



北京大学
PEKING UNIVERSITY

信息科学技术学院

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 知识巩固与提高



北京大学
PEKING UNIVERSITY

信息科学技术学院

郭炜

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中所有可赋值的東西，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`

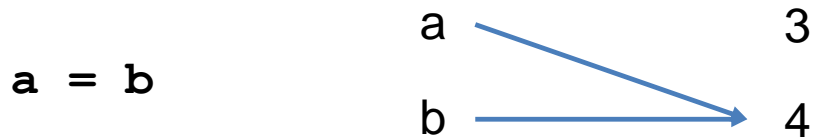
`a`  `3`

`b = 4`

`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



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

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
```

a → [1,2,3,4]

b → [1,2,3,4]

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]
```

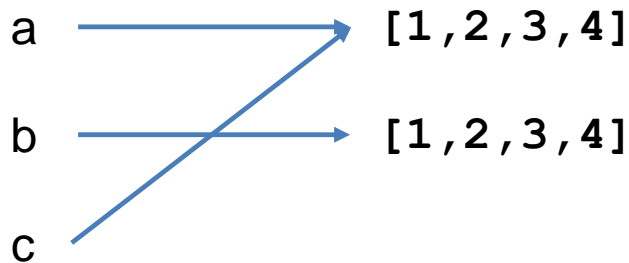
```
print( a == b) #>>True
```

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

```
c = a
```

```
print( a == c) #>>True
```

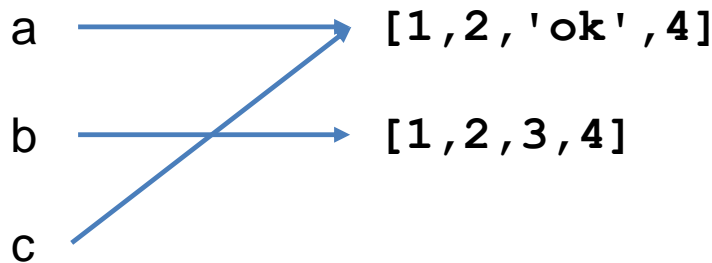
```
print( a is c) #>>True
```



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。

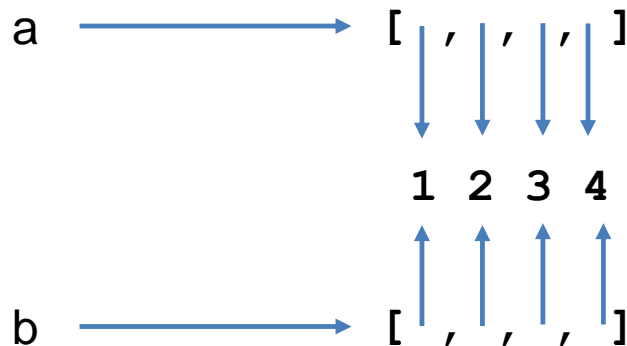
列表元素的指针本质

- 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

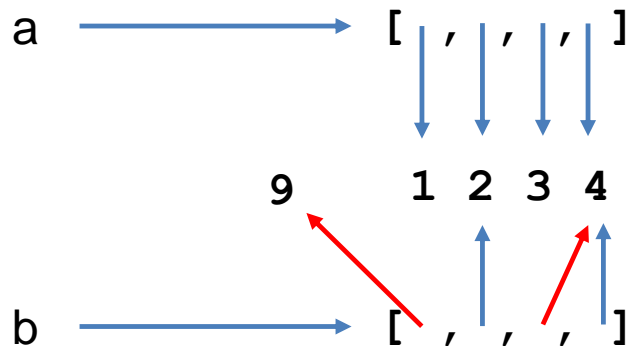
列表元素的指针本质

- 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])
```



北京大学
PEKING UNIVERSITY

信息科学技术学院

郭炜

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
```

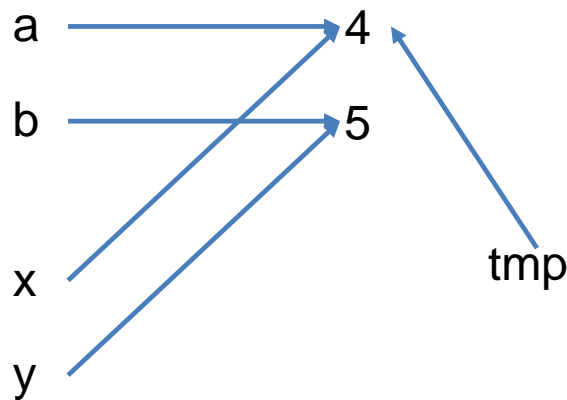
```
a = 4
```

```
b = 5
```

```
Swap(a,b)
```

```
print(a,b)      #>>4, 5
```

tmp = x 刚执行完



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
```

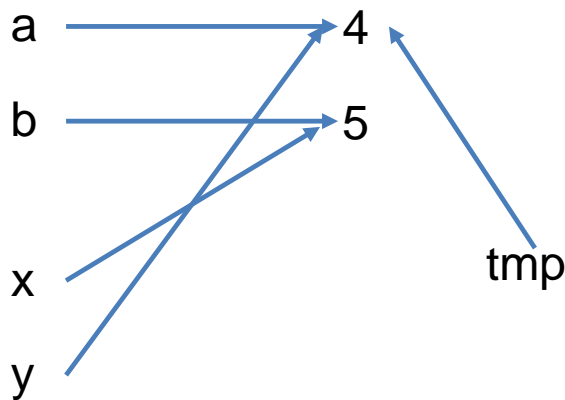
```
a = 4
```

```
b = 5
```

```
Swap(a,b)
```

```
print(a,b)      #>>4, 5
```

y = tmp 执行后



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]
```

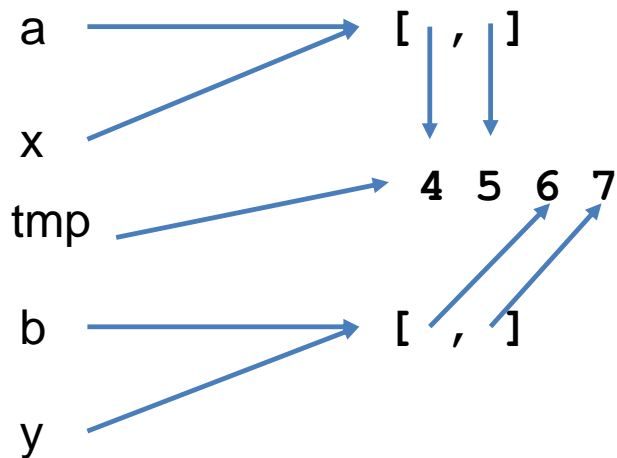
```
Swap(a,b)    #进入函数后，x和a指向相同地方，y和b指向相同地方
```

```
print(a,b)    #>>[6, 5] [4, 7]
```

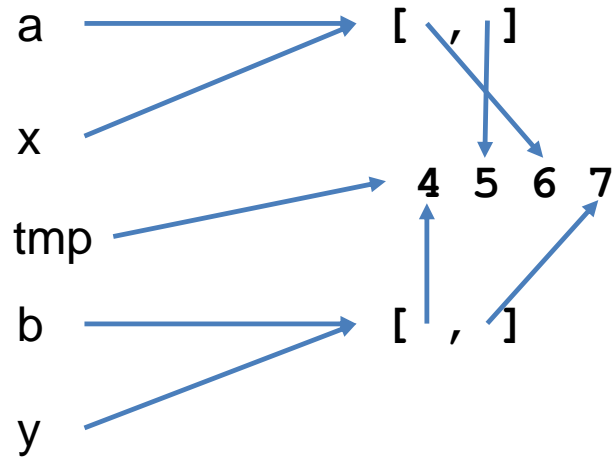
函数参数的传递

进入Swap函数执行完

`tmp = x[0]` 时



Swap函数执行完时:





北京大学
PEKING UNIVERSITY

信息科学技术学院

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)

$O(n \log(n))$: 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直接解释执行一条语句，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



北京大学
PEKING UNIVERSITY

信息科学技术学院

Python中的函数



内蒙古浑善达克沙地

global变量和nonlocal变量

- 'global'关键字用于声明变量是在所有函数外部定义的变量。'nonlocal'变量用于声明变量是在函数的外围函数定义的变量：

```
k1,k2 = 1,2
def f():
    n1,n2 = 10,20
    def g():
        nonlocal n1
        global k1
        k1 *= 10
        n1 *= 10
        print(k2, n2)    #>>2 20
    g()
    print(k1, n1)        #>>10 100
f()
```

闭包

➤ 闭包：带自由变量的函数

```
def func(x):  
    def g(y):  
        nonlocal x    #有了此行，才能在g中对x赋值  
        x += 1  
        return x+y  
    return g  
  
f = func(10)           #f是一个闭包，其自由变量x初值是10  
print(f(4))           #>>15  
print(f(5))           #>>17  
g = func(20)           #g是一个闭包，其自由变量x初值是20  
print(g(4))           #>>25  
print(g(5))           #>>27
```

函数参数的默认值

- 函数允许有些参数有默认值，即调用的时候如果不给出这些参数，这些参数的值就自动取默认值。

```
def f(a ,b = 1 ,op = lambda x,y:x+y):  
    print(a,b,op(a,b))
```

```
f(0)                                #>>0 1 1
```

```
f(1,100)                           #>>1 100 101
```

```
f(2,20,lambda x,y:x*y)             #>>2 20 40
```

生成器(generator)

- yield关键字用来定义生成器（Generator），可以当return使用，从函数里返回一个值
- 使用了 yield 的函数被称为生成器函数。当函数被调用的时候，并不执行函数，而是返回一个生成器(generator)
- 如果 X 是一个生成器函数被调用时的返回值（生成器），则

```
for i in X:  
    print(i)
```

会依次打印X中yield语句返回的结果，直到函数X再也不会执行到yield语句

生成器

```
def test_yield(): #调用则返回迭代器
    yield 1
    yield 2
    yield (1,2)
for x in test_yield():
    print(x)
```

输出:

1

2

(1,2)

生成器

- 使用 yield 实现斐波那契数列：

```
def fib (n): # 生成器函数 - 用于求斐波那契数列前n项
```

```
    a, b, counter = 0, 1, 0
```

```
    while counter <= n:
```

```
        yield a
```

```
        a, b = b, a + b
```

```
        counter += 1
```

```
for x in fib(10):
```

```
    print(x,end = ",")
```

```
0 1 1 2 3 5 8 13 21 34 55
```

生成器的一个优点就是它不要求事先准备好整个迭代过程中所有的元素。生成器仅仅在迭代至某个元素时才计算该元素，而在这之前或之后，元素可以不存在或者被销毁。这个特点使得它特别适合用于遍历一些巨大的或是无限的集合，比如几个G的文件，或是斐波那契数列等等。这个特点被称为延迟计算或惰性求值(Lazy evaluation)。



北京大学
PEKING UNIVERSITY

信息科学技术学院

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，小数所属的类是float，字符串所属的类是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())
    rect = rectangle(w,h)
    print(rect.area(),rect.perimeter())
    rect.w,rect.h = 10,20
    print(rect.area(),rect.perimeter())
    rect2 = rectangle(2,3)
    print(rect2.area(), rect2.perimeter())

main()
```

#假设输入2 3
#rect是对象 用构造函数初始化
#>>6 10
#>>200 60
#>>6 10

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
```

对象的拷贝

- 要在对象间进行复制，编写一个`copy`函数比较方便

```
class point:
    def __init__(self,x,y):
        self.x,self.y = x,y
    def copy(self):
        return point(self.x,self.y)
a = point(3,4)
b = a.copy()
```


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(单元)里。若`a==b`成立,则只能保留一个(字典的键类似处理);若不成立,可以都保留。一个槽可能放多个对象,查找的时候,根据被查找元素的哈希值 $O(1)$ 时间找到槽,然后再到槽里用`==`顺序查找。哈希值不同的对象,放在不同的槽里面。
- 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。也可以同时重写`__eq__`和 `__hash__`

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__())
#>>230584300921372695 230584300921372695

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__())
#>>-423760875654480603 -423760875654480603
```


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
```

迭代器

如果一个类实现了 `__next__()` 方法和 `__iter__()` 方法，并且 `__iter__()` 方法返回对象自身，则该类的对象就称为“迭代器” (iterator)。迭代器对象通常用于存储一些元素，一般 `__next__()` 方法会用来返回对象中的下一个元素。

迭代器

➤ Python的for循环其实是用while实现

```
for i in a:  
    语句组
```

真实实现:

```
it = iter(a)                #等价于 it = a.__iter__()  
while True:  
    try:  
        i = next(it) #等价于 i = it.__next__()  
        语句组  
    except StopIteration:  
        break
```

iter,next都是Python库函数

迭代器

➤ 迭代器示例

```
class MyRange:
```

```
    def __init__(self, n):
```

```
        self.idx = 0
```

```
        self.n = n
```

```
    def __iter__(self):
```

```
        return self    #返回对象自身
```

```
    def __next__(self):
```

```
        if self.idx < self.n:
```

```
            val = self.idx
```

```
            self.idx += 1
```

```
            return val
```

```
        else:
```

```
            raise StopIteration()    #引发异常
```

迭代器

➤ 迭代器示例

```
for i in MyRange(5): #>>0,1,2,3,4,
    print(i,end=",")
print ([i for i in MyRange(5)]) #>>[0, 1, 2, 3, 4]
x = MyRange(4)
for i in x: #>>0,1,2,3,
    print(i,end=",")
for i in x: #>>无输出
    print(i, end=",") #本句不会被执行
```

类的一些特殊方法

对对象x调用的方法	等价的操作
<code>x.__len__()</code>	<code>len(x)</code> , 一般用于求x的长度
<code>x.__str__()</code>	<code>str(x)</code> , 将x转换成字符串
<code>x.__contains__(y)</code>	<code>y in x</code> , 用于判断y在不在x里面
<code>x.__getitem__(index)</code>	<code>x[index]</code> 根据下标访问x的元素, 下标未必是整数
<code>x.__setitem__(index,item)</code>	<code>x[index] = item</code>

类的一些特殊方法

```
class TaggedList:    #元素带标签的列表
    def __init__(self, data, tags):
        #要求data是个列表或元组, tags是个字符串列表或元组。tags须和data一样长
        self._data = data[:] #成员变量以下划线打头则不易见
        self._tags = tags[:]
        self._tagIdx = {} #该字典用于根据标签检索对应的下标
        for i in range(len(tags)):
            self._tagIdx[tags[i]] = i #标签tags[i]对应下标i
    def __len__(self):
        return len(self._data)
    def __str__(self):
        result = ""
        for i in range(len(self._data)):
            result += \
                self._tags[i]+":"+str(self._data[i])+","
        return result
```


类的一些特殊方法

```
def __contains__(self,x):  
    return x in self._data  
def __getitem__(self,index):  
    if isinstance(index,int): #下标index是整数  
        return self._data[index]  
    elif isinstance(index,str): #下标index是字符串  
        return self._data[self._tagIdx[index]]  
    else: #下标index不是整数也不是字符串，则主动引发异常  
        raise Exception("wrong key type")  
def __setitem__(self,index, item):  
    if isinstance(index,int):  
        self._data[index] = item  
    else:  
        self._data[self._tagIdx[index]] = item
```

类的一些特殊方法

```
a = TaggedList([70,80,90,100],["语文","数学","英语","物理"])
print(len(a),78 in a, 80 in a) #>>4 False True
print(str(a)) #>>语文:70,数学:80,英语:90,物理:100,
print(a[0],a['数学']) #>>70 80    标签也可以作为下标访问元素
a[1] = a['物理'] = 85
print(a) #>>语文:70,数学:85,英语:90,物理:85,
```