函数1(self,参数1,参数2......):
        .......
    def 成员函数2(self,参数1,参数2......):
        .......
    ......
    def 成员函数n(self,参数1,参数2......):
        .....
```

类和对象的概念

- 类概括了一种事物的特点，包括属性和方法(成员函数)
- 对象是通过类定义的变量，一个对象就是一个类的实例
- Python中所有的变量，以及小数、复数、字符串、元组、列表、集合、字典等组合数据类型的常量，都是对象，函数也是对象，但整数型常量不是对象。
- 整型变量所属的类是int，小数所属于的类是float,字符串所属于的类是str，列表所属的类是list....

类和对象示例

```
class rectangle:
    def __init__(self,w,h):
        self.w,self.h = w,h
    def area(self):
        return self.w * self.h
    def perimeter(self):
        return 2 * (self.w + self.h)
def main():
    w,h = map(int,input().split())                    #假设输入2 3
    rect = rectangle(w,h)              #rect是对象 用构造函数初始化
    print(rect.area(),rect.perimeter())               #>>6 10
    rect.w,rect.h = 10,20
    print(rect.area(),rect.perimeter())               #>>200 60
    rect2 = rectangle(2,3)
    print(rect2.area(), rect2.perimeter())            #>>6 10
main()
```

对象的比较

- 默认情况下，自定义类的对象a和b，只能用 == 比较，且 a == b 等价于a is b
- 所有类都有 __eq__方法。 x == y 等价于 x.__eq__(y)，若x.__eq__(y)无定义，则等价于 y.__eq__(x) （x,y都是整型常量例外）

```
print(24.5.__eq__(24.5))      #>>True
```

- 默认情况下，自定义类的__lt__、__gt__、__le__、__ge__方法都被设置成了None，通过重写__eq__和这些成员函数，可以让 == 含义变化，以及对象可以用<,>,<=,>=进行比较

对象的比较

```
class point:
    def __init__(self, x, y = 0):
        self.x , self.y = x,y
    def __eq__(self,other):
        return self.x == other.x and self.y == other.y
    def __lt__(self,other):      #使得两个point对象可以用<进行比较
        if self.x == other.x:
            return self.y < other.y
        else:
            return self.x < other.x
a,b = point(1,2),point(1,2)
print(a == b)                    #>>True
print(a != b)                    #>>False
print(a < point(0,1))            #>>False
print(a < point(1,3))            #>>True
```

对象的比较

```
lst = [a,point(-2,3),point(7,8),point(5,9),point(5,0)]
lst.sort()
for p in lst:                          #>>-2 3,1 2,5 0,5 9,7 8,
    print(p.x,p.y ,end = ",")
```

对象的比较

- 改写__eq__使得对象不可比较

```
class A:
    def __init__(self,x):
        self.x = x
        A.__eq__ = None
a,b = A(3),A(4)
print( a == b )        #runtime error
```

继承和派生

- 可以从已经有的类派生出新类，以进行代码复用
- 派生类自动拥有基类的全部属性和方法，还可以覆盖基类的方法

```
import datetime
class student:
    def __init__(self,id,name,gender,birthYear):
        self.id,self.name,self.gender,self.birthYear = \
            id,name,gender,birthYear
    def printInfo(self):
        print("Name:",self.name)
        print("ID:", self.id)
        print("Birth Year:",self.birthYear)
        print("Gender:",self.gender)
        print("Age:",self.countAge())
    def countAge(self):
        return datetime.datetime.now().year - self.birthYear
```

继承和派生

```
class undergraduateStudent(student): #本科生类，继承了student类
    def __init__(self,id,name,gender,birthYear,department):
        student.__init__(self,id,name,gender,birthYear)
        self.department = department
    def qualifiedForBaoyan(self):              #给予保研资格
        print("Qualified for baoyan")
    def printInfo(self):                       #基类中有同名方法
        student.printInfo(self)                #调用基类的PrintInfo
        print("Department:" ,self.department)

s2 = undergraduateStudent("118829212","Harry Potter"
            ,"M",2000, "Computer Science")
s2.printInfo()
s2.qualifiedForBaoyan()
if s2.countAge() > 18:
    print(s2.name , "is older than 18")
```

继承和派生

- Python所有类,包括自定义类，均派生自object类，因而自动继承object类的方法

```
class A:
    def func(x):
        pass
print(dir(A))          #列出类A的方法
```

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

静态属性和静态方法

- 静态属性被所有对象共享，不是每个对象各自一份
- 静态方法不是作用在某个具体对象上面的

```
class employee:
    totalSalary = 0                    #静态属性，记录发给员工的工资总数
    def __init__(self,name,income):
        self.name,self.income = name, income
    def pay(self,salary):
        self.income += salary
        employee.totalSalary += salary

    @staticmethod
    def printTotalSalary():        # 静态方法
        print(employee.totalSalary)
```

静态属性和静态方法

```
e1 = employee("Jack",0)
e2 = employee("Tom",0)
e1.pay(100)
e2.pay(200)
employee.printTotalSalary()      #>>300
e1.printTotalSalary()            #>>300
e2.printTotalSalary()            #>>300
print(employee.totalSalary)      #>>300
```

对象作为字典的键或集合的元素

- 默认情况下，自定义类的对象，可以作为集合元素或字典的键，被作为集合元素或字典键的，是对象的id，因此意义不大

```
class A:
    def __init__(self,x):
        self.x = x

a,b = A(5),A(5)          #两个A(5)不是同一个，因此a和b的id不同
dt = {a:20,A(5):30,b:40}          #三个元素的键id不同，因此在不同槽里
print(len(dt),dt[a],dt[b])       #>>3 20 40
print(dt[A(5)])                   #runtime error
```

对象作为字典的键或集合的元素

- 集合和字典都是哈希表，可哈希的类的对象才可以作为集合的元素或者字典的键
- 一个类，有__hash__方法，即为可哈希。自定义类的默认__hash__方法根据对象id算哈希值，哈希值是个整数
- __hash__函数返回值相同的两个对象a,b，在集合中放在同一个槽里。若a==b成里，则只能保留一个(字典的键类似处理）。
- 如果为自定义类重写__eq__方法，则其__hash__方法会被Python自动变成None，其变成不可哈希

对象作为字典的键或集合的元素

- 类的__hash__方法示例

```
x = 23.1
print(x.__hash__(),23.1.__hash__())
#>>2305843009213726952305843009213726952
x = 23
print(x.__hash__(),hash(23))    #>>23 23
x = (1,2)
print(x.__hash__(),(1,2).__hash__(),hash(x))
#>>3713081631934410656 3713081631934410656 3713081631934410656
x = "ok"
print(x.__hash__(), "ok".__hash__())
#>>-4237608756544806032-4237608756544806032
```

对象作为字典的键或集合的元素

- 为自定义类重写__eq__和__hash__方法，可以做到用对象的值作为集合元素或字典的键

```
class  A:
    def __init__(self,x):
        self.x = x
    def __eq__(self,other):
        if isinstance(other,A): #判断other是不是类A的对象
            return self.x == other.x
        elif isinstance(other,int): #如果other是整数
            return self.x == other
        else:
            return False
    def __hash__(self):
        return self.x
```

对象作为字典的键或集合的元素

```
a = A(3)
print(3 == a)                          #>>True
b = A(3)
d = {A(5):10,A(3):20,a:30}
print(len(d),d[a],d[b],d[3])           #>>2 30 30 30
```