目　录

计算余数

删除list中的重复元素

将字符串写成驼峰样式

九宫格问题

致谢

当前文档 《各种算法的Python实现方案（Python and Algorithm）》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-02-21。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常生活、工作和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/Python-and-Algorithm

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

阅前必读

如果要成为一个有一定水平的程序员，算法是必须要会的。算法，让你的程序变得更有灵气。

这里集中了一些算法问题，供看官参考。

也殷切盼望看官能够参与某些算法问题解决的优化。

联系方式: qiwsir (at) gmail.com

官方网站: http://qiwsir.github.io

目录

- 整数list，将偶数放到前面，奇数放到后面，源码
- 斐波那契数列的多种实现方式，源码
- 折半查询查找list中某元素位置，源码
- 排序之用python堆heapq模块
- 排序之归并方法，源码
- 排序之heapq模块详解
- 排序之python sorted性能分析
- 排序之快速排序算法，源码
- 排序算法的比较和选择
- 按照指定字母顺序排序，源码
- 将一个整数分拆为若干整数和，源码，源码2,对前面问题的深化
- 判断一个数是否为素数的多种方法，源码
- 将list中的数字组合成最小的整数，源码，源码2
- 无向图最小生成树Kruskal算法，源码
- 无向图最小生成树的Prim算法，源码
- LUA按照指定字符分割字符串
- 查找字符串中出现最多的字符和个数，源码，源码2
- list中数字的和、最值、均值，源码
- 寻找完全数，源码
- 计算余数，源码
- 删除list中的重复元素，源码
- 将字符串写成驼峰样式，源码
- 九宫格问题，源码

内容还在不断更新增加，请关注算法问题

对一个二维数组的操作

问题

定义一个20*5的二维数组，用来存储某班级20位学员的5门课的成绩；这5门课按存储顺序依次为：core C++，coreJava，Servlet，JSP和EJB。

（1）循环给二维数组的每一个元素赋0~100之间的随机整数。

（2）按照列表的方式输出这些学员的每门课程的成绩。

（3）要求编写程序求每个学员的总分，将其保留在另外一个一维数组中。

（4）要求编写程序求所有学员的某门课程的平均分。

解决（python）

```
#! /usr/bin/env python
#coding:utf-8

from __future__ import division
import random


def score(score_list,course_list,student_num):
    course_num = len(course_list)
```

```
10.
11.     every_score = [[score_list[j][i] for j in range(course_num)]
    for i in range(student_num)]
12.
13.     every_total = [sum(every_score[i]) for i in range(student_num)]
14.
15.     ave_course = [sum(score_list[i])/len(score_list[i]) for i in
    range(len(score_list))]
16.
17.     return (every_score,every_total,ave_course)
18.
19. if __name__=="__main__":
20.
21.     course_list = ["C++","Java","Servlet","JSP","EJB"]
22.     student_num = 20
23.
24.     score_list = [[random.randint(0,100) for i in
    range(student_num)] for j in range(len(course_list))]
25.     for i in range(len(course_list)):
26.         print "score of every one in %s:"%course_list[i]
27.         print score_list[i]
28.
29.     every_score,every_total,ave_one_course =
    score(score_list,course_list,student_num)
30.     print "\n"
31.     print "NEXT IS EVERY ONE SCORE IN EVERY COURSE:"
32.     for name in course_list:
33.         print name,
34.     print "\t"
35.     print every_score
36.     print "\n"
37.     print "every one all score:\t",every_total
38.     print "every course of average score:\t",ave_one_course
```

qiwsir#gmail.com (# to @)

解法 (racket 5.2.1)

```
#lang racket

(define (2d-list)
  (let*
    ([rand-100 (lambda () (random 101))]
     [nth-picker (lambda (n) (lambda (l) (list-ref l n)))]
     [average (lambda (number-list) (exact->inexact (/ (apply + number-list) (length number-list))))]
     [course-list (list "coreC++" "coreJava" "Servlet" "JSP" "EJB")]
     [score-list (for/list ([i 20]) (for/list ([j 5]) (rand-100)))]
     [score-by-course-list (for/list ([i 5]) (list ((nth-picker i) course-list) (map (nth-picker i) score-list)))]
     [score-by-student-list (for/list ([i 20]) (list-ref score-list i))]
     [total-by-student-list (for/list ([i 20]) (apply + (list-ref score-list i)))]
     [average-by-course-list (for/list ([i 5]) (list ((nth-picker i) course-list) (average (map (nth-picker i) score-list))))])
    (for ([i 5])
      (display "score of every one in ")
      (displayln (first ((nth-picker i) score-by-course-list)))
      (displayln (second ((nth-picker i) score-by-course-list))))
    (displayln "")
    (displayln "NEXT IS EVERY ONE SCORE IN EVERY COURSE: ")
    (displayln course-list)
    (for ([i 10])
      (displayln (list-ref score-by-student-list i)))
    (displayln "")
    (displayln "every one all score: ")
    (displayln total-by-student-list)
    (displayln "")
    (displayln "every course of average score: ")
    (displayln average-by-course-list)))

; 调用函数, 正常时应输出类似如下结果
;score of every one in coreC++
```

```
;(12 58 60 28 78 23 34 83 19 83 78 26 51 94 93 74 53 89 8 23)
;... ...
;NEXT IS EVERY ONE SCORE IN EVERY COURSE:
;(coreC++ coreJava Servlet JSP EJB)
;(12 49 75 88 68)
;(58 78 6 88 81)
;... ...
;every one all score:
;(292 311 370 241 289 250 254 258 147 232 271 170 224 248 317 286 246 270 186 212)
;
;every course of average score:
;((coreC++ 53.35) (coreJava 53.9) (Servlet 51.95) (JSP 49.6) (EJB 44.9))
(2d-list)
```

将数组中的元素依次前移

问题

定义一个int型的一维数组，包含10个元素，分别赋值为1~10， 然后将数组中的元素都向前移一个位置，

即，a[0]=a[1],a[1]=a[2],…最后一个元素的值是原来第一个元素的值，然后输出这个数组。

解决(Python)

```
#!/usr/bin/env python
#coding:utf-8


def ahead_one():
    a = [i for i in range(10)]
    b = a.pop(0)
    a.append(b)
    return a

if __name__ =="__main__":
    print ahead_one()
```

解决(racket 5.2.1)

```
#lang racket

```

```
; 定义函数 ahead-one
; 输入为一个整数列表 int-list，假设其长度为 N
; 输出为长度相同的整数列表，其第 N 位的元素为 int-list 的第 1 位的元素，
; 其 1~N-1 位的元素为 int-list 的第 2~N 位的元素
(define (ahead-one int-list)
  (append (rest int-list) (list (first int-list))))

; 函数调用，正常运行时应输出 '(2 3 4 5 6 7 8 9 10 1)
(ahead-one (list 1 2 3 4 5 6 7 8 9 10))
```

求list的平均分并排序

问题

定义一个int型的一维数组，包含40个元素，用来存储每个学员的成绩，循环产生40个0~100之间的随机整数，
(1)将它们存储到一维数组中，然后统计成绩低于平均分的学员的人数，并输出出来。
(2)将这40个成绩按照从高到低的顺序输出出来。

解决（python）

```
#! /usr/bin python
#coding:utf-8


from __future__ import division     #实现精确的除法，例如4/3=1.333333
import random

def make_score(num):
    score = [random.randint(0,100) for i in range(num)]
    return score

def less_average(score):
    num = len(score)
    sum_score = sum(score)
    ave_num = sum_score/num
    less_ave = [i for i in score if i<ave_num]
    return len(less_ave)

if __name__=="__main__":
```

```
20.      score = make_score(40)
21.      print "the number of less average is:",less_average(score)
22.      print "the every socre is[from big to
     small]:",sorted(score,reverse=True)
```

用阿拉伯乘法解决大整数相乘问题

问题

大整数相乘

思路说明

对于大整数计算，一般都要用某种方法转化，否则会溢出。但是python无此担忧了。

Python支持“无限精度”的整数，一般情况下不用考虑整数溢出的问题，而且Python Int类型与任意精度的Long整数类可以无缝转换，超过Int 范围的情况都将转换成Long类型。

例如：

```
>>> 2899887676637907866*178877899278834827738994 3
518725815741570023603416979133706258899163 8L
```

注意：前面的“无限精度”是有引号的。事实上也是有限制的，对于32位的机器，其上限是：2^32-1。真的足够大了。

为什么Python能够做到呢？请有兴趣刨根问底的去看Python的有关源码。本文不赘述。

在其它语言中，通常用“分治法”解决大整数相乘问题。

但是，这里提供一个非常有意思的计算两个整数相乘的方法，算是做为大整数相乘的演示。

两个整数相乘：阿拉伯乘法。关于这个乘法的详细描述，请看：http://ualr.edu/lasmoller/medievalmult.html

解决（Python）

```
#!/usr/bin/env python
#coding:utf-8

#阿拉伯乘法
def arabic_multiplication(num1,num2):
    num_lst1 = [int(i) for i in str(num1)]  #将int类型的123，转化为list类型的[1,2,3]，每个元素都是int类型
    num_lst2 = [int(i) for i in str(num2)]

    #两个list中整数两两相乘
    int_martix = [[i*j for i in num_lst1] for j in num_lst2]

    #将上述元素为数字的list转化为元素类型是str，主要是将9-->'09'
    str_martix = [map(convert_to_str,int_martix[i]) for i in range(len(int_martix))]

    #将上述各个list中的两位数字分开：['01','29','03']-->[0,2,0],[1,9,3]
    martix = [[int(str_martix[i][j][z]) for j in range(len(str_martix[i]))] for i in range(len(str_martix)) for z in range(2)]

    #计算阿拉伯乘法表的左侧开始各项和
    sum_left = summ_left(martix)

    #计算阿拉伯乘法表的底部开始各项和
    sum_end = summ_end(martix)
```

```
23.
24.     #将上述两个结果合并后翻转
25.     sum_left.extend(sum_end)
26.     sum_left.reverse()
27.
28.     #取得各个和的个位的数字（如果进位则加上）
29.     result = take_digit(sum_left)
30.
31.     #翻转结果并合并为一个结果字符串数值
32.     result.reverse()
33.     int_result = "".join(result)
34.     print "%d*%d="%(num1,num2)
35.     print int_result
36.
37.
38. #将int类型转化为str类型，9-->'09'
39.
40. def convert_to_str(num):
41.     if num<10:
42.         return "0"+str(num)
43.     else:
44.         return str(num)
45.
46.
47. #计算阿拉伯乘法表格左侧开始的各项之和
48.
49. def summ_left(lst):
50.     summ = []
51.     x = [i for i in range(len(lst))]
52.     y = [j for j in range(len(lst[0]))]
53.     sx = [i for i in x if i%2==0]
54.     for i in sx:
55.         s=0
56.         j=0
57.         while i>=0 and j<=y[-1]:
58.             s = s+ lst[i][j]
59.             if i%2==1:
60.                 j = j+1
```

```
            else:
                j = j
            i = i-1
        summ.append(s)
    return summ


#计算阿拉伯乘法表格底部开始的各项之和

def summ_end(lst):
    summ=[]
    y = [j for j in range(len(lst[0]))]
    ex = len(lst)-1
    for m in range(len(y)):
        s = 0
        i=ex
        j=m
        while i>=0 and j<=y[-1]:
            s= s+lst[i][j]
            if i%2==1:
                j = j+1
            else:
                j=j
            i = i-1
        summ.append(s)

    return summ

#得到各个元素的个位数，如果是大于10则向下一个进位

def take_digit(lst):
    tmp = 0
    digit_list = []
    for m in range(len(lst)):
        lstm = 0
        lstm = lst[m]+tmp
        if lstm<10:
```

```
99.             tmp = 0
100.            digit_list.append(str(lstm))
101.        else:
102.            tmp = lstm/10
103.            mm = lstm-tmp*10
104.            digit_list.append(str(mm))
105.    return digit_list
106.
107.
108. if __name__=="__main__":
109.    arabic_multiplication(469,37)
```

联系方法

- qiwsir#gmail.com
- https://qiwsir.github.io
- http://weibo.com/qiwsir

二分查找的python模块

问题

二分查找

list.index()无法应对大规模数据的查询，需要用其它方法解决，这里谈的就是二分查找

思路说明

在查找方面，python中有list.index()的方法。例如:

```
>>> a=[2,4,1,9,3]           #list可以是无序，也可以是有序
>>> a.index(4)              #找到后返回该值在list中的位置
1
>>> a.index(5)              #如果没有该值，则报错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 5 is not in list
```

这是python中基本的查找方法，虽然简单，但是，如果由于其时间复杂度为O(n)，对于大规模的查询恐怕是不足以胜任的。二分查找就是一种替代方法。

二分查找的对象是：有序数组。这点特别需要注意。要把数组排好序先。怎么排序，可以参看我这里多篇排序问题的文章。

基本步骤:

1. 从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜素过程结束；
2. 如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。
3. 如果在某一步骤数组为空，则代表找不到。

这种搜索算法每一次比较都使搜索范围缩小一半。时间复杂度：O(logn)

解决(Python)

```
1.  def binarySearch(lst, value,low,high):          #low,high是lst的查找范围
2.      if high < low:
3.          return -1
4.      mid = (low + high)/2
5.      if lst[mid] > value:
6.          return binarySearch(lst, value, low, mid-1)
7.      elif lst[mid] < value:
8.          return binarySearch(lst, value, mid+1, high)
9.      else:
10.         return mid
11.
12. #也可以不用递归方法，而采用循环，如下：
13.
14. def bsearch(l, value):
15.     lo, hi = 0, len(l)-1
16.     while lo <= hi:
17.         mid = (lo + hi) / 2
18.         if l[mid] < value:
19.             lo = mid + 1
20.         elif value < l[mid]:
21.             hi = mid - 1
22.         else:
23.             return mid
```

```
24.     return -1
25.
26. if __name__ == '__main__':
27.     l = range(50)
28.     print binarySearch(l,10,0,49)
29.     print bsearch(l,10)
```

对于python，不能忽视其强大的标准库。经查阅，发现标准库中就有一个模块，名为：bisect。其文档中有这样一句话：

> *This module provides support for maintaining a list in sorted order without having to sort the list after each insertion. For long lists of items with expensive comparison operations, this can be an improvement over the more common approach. The module is called bisect because it uses a basic bisection algorithm to do its work. The source code may be most useful as a working example of the algorithm (the boundary conditions are already right!).*

当我把这段话输入到百度翻译中，天才的百度翻译给我的结果是：

> 这个模块提供支持，维护*list in order for without having to*类法术*the list After each*插入。久*lists of items*以及昂贵的比较操作，这可以改善*over the more common approach*年。*bisect because it is called the*模块基本分割算法用*to do its work*。源代码可能是最有用的工作*example of the*算法（*the*边界条件*are already right*！）。

这就是百度的水平，只可惜在贵国不能用google。

看官就凭借自己的英语水平理解吧。这段话的关键点是在说明：

- 模块接受排序后的列表。
- 本模块同样适用于长列表项。因为它就是用二分查找方法实现的，有兴趣可以看其源码（源码是一个很好的二分查找算法的例子，特别是很好地解决了边界条件极端的问题.）
- 关于bisect模块的更多内容，可以参看官方文档

下面演示这个模块的一个函数

```
1.  from bisect import *
2.
3.  def bisectSearch(lst, x):
4.      i = bisect_left(lst, x)          #bisect_left(lst,x), 得到x在已经
    排序的lst中的位置
5.      if i != len(lst) and lst[i] == x:
6.          return i
7.      raise ValueError
8.
9.  if __name__=="__main__":
10.     lst = sorted([2,5,3,8])
11.     print bisectSearch(lst,5)
```

二叉树查找之基本思想

问题

二叉树查找

思路说明

二叉树查找是一个面对动态数据比较常用的查找算法。本文根据下面地址文章翻译，并根据本人的理解进行适当修改。

原文地址：http://www.laurentluce.com/posts/binary-search-tree-library-in-python/comment-page-1/

二叉树查找的定义

定义内容可以参阅Wikipedia:http://en.wikipedia.org/wiki/Binary_tree

这里是中文的：http://zh.wikipedia.org/wiki/%E4%BA%8C%E5%85%83%E6%90%9C%E5%B0%8B%E6%A8%B9

摘要其中对二叉树的描述：

> 二叉树查找的性质：
>
> 1. 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

> 2. 任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
> 3. 任意节点的左、右子树也分别为二叉查找树。
> 4. 没有键值相等的节点（*no duplicate nodes*）。
>
> 二叉查找树相比于其他数据结构的优势在于查找、插入的时间复杂度较低。为*O(log n)*。二叉查找树是基础性数据结构，用于构建更为抽象的数据结构，如集合、*multiset*、关联数组等。
>
> 虽然二叉查找树的最坏效率是*O(n)*,但它支持动态查询,且有很多改进版的二叉查找树可以使树高为*O(logn)*,如*SBT*,*AVL*,红黑树等.

用python实现二叉树查找

以下面图示的二叉树为例说明查找算法

Node 类

创建一个类，命名为Node，做为二叉树节点结构，其中包括：左枝、右枝、节点数据三个变量。

```
1.  class Node:
2.      """
3.      二叉树左右枝
4.      """
5.      def __init__(self, data):
6.          """
7.          节点结构
8.
9.          """
10.         self.left = None
11.         self.right = None
12.         self.data = data
```

例如创建一个含整数8的节点。因为仅仅创建一个节点，所以左右枝都是None。

```
root = Node(8)
```

这样就得到如下图所示的只有一个节点的树。

插入方法

现在已经有了一棵光秃秃的树，要有枝杈和叶子，就必须用插入数据方法，添加新的节点和数据。

```
def insert(self, data):
    """
    插入节点数据
    """
    if data < self.data:
        if self.left is None:
            self.left = Node(data)
        else:
            self.left.insert(data)
    elif data > self.data:
        if self.right is None:
            self.right = Node(data)
        else:
            self.right.insert(data)
```

承接前面的操作，可以用下面的方式增加树的枝杈和叶子（左右枝以及节点数据）。

```
root.insert(3)
root.insert(10)
root.insert(1)
```

当增加了第二个节点数据3,程序会:

- 第一步，root会调用insert()，其参数是data=3

- 第二步，比较3和8（已有的根节点），3比8小。并且树的左枝还是None，于是就在左边建立一个新的节点。

增加第三个节点数据10,程序会:

- 第一步，跟前面的第一步一样，只不过data=10
- 第二步，发现10大于8,同时右边是None,于是就把它做为右边新建分支的节点数据。

增加第四个节点数据1,程序会:

- 第一步，同前，data=1
- 第二步，1小于8,所以要放在树的左枝;
- 第三步，左枝已经有子节点3,该节点再次调用insert()方法，1小于3,所以1就做为3的子节点，且放在原本就是None的左侧。

如此，就形成了下图的树

继续增加节点数据

```
root.insert(6)
root.insert(4)
root.insert(7)
root.insert(14)
root.insert(13)
```

最终形成下图的树

遍历树

此方法用于查找树中的某个节点，如果找到了，就返回该节点，否则返

回None。为了方便，也返回父节点。

```
def lookup(self, data, parent=None):
    """
    遍历二叉树
    """
    if data < self.data:
        if self.left is None:
            return None, None
        return self.left.lookup(data, self)
    elif data > self.data:
        if self.right is None:
            return None, None
        return self.right.lookup(data, self)
    else:
        return self, parent
```

测试一下，找一找数据为6的节点

```
node, parent = root.lookup(6)
```

调用lookup()后，程序会这么干：

1. 调用lookup()，传递参数data=6,默认parent=None
2. data=6，小于根节点的值8
3. 指针转到根节点左侧，此时:data=6,parent=8,再次调用lookup()
4. data=6大于左侧第一层节点数据3
5. 指针转到3的右侧分支，data=6,parent=3,再次调用lookup()
6. 节点数据等于6,于是返回这个节点和它的父节点3

删除方法

删除节点数据。代码如下：

```
def delete(self, data):
    """
    删除节点
    """
    node, parent = self.lookup(data)        #已有节点
    if node is not None:
        children_count = node.children_count()      #判断子节点数
        if children_count == 0:
            # 如果该节点下没有子节点，即可删除
            if parent.left is node:
                parent.left = None
            else:
                parent.right = None
            del node
        elif children_count == 1:
            # 如果有一个子节点，则让子节点上移替换该节点（该节点消失）
            if node.left:
                n = node.left
            else:
                n = node.right
            if parent:
                if parent.left is node:
                    parent.left = n
                else:
                    parent.right = n
            del node
        else:
            # 如果有两个子节点，则要判断节点下所有叶子
            parent = node
            successor = node.right
            while successor.left:
                parent = successor
                successor = successor.left
            node.data = successor.data
            if parent.left == successor:
                parent.left = successor.right
            else:
                parent.right = successor.right
```

在上述方法中，得到当前节点下的子节点数目后，需要进行三种情况的判断

- 如果没有子节点，直接删除
- 如果有一个子节点，要将下一个子节点上移到当前节点，即替换之
- 如果有两个子节点，要对自己点的数据进行判断，并从新安排节点排序

上述方法中用到了统计子节点数目的方法，代码如下：

```
def children_count(self):
    """
    子节点个数
    """
    cnt = 0
    if self.left:
        cnt += 1
    if self.right:
        cnt += 1
    return cnt
```

例1：删除数据为1的节点，它是3的子节点，1后面没有子节点

```
root.delete(1)
```

例2：删除数据为14的节点，它是10的子节点，它下面有唯一一个子节点13,13替换之。

```
root.delete(14)
```

例3：来个复杂的，删除节点数据为3的节点，它下面有两个节点，而节点6下面又有两个4,7。需要一个临时变量successor，将节点3下面的子节点进行查询，并把小于3下面的第一级子节点6左测节点数据4（该数据一定小于其父节点6）替换当前节点3，维持二叉树结构。如下图：

```
root.delete(3)
```

比较两个二叉树

比较两个二叉树的方法中，只要有一个节点（叶子）与另外一个树的不同，就返回False，也包括缺少对应叶子的情况。

```
def compare_trees(self, node):
    """
    比较两棵树
    """
    if node is None:
        return False
    if self.data != node.data:
        return False
    res = True
    if self.left is None:
        if node.left:
            return False
    else:
        res = self.left.compare_trees(node.left)
    if res is False:
        return False
    if self.right is None:
        if node.right:
            return False
    else:
        res = self.right.compare_trees(node.right)
```

```
22.     return res
```

例如，比较tree(3,8,10)和tree(3,8,11)

```
1.  #root2 是tree(3,8,11)的根
2.  #root 是tree(3,8,10)的根
3.  root.compare_trees(root2)
```

执行上面的代码，程序会这么走：

1. root调用compare_trees()方法
2. root有左侧子节点，调用该节点的compare_trees()
3. 两个左侧子节点比较，返回true
4. 按照前面的过程，比较右侧节点，发现不同，则返回False

打印树

把二叉树按照一定的顺序打印出来。不需要参数了。做法就是先左后右（左小于右）。

```
1.  def print_tree(self):
2.      """
3.      按顺序打印数的内容
4.      """
5.      if self.left:
6.          self.left.print_tree()
7.      print self.data,
8.      if self.right:
9.          self.right.print_tree()
```

操作一下：

```
1.  root.print_tree()
```

输出： 1, 3, 4, 6, 7, 8, 10, 13, 14

包含所有树元素的生成器

创建一个包含所有树元素的生成器，有时候是有必要的。考虑到内存问题，没有必要实时生成所有节点数据列表，而是要每次调用此方法时，它返回的下一个节点的值。为此，使用它返回一个对象，并停止在那里，那么该函数将在下一次调用方法时从那里继续通过yield关键字返回值。在这种情况下，要使用堆栈，不能使用递归。

```
def tree_data(self):
    """
    二叉树数据结构
    """
    stack = []
    node = self
    while stack or node:
        if node:
            stack.append(node)
            node = node.left
        else:
            node = stack.pop()
            yield node.data
            node = node.right
```

举例，通过循环得到树：

```
for data in root.tree_data():
    print data
```

程序会按照先左后右边的原子将数据入栈、出栈，顺序取出值，并返回结果

二叉树查找之python模块

问题

Python中的二叉树查找算法模块

思路说明

二叉树查找算法，在开发实践中，会经常用到。按照惯例，对于这么一个常用的东西，Python一定会提供轮子的。是的，python就是这样，一定会让开发者省心，降低开发者的工作压力。

python中的二叉树模块内容:

- BinaryTree: 非平衡二叉树
- AVLTree: 平衡的AVL树
- RBTree: 平衡的红黑树

以上是用python写的，相面的模块是用c写的，并且可以做为Cython的包。

- FastBinaryTree
- FastAVLTree
- FastRBTree

特别需要说明的是：树往往要比**python**内置的**dict**类慢一些，但是它中的所有数据都是按照某个关键词进行排序的，故在某些情况下是必须使用的。

安装和使用

安装方法

安装环境：

ubuntu12.04, python 2.7.6

安装方法

- 下载源码，地址：https://bitbucket.org/mozman/bintrees/src
- 进入源码目录，看到setup.py文件，在该目录内运行

```
python setup.py install
```

安装成功，ok！下面就看如何使用了。

应用

bintrees提供了丰富的API,涵盖了通常的多种应用。下面逐条说明其应用。

- 引用

如果按照一般模块的思路，输入下面的命令引入上述模块

```
>>> import bintrees
```

错了，这是错的，出现如下警告：(×××不可用，用×××)

```
Warning: FastBinaryTree not available, using Python version BinaryTree.
Warning: FastAVLTree not available, using Python version AVLTree.
Warning: FastRBTree not available, using Python version RBTree.
```

正确的引入方式是：

```
>>> from bintrees import BinaryTree     #只引入了BinartTree
>>> from bintrees import *              #三个模块都引入了
```

- 实例化

看例子：

```
>>> btree = BinaryTree()
>>> btree
BinaryTree({})
>>> type(btree)
<class 'bintrees.bintree.BinaryTree'>
```

- 逐个增加键值对：**.setitem**(k,v) .复杂度O(log(n))(后续说明中，都会有复杂度标示，为了简单，直接标明：O(log(n)).)

看例子：

```
>>> btree.__setitem__("Tom","headmaster")
>>> btree
BinaryTree({'Tom': 'headmaster'})
>>> btree.__setitem__("blog","http://blog.csdn.net/qiwsir")
```

```
>>> btree
BinaryTree({'Tom': 'headmaster', 'blog':
'http://blog.csdn.net/qiwsir'})
```

- 批量添加：.update(E) E是dict/iterable，将E批量更新入btree. O(E*log(n))

看例子：

```
>>> adict = [(2,"phone"),(5,"tea"),(9,"scree"),(7,"computer")]
>>> btree.update(adict)
>>> btree
BinaryTree({2: 'phone', 5: 'tea', 7: 'computer', 9: 'scree', 'Tom':
'headmaster', 'blog': 'http://blog.csdn.net/qiwsir'})
```

- 查找某个key是否存在：**.contains**(k) 如果含有键k，则返回True,否则返回False. O(log(n))

看例子：

```
>>> btree
BinaryTree({2: 'phone', 5: 'tea', 7: 'computer', 9: 'scree', 'Tom':
'headmaster', 'blog': 'http://blog.csdn.net/qiwsir'})
>>> btree.__contains__(5)
True
>>> btree.__contains__("blog")
True
>>> btree.__contains__("qiwsir")
False
>>> btree.__contains__(1)
False
```

- 根据key删除某个key-value：**.delitem**(key), O(log(n))

看例子：

```
>>> btree
BinaryTree({2: 'phone', 5: 'tea', 7: 'computer', 9: 'scree', 'Tom': 'headmaster', 'blog': 'http://blog.csdn.net/qiwsir'})
>>> btree.__delitem__(5)        #删除key=5的key-value,即：5:'tea' 被删除.
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree', 'Tom': 'headmaster', 'blog': 'http://blog.csdn.net/qiwsir'})
```

- 根据key值得到该kye的value: **.getitem**(key)

看例子:

```
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree', 'Tom': 'headmaster', 'blog': 'http://blog.csdn.net/qiwsir'})
>>> btree.__getitem__("blog")
'http://blog.csdn.net/qiwsir'
>>> btree.__getitem__(7)
'computer'
>>> btree._getitem__(5)         #在btree中没有key=5，于是报错。
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'BinaryTree' object has no attribute '_getitem__'
```

- 迭代器: **.iter**()

看例子:

```
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree', 'Tom': 'headmaster', 'blog': 'http://blog.csdn.net/qiwsir'})
>>> aiter = btree.__iter__()
>>> aiter
<generator object <genexpr> at 0xb7416dec>
>>> aiter.next()        #注意：next()一个之后，该值从list中删除
2
```

```
>>> aiter.next()
7
>>> list(aiter)
[9, 'Tom', 'blog']
>>> list(aiter)          #结果是空
[]
>>> bool(aiter)          #but,is True
True
```

- 数的数据长度: **.len()**,返回btree的长度。O(1)

看例子:

```
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree', 'Tom':
'headmaster', 'blog': 'http://blog.csdn.net/qiwsir'})
>>> btree.__len__()
5
```

- 找出key最大的k-v对: **.max()**,按照key排列，返回key最大的键值对。
- 找出key最小的键值对: **.min()**

看例子:

```
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree'})
>>> btree.__max__()
(9, 'scree')
>>> btree.__min__()
(2, 'phone')
```

- 两棵树的关系运算

看例子:

```
>>> other = [(3,'http://blog.csdn.net/qiwsir'),(7,'qiwsir')]
>>> bother = BinaryTree()        #再建一个树
>>> bother.update(other)         #加入数据

>>> bother
BinaryTree({3: 'http://blog.csdn.net/qiwsir', 7: 'qiwsir'})
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree'})

>>> btree.__and__(bother)        #重叠部分部分
BinaryTree({7: 'computer'})

>>> btree.__or__(bother)         #全部
BinaryTree({2: 'phone', 3: 'http://blog.csdn.net/qiwsir', 7:
'computer', 9: 'scree'})

>>> btree.__sub__(bother)        #btree不与bother重叠的部分
BinaryTree({2: 'phone', 9: 'scree'})

>>> btree.__xor__(bother)        #两者非重叠部分
BinaryTree({2: 'phone', 3: 'http://blog.csdn.net/qiwsir', 9:
'scree'})
```

- 输出字符串模样，注意仅仅是输出的模样罢了：.repr()

看例子：

```
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree'})
>>> btree.__repr__()
"BinaryTree({2: 'phone', 7: 'computer', 9: 'scree'})"
```

- 清空树中的所有数据:.clear(),O(log(n))

看例子：

```
>>> bother
```

```
BinaryTree({3: 'http://blog.csdn.net/qiwsir', 7: 'qiwsir'})
>>> bother.clear()
>>> bother
BinaryTree({})
>>> bool(bother)
False
```

- 浅拷贝：.copy(),官方文档上说是浅拷贝，但是我做了操作实现，是下面所示，还不是很理解其“浅”的含义。O(n*log(n))

看例子：

```
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree'})
>>> ctree = btree.copy()
>>> ctree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree'})

>>> btree.__setitem__("github","qiwsir")    #增加btree的数据
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree', 'github': 'qiwsir'})
>>> ctree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree'})     #这是不是在说明属于深拷贝呢？

>>> ctree.__delitem__(7)    #删除ctree的一个数据
>>> ctree
BinaryTree({2: 'phone', 9: 'scree'})
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree', 'github': 'qiwsir'})
```

- 移除树中的一个数据：.discard(key)，这个功能与.**delitem**(key)类似.两者都不反悔值。O(log(n))

看例子：

```
>>> ctree
BinaryTree({2: 'phone', 9: 'scree'})
>>> ctree.discard(2)    #删除后，不返回值，或者返回None
>>> ctree
BinaryTree({9: 'scree'})
>>> ctree.discard(2)    #如果删除的key不存在，也返回None
>>> ctree.discard(3)
>>> ctree.__delitem__(3) #但是，.__delitem__(key)则不同，如果key不存在，会报错。
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.7/site-packages/bintrees/abctree.py", line 264, in __delitem__
  self.remove(key)
  File "/usr/local/lib/python2.7/site-packages/bintrees/bintree.py", line 124, in remove
  raise KeyError(str(key))
  KeyError: '3'
```

- 根据key查找，并返回或返回备用值：.get(key[,d])。如果key在树中存在，则返回value,否则如果有d，则返回d值。O(log(n))

看例子：

```
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree', 'github': 'qiwsir'})
>>> btree.get(2,"algorithm")
'phone'
>>> btree.get("python","algorithm") #没有key='python'的值，返回'algorithm'
'algorithm'
>>> btree.get("python")     #如果不指定第二个参数，若查不到，则返回None
>>>
```

- 判断树是否为空：is_empty().根据树数据的长度，如果数据长度为0,则为空。O(1)

看例子：

```
>>> ctree
BinaryTree({9: 'scree'})
>>> ctree.clear()   #清空数据
>>> ctree
BinaryTree({})
>>> ctree.is_empty()
True
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree', 'github': 'qiwsir'})
>>> btree.is_empty()
False
```

- 根据key、value循环从树中取值：

> *.items([reverse])—按照(key,value)结构取值;*
> *.keys([reverse])—key*
> *.values([reverse])—value. O(n)*
> *.iter_items(s,e[,reverse]—s,e是key的范围，也就是生成在某个范围内的key的迭代器 O(n)*

看例子：

```
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree', 'github': 'qiwsir'})
>>> for (k,v) in btree.items():
...     print k,v
...
2 phone
7 computer
9 scree
github qiwsir
```

```
>>> for k in btree.keys():
...     print k
...
2
7
9
github
>>> for v in btree.values():
...     print v
...
phone
computer
scree
qiwsir
>>> for (k,v) in btree.items(reverse=True):  #反序
...     print k,v
...
github qiwsir
9 scree
7 computer
2 phone

>>> btree
BinaryTree({2: 'phone', 5: None, 7: 'computer', 8: 'eight', 9: 'scree', 'github': 'qiwsir'})
>>> for (k,v) in btree.iter_items(6,9):  #要求迭代6<=key<9的键值对数据
...     print k,v
...
7 computer
8 eight
>>>
```

- 删除数据并返回该值：

> *.pop(key[,d])*，根据*key*删除树的数据，并返回该*value*，但是如果没有，并也指定了备选返回的*d*，则返回*d*，如果没有*d*，则报错；
> *.pop_item()*,在树中随机选择*(key,value)*删除，并返回。

看例子：

```
>>> ctree = btree.copy()
>>> ctree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree', 'github':
'qiwsir'})

>>> ctree.pop(2)    #删除key=2的数据，返回其value
'phone'
>>> ctree.pop(2)    #删除一个不存在的key，报错
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "/usr/local/lib/python2.7/site-
packages/bintrees/abctree.py", line 350, in pop
    value = self.get_value(key)
    File "/usr/local/lib/python2.7/site-
packages/bintrees/abctree.py", line 557, in get_value
    raise KeyError(str(key))
    KeyError: '2'

>>> ctree.pop_item()   #随机返回一个(key,value),并已删除之
(7, 'computer')
>>> ctree
BinaryTree({9: 'scree', 'github': 'qiwsir'})

>>> ctree.pop(7,"sing")    #如果没有，可以返回指定值
'sing'
```

- 查找数据,并返回value: .set_default(key[,d])，在树的数据中查找key,如果存在，则返回该value。如果不存在，当指定了d,则将该（key,d）添加到树内；当不指定d的时候，添加(key,None). O(log(n))

看例子：

```
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 9: 'scree', 'github':
'qiwsir'})
```

```
>>> btree.set_default(7)    #存在则返回
'computer'

>>> btree.set_default(8,"eight")  #不存在，则返回后备指定值，并加入到树
'eight'
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 8: 'eight', 9: 'scree',
'github': 'qiwsir'})

>>> btree.set_default(5)    #如果不指定值，则会加入None
>>> btree
BinaryTree({2: 'phone', 5: None, 7: 'computer', 8: 'eight', 9:
'scree', 'github': 'qiwsir'})

>>> btree.get(2)        #注意，.get(key)与.set_default(key[,d])的区别
'phone'
>>> btree.get(3,"mobile")   #不存在的 key,返回但不增加到树
'mobile'
>>> btree
BinaryTree({2: 'phone', 7: 'computer', 8: 'eight', 9: 'scree',
'github': 'qiwsir'})
```

- 根据key删除值

> *.remove(key)*,删除*(key,value)*
> *.remove_items(keys)*,*keys*是一个*key*组成的*list*,逐个删除树中的对应数据

看例子：

```
>>> ctree
BinaryTree({2: 'phone', 5: None, 7: 'computer', 8: 'eight', 9:
'scree', 'github': 'qiwsir'})
>>> ctree.remove_items([5,6])       #key=6，不存在，报错
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "/usr/local/lib/python2.7/site-
packages/bintrees/abctree.py", line 271, in remove_items
    self.remove(key)
```

```
    File "/usr/local/lib/python2.7/site-packages/bintrees/bintree.py", line 124, in remove
    raise KeyError(str(key))
    KeyError: '6'

>>> ctree
BinaryTree({2: 'phone', 7: 'computer', 8: 'eight', 9: 'scree', 'github': 'qiwsir'})
>>> ctree.remove_items([2,7,'github'])  #按照 列表中顺序逐个删除
>>> ctree
BinaryTree({8: 'eight', 9: 'scree'})
```

以上只是入门的基本方法啦，还有更多内容，请移到文章开头的官方网站。

二叉树之递归方法遍历

问题

用递归方式遍历二叉树

思路说明

遍历二叉树的方法有广度优先和深度优先两类，下面阐述的是深度优先。

以下图的二叉树为例:

先定义三个符号标记:

- 访问结点本身（N）
- 遍历该结点的左子树（L）
- 遍历该结点的右子树（R）

有四种方式:

1. 前序遍历(PreorderTraversal,NLR):先访问根结点，然后遍历其左右子树
2. 中序遍历(InorderTraversal,LNR):先访问左子树，然后访问根节点，再访问右子树
3. 后序遍历(PostorderTraversal,LRN):先访问左右子树，再访

问根结点

4. 层序遍历(levelorderTraversal):按照从上到下的层顺序访问

上面的数，按照以上四种方式遍历，得到的结果依次是:

1. preorder: 1 2 4 7 5 3 6 8 9
2. inorder: 7 4 2 5 1 8 6 9 3
3. postorder: 7 4 5 2 8 9 6 3 1
4. level-order: 1 2 3 4 5 6 7 8 9

下面用递归的方式，解决此题。

解决(Python)

```
#! /usr/bin/env python
#coding:utf-8

from collections import namedtuple
from sys import stdout

Node = namedtuple('Node', 'data, left, right')
tree = Node(1,
            Node(2,
                 Node(4,
                      Node(7, None, None),
                      None),
                 Node(5, None, None)),
            Node(3,
                 Node(6,
                      Node(8, None, None),
                      Node(9, None, None)),
                 None))


#前序（pre-order, NLR）

```

```
23. def preorder(node):
24.     if node is not None:
25.         print node.data,
26.         preorder(node.left)
27.         preorder(node.right)
28.
29.
30. #中序（in-order, LNR）
31.
32. def inorder(node):
33.     if node is not None:
34.         inorder(node.left)
35.         print node.data,
36.         inorder(node.right)
37.
38.
39. #后序（post-order, LRN）
40.
41. def postorder(node):
42.     if node is not None:
43.         postorder(node.left)
44.         postorder(node.right)
45.         print node.data,
46.
47.
48. #层序（level-order）
49.
50. def levelorder(node, more=None):
51.     if node is not None:
52.         if more is None:
53.             more = []
54.         more += [node.left, node.right]
55.         print node.data,
56.     if more:
57.         levelorder(more[0], more[1:])
58.
59. if __name__=="__main__"
60.     print '  preorder: ',
```

```
61.     preorder(tree)
62.     print '\t\n   inorder: ',
63.     inorder(tree)
64.     print '\t\n postorder: ',
65.     postorder(tree)
66.     print '\t\nlevelorder: ',
67.     levelorder(tree)
68.     print '\n'
```

兑换硬币问题之贪婪算法解决

问题:

以人民币的硬币为例，假设硬币数量足够多。要求将一定数额的钱兑换成硬币。要求兑换硬币数量最少。

思路说明:

这是用贪婪算法的典型应用。在本例中用python来实现，主要思想是将货币金额除以某硬币单位，然后去整数，即为该硬币的个数；余数则做为向下循环计算的货币金额。

这个算法的问题在于，得出来的结果不一定是最有结果。比如，硬币单位是[1,4,6],如果将8兑换成硬币，按照硬币数量最少原则，应该兑换成为两个4（单位）的硬币，但是，按照本算法，得到的结果是一个6单位和两个1单位的硬币。这也是本算法的局限所在。所谓贪婪算法，本质就是只要找出一个结果，不考虑以后会怎么样。

解决(Python)

```
1. #!/usr/bin/env python
2. #coding:utf-8
3. 
4. 
5. def change_coin(money):
```

```
    coin = [1,2,5,10,20,50,100]      #1分，2分，5分，1角，2角，5角，1元
    coin.sort(reverse=True)
    money = money*100                #以分为单位进行计算
    change = {}

    for one in coin:
        num_coin = money//one        #除法，取整，得到该单位硬币的个数
        if num_coin>0:
            change[one]=num_coin
        num_remain = money%one       #取余数，得到剩下的钱数
        if num_remain==0:
            break
        else:
            money = num_remain
    return change

if __name__=="__main__":
    money = 3.42
    num_coin = change_coin(money)
    result = [(key,num_coin[key]) for key in
sorted(num_coin.keys())]
    print "You have %s RMB"%money
    print "I had to change you:"
    print "    Coin    Number"
    for i in result:
        if i[0]==100:
            print "Yuan    %d    %d"%(i[0]/100,i[1])
        elif i[0]<10:
            print "Fen    %d    %d"%(i[0],i[1])
        else:
            print "Jiao    %d    %d"%(i[0]/10,i[1])

#执行结果
#You have 3.42 RMB
#I had to change you:
#    Coin    Number
#    Fen    2    1
#    Jiao    2    2
```

```
43. #    Yuan    1    3
```

解决2(Python)

以下方法，以动态方式，提供最小的硬币数量。避免了贪婪方法的问题。

```
1.  def coinChange(centsNeeded, coinValues):
2.      minCoins = [[0 for j in range(centsNeeded + 1)] for i in
    range(len(coinValues))]
3.      minCoins[0] = range(centsNeeded + 1)
4.
5.      for i in range(1,len(coinValues)):
6.          for j in range(0, centsNeeded + 1):
7.              if j < coinValues[i]:
8.                  minCoins[i][j] = minCoins[i-1][j]
9.              else:
10.                 minCoins[i][j] = min(minCoins[i-1][j], 1 +
    minCoins[i][j-coinValues[i]])
11.     return minCoins[-1][-1]
```

索引查找概述

问题

索引查找

索引查找的定义，来源百度百科

> 索引查找是在索引表和主表(即线性表的索引存储结构)上进行的查找。
> 索引查找的过程是：
>
> - 首先根据给定的索引值*K1*，在索引表上查找出索引值等于*K1*的索引项，以确定*K1*对应的子表在主表中的开始位置和长度，
> - 然后再根据给定的关键字*K2*，在对应的子表中查找出关键字等于*K2*的元素(结点)。

思路说明

对于一个list或者dictionary类型的数据，python有专门的内置函数index()进行索引查找，当然，这个查找的过程完全由python自己完成，不需要我们重写。

list索引: list.index(x)

python的官方解释是：

> *Return the index in the list of the first item whose value is x.*
> *It is an error if there is no such item.*

翻译：返回list中的第一个值为x的元素索引，如果找不到返回错误。

例如：

```
>>> alist = ['I','am','a','human','you','are','a','programmer']
>>> alist.index('a')        #注意：alist中有两个"a",但是只返回第一个索引值
2
>>> alist.index('am')
1
>>> alist.index('he')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ValueError: 'he' is not in list
```

这个索引查找的复杂度是O(n)

对于较小的数据，list.index(x)足矣，但是，如果对象数据量比较大了，这个就有点小马拉大车的感觉了，怎么办？熟悉本博客风格的朋友肯定会想到，绝对不是让我们自己动手写一个索引查找的东西，虽然写一个不是不可以，但是本着“拿来主义”的精神，一定要先查找一下，看看python是否已经为我们做好了轮子？

Whoosh：全文索引

把官方文档的一段话拿过来：

> *Whoosh is a library of classes and functions for indexing text and then searching the index. It allows you to develop custom search engines for your content. For example, if you were creating blogging software, you could use Whoosh to add a search function to allow users to search blog entries.*

简单翻译：

> *Whoosh*是一个索引文本和搜索的库，允许你为你的内容设置自定义搜索引擎。比如如果

> 创建一个博客，可以用*whoosh*为它添加一个搜索功能，以便用户来搜索博文。

python就是这么善解人意，就是这么高大上。

这个东西怎么用？

Whoosh安装

在ubuntu环境下，如果已经有pip或者easy_install，只需要直接运行：

```
1. $ sudo easy_install Whoosh
```

或者：

```
1. $ sudo pip install Whoosh
```

即可轻松安装。windows的朋友，是不是用linux的优势在这里体现出来呢？请用。

当然，也可以到官方下载源码进行安装

安装之后，输入下面的内容，如果不报错，就说明已经安装成功(一个就可以检验)。

```
1. >>> from whoosh.fields import *
2. >>> from whoosh.index import create_in
```

Whoosh应用

在官方文档上，有完整的应用讲述：https://pythonhosted.org/Whoosh/quickstart.html#a-quick-introduction

此外，有几篇文章，是不错的，列在这里备查

- http://blog.sina.com.cn/s/blog_819588bc0101co4b.html
- http://www.cnblogs.com/chang/archive/2013/01/10/2855321.html
- http://ashin.sinaapp.com/article/118/

除了上述内容，在其源码存放地点，也有一些问题回答：https://bitbucket.org/mchaput/whoosh/overview

索引查找之Whoosh使用详解

问题

Whoosh是python中解决索引查找的模块，在讨论索引查找的文章已经对有关索引查找进行了阐述，此处详细说明Whoosh模块的应用。

思路说明

Whoosh的安装

这里有详细内容

whoosh的使用步骤

whoosh在应用上划分三个步骤：

1. 建立索引和模式对象
2. 写入索引文件

3. 搜索

下面依次阐述各步骤

建立索引和模式对象

建立索引模式

使用Whoosh的第一步就是要建立索引对象。首先要定义索引模式，以字段的形式列在索引中。例如：

```
1. >>> from whoosh.fields import *
2. >>> schema = Schema(title=TEXT, path=ID, content=TEXT)
```

title/path/content就是所谓的字段。每个字段对应索引查找目标文件的一部分信息，上面的例子中就是建立索引的模式：索引内容包括title/path/content。一个字段建立了索引，意味着它能够被搜索；也能够被存储，意味着返回结果。例如上面的例子，可以写成：

```
1. >>> schema = Schema(title=TEXT(stored=True), path=ID(stored=True),
   content=TEXT)
```

这里在某些字段后面添加了(stored=True)，意味着将返回该字段的搜索结果。

以上就建立好了索引模式，不需要重复建立索引模式，因为一旦此模式建立，将随索引保存。

在生产过程中，如果你愿意，还可以建立一个类用于建立索引模式。如下例子：

```
1. from whoosh.fields import SchemaClass, TEXT, KEYWORD, ID, STORED
2.
```

```
class MySchema(SchemaClass):
    path = ID(stored=True)
    title = TEXT(stored=True)
    content = TEXT
    tags = KEYWORD
```

索引字段类型

在上例中，title=TEXT，title是字段名称，后面的TEXT是该字段的类型。这两个分别说明了索引内容和查找对象类型。whoosh有如下字段类型，供建立所以模式使用。

- whoosh.fields.ID:仅能为一个单元值，即不能分割为若干个词，通常用于诸如文件路径，URL，日期，分类。
- whoosh.fields.STORED:该字段随文件保存，但是不能被索引，也不能被查询。常用于显示文件信息。
- whoosh.fields.KEYWORD:用空格或者逗号（半角）分割的关键词，可被索引和搜索。为了节省空间，不支持词汇搜索。
- whoosh.fields.TEXT:文件的文本内容。建立文本的索引并存储，支持词汇搜索。
- whoosh.fields.NUMERIC:数字类型，保存整数或浮点数。
- whoosh.fields.BOOLEAN:布尔类值
- whoosh.fields.DATETIME:时间对象类型

关于索引字段类型的更多内容，请看这里.

建立索引存储目录

索引模式建立之后，还要建立索引存储目录。如下：

```
import os.path
from whoosh.index import create_in
from whoosh.index import open_dir
```

```
4.
5.  if not os.path.exists('index'):      #如果目录index不存在则创建
6.      os.mkdir('index')
7.  ix = create_in("index",schema)       #按照schema模式建立索引目录
8.  ix = open_dir("index")               #打开该目录一遍存储索引文件
```

上例中，用create_in创建一个具有前述索引模式的索引存储目录对象，所有的索引将被保存在该目录（index）中。

之后，用open_dir打开这个目录。

第一步到此结束。

把上面的代码整理一下，供参考：

```
1.  import os.path
2.
3.  from whoosh import fields
4.  from whoosh import index
5.
6.  schema = fields.Schema(title=TEXT(stored=True),
    path=ID(stored=True), content=TEXT)
7.
8.  if not os.path.exists("index"):
9.      os.mkdir("index")
10.
11. ix = index.create_in("index",schema)
12. ix = index.open_dir("index")
```

写索引文件

下面开始写入索引内容，过程如下：

```
1.  writer = ix.writer()
2.  writer.add_document(title=u"my document", content=u"this is my
    document", path=u"/a", tags=u"firlst short",
```

```
    icon=u"/icons/star.png")
3.  writer.add_document(title=u"my second document", content=u"this is
    my second document", path=u"/b", tags=u"second short",
    icon=u"/icons/sheep.png")
4.  writer.commit()
```

特别注意：

- 字段的值必须是unicode类型
- 不是每个字段都必须赋值

更多的内容，请参考：如何索引文件官方文档

搜索

开始搜索，需要新建立一个对象，如：

```
1.  searcher = ix.searcher()
```

一般来讲，不是这么简单地，建立对象相当于开始搜索，完事之后要关闭，所以在实战中，常常写成：

```
1.  withe ix.searcher() as searcher:
2.      (do somthing)
```

或者写成（与上面的等效）：

```
1.  try:
2.      searcher = ix.searcher()
3.      (do somthing)
4.  finally:
5.      searcher.close()
```

接下来就开始搜索了，以搜索content为例：

```
from whoosh.qparser import QueryParser
with ix.searcher() as searcher:
    query = QueryParser("content",ix.schema).parse("second")
    result = searcher.search(query)
    results[0]
```

返回显示：

```
{"title":u"my second document","path":u"/a"}
```

前面已经将上述两个字段设置为stored=True.

中文分词

中文分词中，结巴分词是不错的。以下两个内容解决中文分析问题：

- 结巴分词
- whoosh and 结巴分词

删除一个字符串中连续一次以上出现的空格

问题

删除一个字符串中连续超过一次的空格。

解决（Python）

```
#! /usr/bin/env python
#coding:utf-8

def del_space(string):
    split_string = string.split(" ")    #以空格为分割，生成list，list中如果含有空格，则该空格是连续空格中的后一个
    string_list = [i for i in split_string if i!=""]    #去掉空格，生成list
    result_string = " ".join(string_list)
    return result_string

if __name__=="__main__":
    one_str = "Hello,  I am  Qiwsir."
    string = del_space(one_str)
    print one_str
    print string
```

最短路径问题的Dijkstra算法

问题

最短路径问题的Dijkstra算法

是由荷兰计算机科学家艾兹赫尔·戴克斯特拉提出。迪科斯彻算法使用了广度优先搜索解决非负权有向图的单源最短路径问题，算法最终得到一个最短路径树。该算法常用于路由算法或者作为其他图算法的一个子模块。

这个算法的python实现途径很多，网上能够发现不少。这里推荐一个我在网上看到的，本来打算自己写，看了这个，决定自己不写了，因为他的已经太好了。

解决（Python）

```
1.  #!/usr/bin/env python
2.  #coding:utf-8
3.
4.  # Dijkstra's algorithm for shortest paths
5.  # David Eppstein, UC Irvine, 4 April 2002
6.  # code
    source:http://www.algolist.com/code/python/Dijkstra%27s_algorithm
7.
8.  from priodict import priorityDictionary
9.
10. def Dijkstra(G,start,end=None):
11.     D = {}  # dictionary of final distances
12.     P = {}  # dictionary of predecessors
```

```
13.     Q = priorityDictionary()   # est.dist. of non-final vert.
14.     Q[start] = 0
15.
16.     for v in Q:
17.         D[v] = Q[v]
18.
19.         if v == end: break
20.
21.         for w in G[v]:
22.             vwLength = D[v] + G[v][w]
23.             if w in D:
24.                 if vwLength < D[w]:
25.                     raise ValueError, "Dijkstra: found better path
    to already-final vertex"
26.
27.                 elif w not in Q or vwLength < Q[w]:
28.                     Q[w] = vwLength
29.                     P[w] = v
30.
31.         return (D,P)
32.
33. def shortestPath(G,start,end):
34.     D,P = Dijkstra(G,start,end)
35.     Path = []
36.     while 1:
37.
38.         Path.append(end)
39.         if end == start: break
40.         end = P[end]
41.
42.     Path.reverse()
43.     return Path
```

整数list，将偶数放到前面，奇数放到后面

问题

一个数组由若干个整数组成，现要求：将偶数放到前面，奇数放到后面，并输出数组。

解决（python）

```
#coding:utf-8
is_odd_number = lambda data:(data%2!=0)

def odd_even_sort(lst):
    """利用list conprehension"""
    tmp_list1 = [item for item in lst if is_odd_number(item)]
    tmp_list2 = [item for item in lst if not is_odd_number(item)]

test_lst = [7,9,12,5,4,9,8,3,12,89]

print (odd_even_sort(test_lst))
```

本问题由黄老师提供并解决，他的微博

解决（python）

```
def odd(x):return x%2==1    #判断是否为奇数，是则返回true
def even(x):return x%2==0

if __name__=="__main__":
    test_lst = [7,9,12,5,4,9,8,3,12,89]
    print filter(even,test_lst)+filter(odd,test_lst)    #利用filter函数
```

解决（racket 5.2.1）

```
#lang racket

; 定义函数 odd-even-separator
; 输入一个由整数构成的列表
; 输出一个新的列表，其元素取自输入的列表
; 假设输入列表长度为 N，列表元素中有 k 个偶数，N-k 个奇数
; 那么输出的列表中，前 k 个元素就是输入列表中的 k 个偶数
; 后 N-k 个元素就是输入列表中的 N-k 个奇数.
(define (odd-even-separator num-array)
  (let*
      ([odd-arr (filter odd? arr1)]   ; 取出全部奇数形成新列表
       [even-arr (filter even? arr1)] ; 取出全部偶数形成新列表
       [separated-arr
        (append even-arr odd-arr)])   ; 把两个新列表连接起来
    (displayln separated-arr)))       ; 打印到标准输出

; 函数调用，正常运行后，应该显示 (2 6 4 1 3 5 7 9)
(odd-even-separator '(1 3 2 6 5 7 9 4))
```

斐波那契数列的多种实现方式

问题

费波那契数列（意大利语：Successione di Fibonacci），又译费波拿契数、斐波那契数列、斐波那契数列、黄金分割数列。

在数学上，费波那契数列是以递归的方法来定义：

```
1. F0 = 0     (n=0)
2. F1 = 1    (n=1)
3. Fn = F[n-1]+ F[n-2](n=>2)
```

关于Fibonacci的精彩解释，请看下列视频：

TED-神奇的斐波那契数列

如果要查看文字解释，请看维基百科词条：斐波那契数列

思路说明

几乎所有的高级语言都要拿Fibonacci数列为例子，解释递归、循环等概念。这里，我要用Python来演示一下，各种不同的写法，供参

考。

解决（python）

递归—按照定义直接写

这种方法不是一个好方法，因为它的开销太大，比如计算fib1(100)，就需要耐心等待较长一段时间了。所以，这是一种不实用的方法。但是，因为简单，列为第一种。

```
1. def fib1(n):
2.     if n==0:
3.         return 0
4.     elif n==1:
5.         return 1
6.     else:
7.         return fib1(n-1) + fib1(n-2)
```

递归，进行初始化

fib1的慢，就是因为每次都要计算前面已经算过的项目.这里将上述算法进行稍微改进。速度快了很多。

```
1. memo = {0:0, 1:1}
2. def fib2(n):
3.     if not n in memo:
4.         memo[n] = fib2(n-1)+fib2(n-2)
5.     return memo[n]
```

迭代

```
1. def fib3(n):
```

```
    a, b = 0, 1
    for i in range(n):
        a, b = b, a+b
    return a
```

直接理论数学结论

在维基百科的词条 里面，已经列出了不同形式的Fibonacci数列的数学结果，可以直接将这些结果拿过来，通过程序计算，得到斐波那契数。此类程序，本文略。

这种方法来自网络

```
print('!* Fibonacci Sequence python \n')
def Fibonacci_Series():
    x = input('Enter Series length to print fibonacci sequence')

    d,e=0,1
    a = []
    a.append(d)
    a.append(e)
    while(x!=2):
        c = d + e
        d = e
        e = c
        a.append(c)
        x = x -1
    print(a)
```

折半查询查找list中某元素位置

问题

查找某个值在list中的位置

解决思路

可以用折半查询的方法解决此问题。

解决（Python）

```
#! /usr/bin/env python
#coding:utf-8

#折半查找某个元素在list中的位置

def half_search(lst,value,left,right):
    length = len(lst)
    while left<right:
        middle = (right-left)/2
        if lst[middle]>value:
            right = middle-1
        elif lst[middle]<value:
            left = middle+1
        else:
            return middle
```

```
16.
17. if __name__=="__main__":
18.     lst=sorted([2,4,5,9])       #折半算法中list要进行排序
19.     length = len(lst)
20.     left = 0
21.     right = length-1
22.     value =4
23.     result = half_search(lst,value,left,right)
24.     if result:
25.         print result
26.     else:
27.         print "There is no the value that you want to search."
```

再思考

对于上面的折半方法，在python中，可以通过一个函数实现

```
1. lst = sorted([2,4,5,9])       #这里进行排序，主要是为了得到与上面方法一样的结果。事实上，list.index()可以针对任何list操作，不一定非要排序
2. result = lst.index(4)
```

此外，如果遇到list中有多个相同的元素，应该如何将这些元素的位置都查询出来呢？下面的方法是用python实现。

```
1. def find_value_location(lst,value):
2.     result = [i for i in range(len(lst)) if value==lst[i]]
3.     return result
```

qiwsir#gmail.com

排序之归并方法

问题

归并排序

思路说明

归并操作过程：

1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置
3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置
4. 重复步骤3直到某一指针达到序列尾
5. 将另一序列剩下的所有元素直接复制到合并序列尾

上述说法是理论表述，下面用一个实际例子说明：

例如一个无序数组[6,2,3,1,7]

首先将这个数组通过递归方式进行分解，直到：[6],[2],[3],[1],[7]

然后开始合并排序，也是用递归的方式进行：

1. 两个两个合并排序，得到：[2,6],[1,3],[7]

2. 上一步中，其实也是按照本步骤的方式合并的，只不过由于每个list中一个数，不能完全显示过程。下面则可以完全显示过程。

   初始:
   a = [2,6]
   b = [1,3]
   c = []
   第1步，顺序从a,b中取出一个数字: 2,1
   比较大小后放入c中，并将该数字从原list中删除，结果是:
   a = [2,6]
   b = [3]
   c = [1]
   第2步，继续从a,b中按照顺序取出数字，也就是重复上面步骤，这次是: 2,3
   比较大小后放入c中，并将该数字从原list中删除，结果是:
   a = [6]
   b = [3]
   c = [1,2]
   第3步,再重复前边的步骤,结果是:
   a = [6]
   b = []
   c = [1,2,3]
   最后一步，将6追加到c中，结果形成了:
   a = []
   b = []
   c = [1,2,3,6]

3. 通过反复应用上面的流程，实现[1,2,3,6]与[7]的合并

4. 最终得到排序结果[1,2,3,6,7]

本文列举了三种python的实现方法。

解决(Python)

```python
#! /usr/bin/env python
#coding:utf-8

#方法1：将前面讲述的过程翻译过来了，略先拙笨

def merge_sort(seq):
    if len(seq) ==1:
        return seq
    else:
        middle = len(seq)/2
        left = merge_sort(seq[:middle])
        right = merge_sort(seq[middle:])

        i = 0   #left 计数
        j = 0   #right 计数
        k = 0   #总计数

        while i < len(left) and j < len(right):
            if left[i] < right [j]:
                seq[k] = left[i]
                i +=1
                k +=1
            else:
                seq[k] = right[j]
                j +=1
                k +=1

        remain = left if i<j else right
        r = i if remain ==left else j

        while r<len(remain):
            seq[k] = remain[r]
            r +=1
```

```
34.             k +=1
35.
36.         return seq
37.
38. #方法2：在按照顺序取数值方面，应用了list.pop()方法，代码更紧凑简洁
39. #此方法来[自维基百科：归并操作]
    (http://zh.wikipedia.org/zh/%E5%BD%92%E5%B9%B6%E6%8E%92%E5%BA%8F)
40.
41. def merge_sort(lst):   #此方法来自维基百科：
    http://zh.wikipedia.org/zh/%E5%BD%92%E5%B9%B6%E6%8E%92%E5%BA%8F
42.     if len(lst) <= 1:
43.         return lst
44.
45.     def merge(left, right):
46.         merged = []
47.
48.         while left and right:
49.             merged.append(left.pop(0) if left[0] <= right[0] else
    right.pop(0))
50.
51.         while left:
52.             merged.append(left.pop(0))
53.
54.         while right:
55.             merged.append(right.pop(0))
56.
57.         return merged
58.
59.     middle = int(len(lst) / 2)
60.     left = merge_sort(lst[:middle])
61.     right = merge_sort(lst[middle:])
62.     return merge(left, right)
63.
64. #方法3：原来在python的模块heapq中就提供了归并排序的方法，只要将分解后的结果导
    入该方法即可
65. #强大。
66. #以下方法来自[resettacode]
    (http://rosettacode.org/wiki/Sorting_algorithms/Merge_sort#Python),
```

```
    并稍作修改
67.
68. from heapq import merge
69.
70. def merge_sort(seq):
71.     if len(seq) <= 1:
72.         return m
73.     else:
74.         middle = len(seq)/2
75.         left = merge_sort(seq[:middle])
76.         right = merge_sort(seq[middle:])
77.         return list(merge(left, right))         #heapq.merge()
78.
79. if __name__=="__main__":
80.     seq = [1,3,6,2,4]
81.     print merge_sort(seq)
```

排序之heapq模块详解

python中的堆排序peapq模块

heapq模块实现了python中的堆排序，并提供了有关方法。让用Python实现排序算法有了简单快捷的方式。

heapq的官方文档和源码：8.4.heapq-Heap queue algorithm

下面通过举例的方式说明heapq的应用方法

实现堆排序

```
#! /usr/bin/evn python
#coding:utf-8

from heapq import *

def heapsort(iterable):
    h = []
    for value in iterable:
```

```
            heappush(h,value)
        return [heappop(h) for i in range(len(h))]

if __name__=="__main__":
        print heapsort([1,3,5,9,2])
```

heappush()

heapq.heappush(heap, item):将item压入到堆数组heap中。如果不进行此步操作，后面的heappop()失效

heappop()

heapq.heappop(heap):从堆数组heap中取出最小的值，并返回。

```
>>> h=[]                        #定义一个list
>>> from heapq import *         #引入heapq模块
>>> h
[]
>>> heappush(h,5)                   #向堆中依次增加数值
>>> heappush(h,2)
>>> heappush(h,3)
>>> heappush(h,9)
>>> h                               #h的值
[2, 5, 3, 9]
>>> heappop(h)                      #从h中删除最小的，并返回该值
2
>>> h
[3, 5, 9]
>>> h.append(1)                     #注意，如果不是压入堆中，而是通过append追加一个数值
>>> h                               #堆的函数并不能操作这个增加的数值，或者说它堆对来讲是不存在的
[3, 5, 9, 1]
>>> heappop(h)                      #从h中能够找到的最小值是3,而不是1
3
```

```
>>> heappush(h,2)                    #这时，不仅将2压入到堆内，而且1也进入了堆。
>>> h
[1, 2, 9, 5]
>>> heappop(h)                       #操作对象已经包含了1
1
```

heapq.heappushpop(heap, item)

是上述heappush和heappop的合体，同时完成两者的功能.注意：相当于先操作了heappush(heap,item),然后操作heappop(heap)

```
>>> h
[1, 2, 9, 5]
>>> heappop(h)
1
>>> heappushpop(h,4)                 #增加4同时删除最小值2并返回该最小值，与下列操作等同：
2                                    #heappush(h,4),heappop(h)
>>> h
[4, 5, 9]
```

heapq.heapify(x)

x必须是list，此函数将list变成堆，实时操作。从而能够在任何情况下使用堆的函数。

```
>>> a=[3,6,1]
>>> heapify(a)                       #将a变成堆之后，可以对其操作
>>> heappop(a)
1
>>> b=[4,2,5]                        #b不是堆，如果对其进行操作，显示结果如下
>>> heappop(b)                       #按照顺序，删除第一个数值并返回,不会从中挑选出最小的
4
>>> heapify(b)                       #变成堆之后，再操作
```

```
>>> heappop(b)
2
```

heapq.heapreplace(heap, item)

是heappop(heap)和heappush(heap,item)的联合操作。注意，与heappushpop(heap,item)的区别在于，顺序不同，这里是先进行删除，后压入堆

```
>>> a=[]
>>> heapreplace(a,3)            #如果list空，则报错
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: index out of range
>>> heappush(a,3)
>>> a
[3]
>>> heapreplace(a,2)            #先执行删除（heappop(a)->3),再执行加入（heappush(a,2))
3
>>> a
[2]
>>> heappush(a,5)
>>> heappush(a,9)
>>> heappush(a,4)
>>> a
[2, 4, 9, 5]
>>> heapreplace(a,6)            #先从堆a中找出最小值并返回，然后加入6
2
>>> a
[4, 5, 9, 6]
>>> heapreplace(a,1)            #1是后来加入的，在1加入之前，a中的最小值是4
4
>>> a
[1, 5, 9, 6]
```

heapq.merge(*iterables)

举例:

```
>>> a=[2,4,6]
>>> b=[1,3,5]
>>> c=merge(a,b)
>>> list(c)
[1, 2, 3, 4, 5, 6]
```

在归并排序中详细演示了本函数的使用方法。

heapq.nlargest(n, iterable[, key]),heapq.nsmallest(n, iterable[, key])

获取列表中最大、最小的几个值。

```
>>> a
[2, 4, 6]
>>> nlargest(2,a)
[6, 4]
```

排序之python sorted性能分析

python的排序详解

排序，在编程中经常遇到的算法，我也在几篇文章中介绍了一些关于排序的算法。有的高级语言内置了一些排序函数。本文讲述Python在这方面的工作。供使用python的程序员们参考，也让没有使用python的朋友了解python。领略一番“生命有限，请用Python”的含义。

内置函数sorted()/list.sort()的使用

简单应用

python对list有一个内置函数：sorted(),专门用于排序。举例：

```
>>> a=[5,3,6,1,9,2]
>>> sorted(a)       #a经过sorted之后，得到一个排序结果
[1, 2, 3, 5, 6, 9]  #但是，原有的a并没有受到影响
>>> a
[5, 3, 6, 1, 9, 2]
```

也可以使用list.sort()来进行上述操作。

```
>>> a.sort()
>>> a               #注意这里，经过list.sort()之后，原有
[1, 2, 3, 5, 6, 9]  #a的顺序已经发生变化，与上述不同之处。
```

sorted和list.sort()的区别:list.sort()只能对list类型进行排序。如下:

```
>>> b_dict={1:'e',3:'m',9:'a',5:'e'}
>>> b_dict.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  AttributeError: 'dict' object has no attribute 'sort'
```

而sorted则不然，看例子:

```
>>> b_dict
{1: 'e', 3: 'm', 5: 'e', 9: 'a'}
>>> sorted(b_dict)
[1, 3, 5, 9]
```

sorted之后，上述对dictinoary中，将key值取出并排序，返回list类型的排序结果。

按照指定关键词排序

在list.sort()和sorted中，都可以根据指定的key值排序。例如:

sorted的例子:

```
>>> qw="I am Qiwsir you can read my articles im my blog".split()
>>> qw
['I', 'am', 'Qiwsir', 'you', 'can', 'read', 'my', 'articles', 'im', 'my', 'blog']
>>> sorted(qw,key=str.lower)        #按照字母升序排列
['am', 'articles', 'blog', 'can', 'I', 'im', 'my', 'my', 'Qiwsir', 'read', 'you']
```

list.sort()的例子:

```
>>> qw
['I', 'am', 'Qiwsir', 'you', 'can', 'read', 'my', 'articles', 'im', 'my', 'blog']
>>> qw.sort(key=str.lower)
>>> qw
['am', 'articles', 'blog', 'can', 'I', 'im', 'my', 'my', 'Qiwsir', 'read', 'you']
```

此外，key还可以接收函数的单一返回值，按照该值排序。例如：

```
>>> name_mark_age = [('zhangsan','A',15),('LISI','B',14),('WANGWU','A',16)]
>>> sorted(name_mark_age, key = lambda x: x[2])     #根据年龄排序
[('LISI', 'B', 14), ('zhangsan', 'A', 15), ('WANGWU', 'A', 16)]

>>> sorted(name_mark_age, key = lambda x: x[1])     #根据等级排序
[('zhangsan', 'A', 15), ('WANGWU', 'A', 16), ('LISI', 'B', 14)]

>>> sorted(name_mark_age, key = lambda x: x[0])     #根据姓名排序
[('LISI', 'B', 14), ('WANGWU', 'A', 16), ('zhangsan', 'A', 15)]
```

除了上述方式，python中还提供了一个选择循环选择指定元组值的模块

```
>>> from operator import itemgetter    #官方文档：https://docs.python.org/2/library/operator.html#module-operator
>>> name_mark_age.append(('zhaoliu','B',16))
>>> name_mark_age
[('zhangsan', 'A', 15), ('LISI', 'B', 14), ('WANGWU', 'A', 16), ('zhaoliu', 'B', 16)]

>>> sorted(name_mark_age,key=itemgetter(2))     #按照年龄排序
[('LISI', 'B', 14), ('zhangsan', 'A', 15), ('WANGWU', 'A', 16), ('zhaoliu', 'B', 16)]

>>> sorted(name_mark_age,key=itemgetter(1,2))   #先按照等级排序，相同等级看年龄
```

```
10. [('zhangsan', 'A', 15), ('WANGWU', 'A', 16), ('LISI', 'B', 14),
    ('zhaoliu', 'B', 16)]
```

在官方文档上，有这样一个例子，和上面的操作是完全一样的。

```
1.  >>> class Student:
2.          def __init__(self, name, grade, age):
3.              self.name = name
4.              self.grade = grade
5.              self.age = age
6.
7.          def __repr__(self):
8.              return repr((self.name, self.grade, self.age))
9.
10. >>> student_objects = [
11.         Student('john', 'A', 15),       #注意这里，用class Student来生
    成列表内的值
12.         Student('jane', 'B', 12),       #因此，可以通过
    student_objects[i].age来访问某个名称的年龄,i=0,则是john的年龄
13.         Student('dave', 'B', 10),
14.         ]
15.
16. >>> sorted(student_objects, key=lambda student: student.age)
17. [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

也可以引用operator模块来实现上述排序

```
1. >>>from operator import attrgetter
2. >>> sorted(student_objects, key=attrgetter('age'))
3. [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
4. >>> sorted(student_objects, key=attrgetter('grade', 'age'))
5. [('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
```

总结：sorted的能力超强，不仅实现排序，还能按照指定关键词排序。

以上例子都是升序，如果，增加reverse=True。例如：

```
>>>from operator import itemgetter
>>> name_mark_age
[('zhangsan', 'A', 15), ('LISI', 'B', 14), ('WANGWU', 'A', 16),
('zhaoliu', 'B', 16)]
>>> sorted(name_mark_age, key=itemgetter(2),reverse=True)
[('WANGWU', 'A', 16), ('zhaoliu', 'B', 16), ('zhangsan', 'A', 15),
('LISI', 'B', 14)]
```

sorted的算法

python中的sorted算法，网上有人撰文，说比较低级。其实不然，通过阅读官方文档，发现python中的sorted排序，真的是高大上，用的Timsort算法。什么是Timsort，请看 wiki的解释：http://en.wikipedia.org/wiki/Timsort，另外，国内有一个文档，适当翻译：http://blog.csdn.net/yangzhongblog/article/details/8184707，这里截取一个不同排序算法比较的图示，就明白sorted的威力了。

从时间复杂度来看，Timsort是威武的。

从空间复杂度来讲，需要的开销在数量大的时候会增大。

综上，可以看出，就一般情况，使用sorted足以能够完成排序的要求，并且是稳定的。

当然，python中也有其它一些排序模块，都可以直接拿过来使用。

本文作者在博客和github上都有多种关于python排序方法和模块的文

章说明。

排序之快速排序算法

问题

快速排序，这是一个经典的算法，本文给出几种python的写法，供参考。

特别是python能用一句话实现快速排序。

思路说明

快速排序是C.R.A.Hoare于1962年提出的一种划分交换排序。它采用了一种分治的策略，通常称其为分治法(Divide-and-ConquerMethod)。

（1） 分治法的基本思想

分治法的基本思想是：将原问题分解为若干个规模更小但结构与原问题相似的子问题。递归地解这些子问题，然后将这些子问题的解组合为原问题的解。

（2）快速排序的基本思想

设当前待排序的无序区为R[low..high]，利用分治法可将快速排序的基本思想描述为:

分解:

在R[low..high]中任选一个记录作为基准(Pivot)，以此基准将当前无序区划分为左、右两个较小的子区间R[low..pivotpos-1)和R[pivotpos+1..high]，并使左边子区间中所有记录的关键字均小于等于基准记录(不妨记为pivot)的关键字pivot.key，右边的子区间中所有记录的关键字均大于等于pivot.key，而基准记录pivot则位于正确的位置(pivotpos)上，它无须参加后续的排序。

注意:

划分的关键是要求出基准记录所在的位置pivotpos。划分的结果可以简单地表示为(注意pivot=R[pivotpos]):

R[low..pivotpos-1].keys≤R[pivotpos].key≤R[pivotpos+1..high].keys

其中low≤pivotpos≤high。

求解:

通过递归调用快速排序对左、右子区间R[low..pivotpos-1]和R[pivotpos+1..high]快速排序。

组合:

因为当”求解”步骤中的两个递归调用结束时，其左、右两个子区间已有序。对快速排序而言，”组合”步骤无须做什么，可看作是空操作。

解决(Python)

```
1. #!/usr/bin/env python
2. #coding:utf-8
```

```
3.
4.  #方法1
5.
6.  def quickSort(arr):
7.      less = []
8.      pivotList = []
9.      more = []
10.     if len(arr) <= 1:
11.         return arr
12.     else:
13.         pivot = arr[0]      #将第一个值做为基准
14.         for i in arr:
15.             if i < pivot:
16.                 less.append(i)
17.             elif i > pivot:
18.                 more.append(i)
19.             else:
20.                 pivotList.append(i)
21.
22.         less = quickSort(less)      #得到第一轮分组之后，继续将分组进行下
    去。
23.         more = quickSort(more)
24.
25.         return less + pivotList + more
26.
27. #方法2
28. # 分为<, >, = 三种情况，如果分为两种情况的话函数调用次数会增加许多，以后几个好
    像都有相似的问题
29. # 如果测试1000个100以内的整数，如果分为<, >=两种情况共调用函数1801次，分为<,
    >, = 三种情况，共调用函数201次
30. def qsort(L):
31.     return (qsort([y for y in L[1:] if y <  L[0]]) + L[:1] + [y for
    y in L[1:] if y == L[0] + qsort([y for y in L[1:] if y > L[0]])) if
    len(L) > 1 else L
32.
33. #方法3
34. #基本思想同上，只是写法上又有所变化
35.
```

```
36. def qsort(list):
37.     if not list:
38.         return []
39.     else:
40.         pivot = list[0]
41.         less = [x for x in list     if x <  pivot]
42.         more = [x for x in list[1:] if x >= pivot]
43.         return qsort(less) + [pivot] + qsort(more)
44.
45. #方法4
46.
47. from random import *
48.
49. def qSort(a):
50.     if len(a) <= 1:
51.         return a
52.     else:
53.         q = choice(a)       #基准的选择不同于前，是从数组中任意选择一个值做为基准
54.         return qSort([elem for elem in a if elem < q]) + [q] * a.count(q) + qSort([elem for elem in a if elem > q])
55.
56.
57. #方法5
58. #这个最狠了，一句话搞定快速排序，瞠目结舌吧。
59.
60. qs = lambda xs : ( (len(xs) <= 1 and [xs]) or [ qs( [x for x in xs[1:] if x < xs[0]] ) + [xs[0]] + qs( [x for x in xs[1:] if x >= xs[0]] ) ] )[0]
61.
62.
63. if __name__=="__main__":
64.     a = [4, 65, 2, -31, 0, 99, 83, 782, 1]
65.     print quickSort(a)
66.     print qSort(a)
67.
68.     print qs(a)
```

排序算法的比较和选择

排序算法比较和选择

排序算法有不少，当然，一般的语言中都提供某个排序函数，比如Python中，对list进行排序，可以使用sorted（或者list.sort()），关于这方面的使用，在我的github代码库algorithm中有几个举例，有兴趣的看官可以去那里看看（顺便告知，我在Github中的账号是qiwsir，欢迎follow me）。但是，在某些情况下，语言中提供的排序方法或许不适合，必须选择某种排序算法。

这里整理一些排序算法的比较，供看官参考。

计算复杂度比较

名称	平均速度	最坏情况
冒泡排序法	O(n^2)	O(n^2)
快速排序法	O(nlogn)	O(n^2
选择排序	O(n^2)	O(n^2)
堆排序	O(nlogn)	O(nlogn)
插入排序	O(n^2)	O(n^2)
希尔排序	O(n^3/2)	O(n^2)
归并排序	O(nlogn)	O(nlogn)

系统资源占用比较

上述各个方法中，除了归并排序，只需要使用1个元素的临时存储单元，用于交换数据。归并排序则需要n个元素的存储单元，用来保存多遍合并操作。

稳定度比较

冒泡、插入、归并三种方法稳定性较好，其余稍逊。

如何选择排序法

没有哪一种排序方法是全面胜出的，只有根据具体具体情况选择了。一般下面的原则可以考虑：

- 当数据为正序时，尽可能用冒泡、插入、快速方法
- 如果n值较小（不超过50，但也要根据硬件状况确定），可以采用插入、选择排序。插入排序对小数量较为适用。
- 如果n值较大，采用时间O(nlogn)的排序方法较适宜。
- 如果强调稳定性，则使用归并排序。

按照指定字母顺序排序

问题

已知字母序列【d, g, e, c, f, b, o, a】，请实现一个函数针对输入的一组字符串 input[] = {“bed”, “dog”, “dear”, “eye”}，按照字母顺序排序并打印,结果应为: dear, dog, eye, bed。

说明

本问题在网上比较常见，但这里尝试用另外一个思路，并且用python来写，与众多用c++的有所不同，且似乎短小了不少。自己感觉比网上参考到的更容易理解。

欢迎指点。

解决(python)

```
#! /usr/bin/env python
#coding:utf-8

def char_to_number(by_list,char):    #根据排序依据字母顺序，给另外一个字母编号
    try:
        return by_list.index(char)
    except:
```

```
        return 1000

def sort_by_list(by_list,input_list):

    result={}
    for word in input_list:
        number_list = [char_to_number(by_list,word[i]) for i in range(len(word))]

        #得到形如：{"good":[2,3,3,1],"book":[1,3,3,0]}样式的结果
        result[word] = number_list

    #将得到的result生成[(key1,value1),(key2,value2),...]列表，按照value值排序，取出排序结果中的key即v[0]，生产列表。
    return [v[0] for v in sorted(result.items(),lambda x,y:cmp(x[1],y[1]))]

if __name__=="__main__":
    word = ["bed","dog","dear","eye"]
    by_string = ['d','g','e','c','f','b','o','a']
    print "the word list is:"
    print word
    print "\nwill sorted by:"
    print by_string
    print "\nthe result is:"
    print sort_by_list(by_string,word)
```

解法 （racket 5.2.1）

对语言的掌握还不熟练，所以暂未考虑算法效率

```
#lang racket

; 定义一个函数 filter-by-1st-char
; 输入一个字符 a-ch 和一个字符串 a-str
; 如果 a-ch 与 a-str 的第一个字符相同，
; 则以列表方式输出 a-str，否则输出空列表 '()
```

```
(define (filter-by-1st-char a-ch a-str)
  (if (char=? a-ch (string-ref a-str 0))
    (list a-word) '()))

; 定义一个函数 sort-words-by-char-list
; 它接受以字符串形式输入的一个字符序列 char-list,
; 和以字符串列表形式输入的一个字符串序列 word-list
; 输出一个列表，其所有列表项即 word-list 中的所有字符串
; 并且将根据每个字符串的首字母，依照 char-list 所提供的顺序排序
; 而并非按一般英文词典的 a-z 顺序排序
(define (sort-words-by-char-list char-list word-list)
  (let
    ([sorted-word-list '()]) ; 定义一个空列表用于存储排序后的 word
    (for ([ch char-list])
      (for ([wd word-list])
        (set! sorted-word-list
          (append sorted-word-list ; 把符合条件的单词放进结果列表
            (filter-by-1st-char ch wd)))))
    sorted-word-list))

; 函数调用，正常运行时，应该输出 (dear dog eye bed)
(displayln
  (sort-words-by-char-list "dgecfboa" '("dear" "dog" "eye" "bed")))
```

将一个整数分拆为若干整数和

问题

将一个整数，分拆为若干整数的和。例如实现：

4=3+1

4=2+2

4=2+1+1

4=1+1+1+1

解决(Python)

```
#! /usr/bin/env python
#encoding:utf-8

"""
"""

def int_divided(m,r,out_list):
    if(r==0):
        return True
    tm=r
    while tm>0:
        if(tm<=m):
            out.append(tm)
            if(divide(tm,r-tm,out_list)):
                print out
            out.pop()
        tm = tm-1
    return False

```

```
20.
21.  n=6
22.  output=[]
23.  int_divided(n-1,n,output)
```

判断一个数是否为素数的多种方法

问题

判断一个数是否为素数

思路说明

这个问题有多种解法，以下的解法来自网络整理。供参考使用。

解决（Python）

```
#! /usr/bin/env python
#coding:utf-8

"""
"""

#方法一

import math

def isPrime1(n):
    if n <= 1:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False

    return True
```

```
19.
20. #方法二
21.
22. def isPrime2(n):
23.     if n <= 1:
24.         return False
25.
26.     i = 2
27.     while i*i <= n:
28.         if n % i == 0:
29.             return False
30.         i += 1
31.     return True
32.
33. #方法三
34.
35. from itertools import count
36.
37. def isPrime3(n):
38.     if n <= 1:
39.         return False
40.     for i in count(2):
41.         if i * i > n:
42.             return True
43.         if n % i == 0:
44.             return Fals
45.
46. #方法四
47.
48. def isPrime4(n):
49.     if n <= 1:
50.         return False
51.     if n == 2:
52.         return True
53.     if n % 2 == 0:
54.         return False
55.     i = 3
56.     while i * i <= n:
```

```
57.         if n % i == 0:
58.             return False
59.         i += 2
60.     return True
61.
62.
63. if __name__=="__main__":
64.     a=isPrime4(5)
65.     print a
```

将list中的数字组合成最小的整数

问题

把一个int型数组中的数字拼成一个串，这个串代表的数字最小。

＃思路说明

对这个问题的理解：

1. 有一个元素是int类型的list；
2. 将上述list中的每个元素的数字分别取出来，然后将这些数字的顺序进行从新排列，并将其中的最小整数输入，就是题目中要求的最小数字。

如果按照上述理解，在解题中，最应当小心的是数字如果很大，比如list中的某个int元素是：22222222222222777777777777666666666669999999999988888888888…很大的整数，就不得不转化为字符串操作了。

所以，在本问题中，基本思路是：

1. 将list中的int元素转换为str；
2. 将所有数字(str类型）装入到一个list2
3. 对list2进行排序
4. 将list2中的数字（str类型）组装成一个数值(str类型)

解决(Python)

```python
#! /user/bin/env python
#coding:utf-8


def joint_int(lst):
    str_list = [str(i) for i in lst]
    str_lonely = [str_list[i][j] for i in range(len(str_list)) for j in range(len(str_list[i]))]

    sorted_str = sorted(str_lonely)
    return "".join(sorted_str)

print joint_int([1230975,4087644567856])
```

qiwsir#gmail.com

无向图最小生成树Kruskal算法

问题

最小生成树的Kruskal算法

描述：有A、B、C、D四个点，每两个点之间的距离（无方向）是(第一个数字是两点之间距离，后面两个字母代表两个点）：(1,'A','B'),(5,'A','C'),(3,'A','D'),(4,'B','C'),(2,'B','D'),(1,'C','D')
生成边长和最小的树，也就是找出一种连接方法，将各点连接起来，并且各点之间的距离和最小。

思路说明:

Kruskal算法是经典的无向图最小生成树解决方法。此处列举两种python的实现方法。这两种方法均参考网上，并根据所学感受进行了适当改动。

解决1（Python）

```
1. #! /usr/bin/env python
2. #coding:utf-8
3.
4. #以全局变量X定义节点集合，即类似{'A':'A','B':'B','C':'C','D':'D'},如果A、B两点联通，则会更改为{'A':'B','B':'B",...},即任何两点联通之后，两点的值
```

```
    value将相同。
5.
6.  X = dict()
7.
8.  #各点的初始等级均为0,如果被做为连接的的末端，则增加1
9.
10. R = dict()
11.
12. #设置X R的初始值
13.
14. def make_set(point):
15.     X[point] = point
16.     R[point] = 0
17.
18. #节点的联通分量
19.
20. def find(point):
21.     if X[point] != point:
22.         X[point] = find(X[point])
23.     return X[point]
24.
25. #连接两个分量（节点）
26.
27. def merge(point1,point2):
28.     r1 = find(point1)
29.     r2 = find(point2)
30.     if r1 != r2:
31.         if R[r1] > R[r2]:
32.             X[r2] = r1
33.         else:
34.             X[r1] = r2
35.             if R[r1] == R[r2]: R[r2] += 1
36.
37. #KRUSKAL算法实现
38.
39. def kruskal(graph):
40.     for vertice in graph['vertices']:
41.         make_set(vertice)
```

```
42.
43.     minu_tree = set()
44.
45.     edges = list(graph['edges'])
46.     edges.sort()                    #按照边长从小到达排序
47.     for edge in edges:
48.         weight, vertice1, vertice2 = edge
49.         if find(vertice1) != find(vertice2):
50.             merge(vertice1, vertice2)
51.             minu_tree.add(edge)
52.     return minu_tree
53.
54.
55. if __name__=="__main__":
56.
57.     graph = {
58.         'vertices': ['A', 'B', 'C', 'D', 'E', 'F'],
59.         'edges': set([
60.             (1, 'A', 'B'),
61.             (5, 'A', 'C'),
62.             (3, 'A', 'D'),
63.             (4, 'B', 'C'),
64.             (2, 'B', 'D'),
65.             (1, 'C', 'D'),
66.             ])
67.         }
68.
69.     result = kruskal(graph)
70.     print result
71.
72. """
73. 参考:
74. 1.https://github.com/qiwsir/Algorithms-Book--Python/blob/master/5-
    Greedy-algorithms/kruskal.py
75. 2.《算法基础》(GILLES Brassard,Paul Bratley)
76. """
```

解决2（Python）

以下代码参考http://www.ics.uci.edu/~eppstein/PADS/的源码

```
#! /usr/bin/env python
#coding:utf-8

import unittest

class UnionFind:
    """
    UnionFind的实例：
    Each unionFind instance X maintains a family of disjoint sets of
    hashable objects, supporting the following two methods:

    - X[item] returns a name for the set containing the given item.
      Each set is named by an arbitrarily-chosen one of its members; as
      long as the set remains unchanged it will keep the same name. If
      the item is not yet part of a set in X, a new singleton set is
      created for it.

    - X.union(item1, item2, ...) merges the sets containing each item
      into a single larger set.  If any item is not yet part of a set
      in X, it is added to X as one of the members of the merged set.
    """

    def __init__(self):
        """Create a new empty union-find structure."""
        self.weights = {}
```

```
        self.parents = {}

    def __getitem__(self, object):
        """Find and return the name of the set containing the
object."""

        # check for previously unknown object
        if object not in self.parents:
            self.parents[object] = object
            self.weights[object] = 1
            return object

        # find path of objects leading to the root
        path = [object]
        root = self.parents[object]
        while root != path[-1]:
            path.append(root)
            root = self.parents[root]

        # compress the path and return
        for ancestor in path:
            self.parents[ancestor] = root
        return root

    def __iter__(self):
        """Iterate through all items ever found or unioned by this
structure."""
        return iter(self.parents)

    def union(self, *objects):
        """Find the sets containing the objects and merge them
all."""
        roots = [self[x] for x in objects]
        heaviest = max([(self.weights[r],r) for r in roots])[1]
        for r in roots:
            if r != heaviest:
                self.weights[heaviest] += self.weights[r]
                self.parents[r] = heaviest
```

```
61.
62.
63. """
64. Various simple functions for graph input.
65.
66. Each function's input graph G should be represented in such a way
    that "for v in G" loops through the vertices, and "G[v]" produces a
    list of the neighbors of v; for instance, G may be a dictionary
    mapping each vertex to its neighbor set.
67.
68. D. Eppstein, April 2004.
69. """
70.
71. def isUndirected(G):
72.     """Check that G represents a simple undirected graph."""
73.     for v in G:
74.         if v in G[v]:
75.             return False
76.         for w in G[v]:
77.             if v not in G[w]:
78.                 return False
79.     return True
80.
81.
82. def union(*graphs):
83.     """Return a graph having all edges from the argument graphs."""
84.     out = {}
85.     for G in graphs:
86.         for v in G:
87.             out.setdefault(v,set()).update(list(G[v]))
88.     return out
89.
90.
91. """
92. Kruskal's algorithm for minimum spanning trees. D. Eppstein, April
    2006.
93. """
94.
```

```
95.  def MinimumSpanningTree(G):
96.      """
97.      Return the minimum spanning tree of an undirected graph G.
98.      G should be represented in such a way that iter(G) lists its
99.      vertices, iter(G[u]) lists the neighbors of u, G[u][v] gives
     the
100.     length of edge u,v, and G[u][v] should always equal G[v][u].
101.     The tree is returned as a list of edges.
102.     """
103.     if not isUndirected(G):
104.         raise ValueError("MinimumSpanningTree: input is not
     undirected")
105.     for u in G:
106.         for v in G[u]:
107.             if G[u][v] != G[v][u]:
108.                 raise ValueError("MinimumSpanningTree: asymmetric
     weights")
109.
110.     # Kruskal's algorithm: sort edges by weight, and add them one
     at a time.
111.     # We use Kruskal's algorithm, first because it is very simple
     to
112.     # implement once UnionFind exists, and second, because the only
     slow
113.     # part (the sort) is sped up by being built in to Python.
114.     subtrees = UnionFind()
115.     tree = []
116.     for W,u,v in sorted((G[u][v],u,v) for u in G for v in G[u]):
117.         if subtrees[u] != subtrees[v]:
118.             tree.append((u,v))
119.             subtrees.union(u,v)
120.     return tree
121.
122.
123. # If run standalone, perform unit tests
124.
125. class MSTTest(unittest.TestCase):
126.     def testMST(self):
```

```
127.         """Check that MinimumSpanningTree returns the correct
     answer."""
128.         G = {0:{1:11,2:13,3:12},1:{0:11,3:14},2:{0:13,3:10},3:
     {0:12,1:14,2:10}}
129.         T = [(2,3),(0,1),(0,3)]
130.         for e,f in zip(MinimumSpanningTree(G),T):
131.             self.assertEqual(min(e),min(f))
132.             self.assertEqual(max(e),max(f))
133.
134. if __name__ == "__main__":
135.     unittest.main()
```

无向图最小生成树的Prim算法

问题

无向图最小生成树的Prim算法

思路说明

假设点A,B,C,D,E,F，两点之间有连线的，以及它们的距离分别是：(A-B:7);(A-D:5);(B-C:8);(B-D:9);(B-E:7);(C-E:5);(D-E:15);(D-F:6);(E-F:8);(E-G:9);(F-G:11)

关于Prim算法的计算过程，参与维基百科的词条：普里姆算法

将上述点与点关系以及两点之间距离（边长，有的文献中称之为权重）写成矩阵形式（在list中，每两个点及其之间的距离组成一个tuple）

```
edges = [ (“A”, “B”, 7),
(“A”, “D”, 5),
(“B”, “C”, 8),
(“B”, “D”, 9),
(“B”, “E”, 7),
(“C”, “E”, 5),
(“D”, “E”, 15),
(“D”, “F”, 6),
```

```
(“E”, “F”, 8),
(“E”, “G”, 9),
(“F”, “G”, 11)
]
```

在下面的解决方法中，要计算出与已经选出的若干个点有相邻关系的点中，相应边长最短的点。这本质上是排序之后取出最小的，因为这种排序是动态的，如果用sorted或者list.sort()之类的方法对list排序，一则速度慢（python中的sort方法对大数据时不是很快),二则代码也长了。幸好python提供了一个非常好用的模块：heapq。这个模块是堆排序方法实现排序，并能够随时取出最小值。简化代码，更重要是提升了速度。

就用这个来解决Prim算法问题了。

解决（Python）

```
1.  #! /usr/bin/env python
2.  #coding:utf-8
3.
4.  from collections import defaultdict
5.  from heapq import *
6.
7.  def prim( vertexs, edges ):
8.      adjacent_vertex = defaultdict(list)
9.
10.     """
11.     注意：defaultdict(list)必须以list做为变量，可以详细阅读：
    [collections.defaultdict]
    (https://docs.python.org/2/library/collections.html#collections.defau
12.     """
13.     for v1,v2,length in edges:
14.         adjacent_vertex[v1].append((length, v1, v2))
15.         adjacent_vertex[v2].append((length, v2, v1))
```

```
16.
17.     """
18.     经过上述操作，将edges列表中各项归类成以某点为dictionary的key，其value则是其相邻的点以及边长。如下：
19.
20.     defaultdict(<type 'list'>, {'A': [(7, 'A', 'B'), (5, 'A', 'D')],
21.                                 'C': [(8, 'C', 'B'), (5, 'C', 'E')],
22.                                 'B': [(7, 'B', 'A'), (8, 'B', 'C'), (9, 'B', 'D'), (7, 'B', 'E')],
23.                                 'E': [(7, 'E', 'B'), (5, 'E', 'C'), (15, 'E', 'D'), (8, 'E', 'F'), (9, 'E', 'G')],
24.                                 'D': [(5, 'D', 'A'), (9, 'D', 'B'), (15, 'D', 'E'), (6, 'D', 'F')],
25.                                 'G': [(9, 'G', 'E'), (11, 'G', 'F')],
26.                                 'F': [(6, 'F', 'D'), (8, 'F', 'E'), (11, 'F', 'G')]})
27.
28.     """
29.
30.     mst = []        #存储最小生成树结果
31.
32.     chosed = set(vertexs[0])
33.
34.     """
35.     vertexs是顶点列表，vertexs = list("ABCDEFG")===>vertexs=['A', 'B', 'C', 'D', 'E', 'F', 'G']
36.     >> chosed=set(vertexs[0])
37.     >> chosed
38.     set(['A'])
39.     也就是，首先选一个点（这个点是可以任意选的），以这个点为起点，找其相邻点，以及最短边长。
40.
41.     """
42.
43.     #得到adjacent_vertexs_edges中顶点是'A'（nodes[0]='A')的相邻点list,
```

```
    即adjacent_vertexs['A']=[(7,'A','B'),(5,'A','D')]
44.
45.     adjacent_vertexs_edges = adjacent_vertex[vertexs[0]]
46.
47.     #将usable_edges加入到堆中，并能够实现用heappop从其中动态取出最小值。关于
    heapq模块功能，参考python官方文档
48.
49.     heapify(adjacent_vertexs_edges)
50.
51.     while adjacent_vertexs_edges:
52.         #得到某个定点（做为adjacent_vertexs_edges的键）与相邻点距离（相邻
    点和边长/距离做为该键的值）最小值，并同时从堆中清除。
53.         w, v1, v2 = heappop(adjacent_vertexs_edges)
54.         if v2 not in chosed:
55.
56.             #在used中有第一选定的点'A'，上面得到了距离A点最近的点'D',举例是
    5。将'd'追加到used中
57.             chosed.add(v2)
58.
59.             mst.append((v1,v2,w))           #将v1,v2,w，第一次循环就是
    ('A','D',5) append into mst
60.
61.             #再找与d相邻的点，如果没有在heap中，则应用heappush压入堆内，以
    加入排序行列
62.
63.             for next_vertex in adjacent_vertex[v2]:
64.                 if next_vertex[2] not in chosed:
65.                     heappush( adjacent_vertexs_edges,next_vertex)
66.     return mst
67.
68.
69. #test
70. vertexs = list("ABCDEFG")
71. edges = [ ("A", "B", 7), ("A", "D", 5),
72.           ("B", "C", 8), ("B", "D", 9),
73.           ("B", "E", 7), ("C", "E", 5),
74.           ("D", "E", 15), ("D", "F", 6),
75.           ("E", "F", 8), ("E", "G", 9),
```

```
76.             ("F", "G", 11)]
77.
78.  print "edges:",edges
79.  print "prim:", prim( vertexs, edges )
```

运行结果

edges: [(‘A’, ‘B’, 7), (‘A’, ‘D’, 5), (‘B’, ‘C’, 8), (‘B’, ‘D’, 9), (‘B’, ‘E’, 7), (‘C’, ‘E’, 5), (‘D’, ‘E’, 15), (‘D’, ‘F’, 6), (‘E’, ‘F’, 8), (‘E’, ‘G’, 9), (‘F’, ‘G’, 11)]

prim: [(‘A’, ‘D’, 5), (‘D’, ‘F’, 6), (‘A’, ‘B’, 7), (‘B’, ‘E’, 7), (‘E’, ‘C’, 5), (‘E’, ‘G’, 9)]

查找字符串中出现最多的字符和个数

问题:

查找字符串中出现最多的字符和个数？

如 sdsdsddsssssssdd -> 字符最多的是s，出现9次

思路说明

利用python中的collections模块的Counter，查此函数详细内容.对字符串进行统计。

然后将结果转化为字典类型。

特别注意，在字符串中可能会出现数量并列第一的字符，因此要通过循环找出最大数之后，再通过循环找出最大数对应的字母（键）。

解答1(python)

```
1.  import collections
2.
3.  def most_character_number(one_string):
4.      static_result = collections.Counter(one_string)
```

```
    result = dict(static_result)
    most_number = max([value for value in result.values()])
    most_character = [key for key,value in result.items() if value==most_number]
    return (most_number,most_character)

if __name__ == "__main__":
    (most_num,most_char) = most_character_number("yyyyyyddddddkuuuiii")
    print ("The most character is:%s"%most_num)
    print ("The number is:")
    for char in most_char:
        print char
```

解答2(python)

```
str1 = "sdsdsddsssssssssdd"

def most_character_number(one_string):
    """利用字典key来统计次数"""
    str_dict = {}
    for item in one_string:
        if item in str_dict:
            str_dict[item] +=1
        else:
            str_dict[item] =1


    str_dict = {str_dict[key]:key for key in str_dict}
    return (max(str_dict),str_dict[max(str_dict)])

print (most_character_number(str1))
```

解答2由黄老师提供，他的微博。

欢迎接龙，可以按照上面格式继续。

解答3 (racket 5.2.1)

```
#lang racket

; 需要引用 scheme 标准库中的 list 库 (SRFI-1)
(require srfi/1)

; 定义函数 get-key-by-value
; 输入一个 Hash 表和一个 a-value (用于反向查找 key 的 value)
; 输出 value 等于 a-value 的那些 key,
; 换言之, 此 key 满足条件 (hash-ref key) == a-value.
(define (get-key-by-value a-hash a-value)
  (hash-for-each a-hash
    (lambda (k v)
      (if (= a-value (hash-ref a-hash k))
        (printf "~a: ~a~n" k v) '()))))

; 定义函数 char-cmp
; 输入一个 preset-char (预设字符)
; 返回一个的匿名函数 (其内部包含 preset-char)
; > 其输入为 char-to-be-compared (待比较的字符)
; > 当 char-to-be-compared 与 preset-char
; > 相等时输出 true, 否则输出 false
(define (char-cmp preset-char)
  (lambda (char-to-be-compared)
    (if (char=? preset-char char-to-be-compared) true false)))

; 定义函数 most-character-number
; 输入一个 a-string 任意字符串
; 输出 a-string 中出现次数最多的字符, 及其出现次数
(define (most-character-number a-string)
  (let*
    ([all-chars-list (string->list a-string)]
     [key-chars-list (delete-duplicates all-chars-list)]
     [result-hash (make-hash)])
    (for ([key-char key-chars-list])
```

```
35.       (hash-set! result-hash
36.         key-char (count (char-cmp key-char) all-chars-list)))
37.       (get-key-by-value
38.         result-hash
39.         (apply max (hash-values result-hash)))))
40.
41. ; 以下函数调用在正常运行之后, 应该显示
42. ; f: 3
43. ; d: 3
44. ; a: 3
45. (most-character-number "ssfdaa dffda ")
```

联系老齐: qiwsir#gmail.com

list中数字的和、最值、均值

问题

定义一个int型的一维数组，包含10个元素，赋一些随机整数
然后求出所有元素的最大值，最小值，平均值，和值，并输出出来。

思路说明

本问题是一个普通的对整数数组的操作，在下面的Python解决方法中，主要是尝试了python的一个内置函数reduce。

解决（Python）

```
#! /usr/bin/env python
#coding:utf-8

from __future__ import division     #实现精确的除法，例如4/3=1.333333
import random

def add(x,y):
    return x+y

def operate_int_list():
    int_list = [random.randint(1,100) for i in range(10)]     #在1,100范围内，随机选择10个整数组成一个list
    max_num = max(int_list)
    min_num = min(int_list)
```

```
    sum_num = reduce(add,int_list)      #这里使用了reduce函数，也可以使用for循环，如下所示
    """
    sum_num = 0
    for i in int_list:
        sum_num = sum_num+i
    """
    ave_num = sum_num/len(int_list)

    return (int_list,max_num,min_num,sum_num,ave_num)

if __name__=="__main__":
    int_list,max_num,min_num,sum_num,ave_num = operate_int_list()
    print "the int list is:",int_list
    print "max number is:",max_num
    print "min number is:",min_num
    print "the sum of all int in the list:",sum_num
    print "the average of the sum:",ave_num
```

解法 (racket 5.2.1)

```
#lang racket

; 定义一个函数 operate-int-list
; 它接受一个正整数输入 n
; 它的输出是一个列表
; 列表的第一项是一个长度为 n 的列表，此列表的每个元素都是一个 1~100 之间的伪随机数
; 列表的第二、三、四、五项分别为
; 所有伪随机数中的最大者
; 所有伪随机数中的最小者
; 所有伪随机数的总和
; 所有伪随机数中平均值，以有理数形式表示
(define (operate-int-list n)
  (if (<= n 0)
      false
      (let*
```

```
          ([rand1to100 (lambda () (+ 1 (random 100)))]
           [random-list (for/list ([i n]) (rand1to100))]
           [max-int (apply max random-list)]
           [min-int (apply min random-list)]
           [sum (apply + random-list)]
           [average (/ (apply + random-list) (length random-
list))])
        (list random-list
              max-int min-int sum average))))

; 函数调用，当程序运行正常时，运算结果在形式上类似于以下结果，
; 但每次的结果列表都有一定的随机性
; '((19 77 15 49 39 84 45 72 75 100) 100 15 575 57½)
(operate-int-list 10)
```

寻找完全数

问题

寻找完全数。

思路说明

所谓完全数，从维基百科的完全数词条中得到：

> 完全数，又称完美数或完备数，是一些特殊的自然数：它所有的真因子（即除了自身以外的约数）的和，恰好等于它本身，完全数不可能是楔形数。
> 例如：第一个完全数是*6*，它有约数*1*、*2*、*3*、*6*，除去它本身*6*外，其余*3*个数相加，*1+2+3 = 6*，恰好等于本身。第二个完全数是*28*，它有约数*1*、*2*、*4*、*7*、*14*、*28*，除去它本身*28*外，其余*5*个数相加，*1+2+4+7+14 = 28*，也恰好等于本身。后面的数是*496*、*8128*。

更多完全数内容请看该词条。

所以，要找一个完全数的解决思路是：

1. 找出一个数的所有因数
2. 验证这些因数的和是否等于这个数，如果等于，就是完全数。

解决(python)

```
1. #!/usr/bin/env python
2. #coding:utf-8
3. 
4. #找出一个数的因数
5. def factors(n):
```

```
6.      #return [i for i in range(1,n/2+1) if n%i == 0]
7.      #如果仅仅是为了得到因数，可以用上面的
8.      #如果是配合下面完全数，最好使用下面的。因为在下面少循环一次，1肯定是任何整数的因数
9.      return [i for i in range(2,n/2+1) if n%i == 0]
10.
11. #找出某个数n以内的所有完全数，即在[1,n]内(含n)
12. def perfect(n):
13.     #从上面的factors中得到的因数列表中，少1,因此在求因数和的时候，要把1加上。
14.     return [i for i in range(2,n+1) if (sum(factors(i))+1)==i]
15.
16. if __name__=="__main__":
17.     print perfect(30)
```

计算余数

问题

要求定义一个int型数组a,包含100个元素,保存100个随机的4位数。再定义一个int型数组b，包含10个元素。统计a数组中的元素对10求余等于0的个数，保存到b[0]中；对10求余等于1的个数，保存到b[1]中，……依此类推。

解决（python）

```
#!/usr/bin/env python
#coding:utf-8

import random
if __name__=="__main__":
    a = [random.randint(1000,9999) for i in range(101)]
    a_remainder = [i%10 for i in a]
    b = [a_remainder.count(i) for i in range(10)]
    print a
    print a_remainder
    print b
```

解决 (racket 5.2.1)

```
#lang racket

; 定义函数 equal-to-k
```

```
4.  ; 它接受一个整数输入 k
5.  ; 它输出一个只返回 true/false 的匿名函数
6.  ; 当 k = 1 时, equal-to-k 的功能可以这样理解
7.  ; (equal-to-k 1) => true
8.  ; (equal-to-k 2) => false
9.  (define (filter-by-k k)
10.   (lambda (n) (if (= k n) true false)))
11.
12. ; 定义函数 random-list-100-remainder-stats
13. ; 它不接受任何输入
14. ; 它输出一个整数型列表 b, 其每个元素的定义如下
15. ; > 给定一个整数型列表 a,包含 100 个元素,保存 100 个随机的 4 位数。
16. ; > 整数型列表 b, 包含 10 个元素。
17. ; > 统计a列表中的元素对10求余等于0的个数, 保存到b[0]中；
18. ; > 对10求余等于1的个数, 保存到b[1]中, ……依此类推。
19. (define (random-list-100-remainder-stats)
20.   (let*
21.       ([rand-e4 (lambda () (+ 1000 (random 9000)))]     ; 定义
    1000~9999 之间的随机数"生成器"
22.        [rand-list-100 (for/list ([i 100]) (rand-e4))]   ; 生成长度为
    100 的列表, 其中每个元素都是 1000~9999 之间的随机数
23.        [get-remainder-by-10 (lambda (n) (modulo n 10))] ; 定义对某整
    数除以 10 取余数的"生成器"
24.        [remainder-list-100 (map get-remainder-by-10 rand-list-100)]
    ; 对前面长为 100 的列表批量除以 10 取余数
25.        ; 最后, 对余数做统计, 生成 b 列表
26.        [remainder-list-stats (for/list ([k (in-range 0 10)]) (count
    (filter-by-k k) remainder-list-100))])
27.     ; 把生成的 b 列表输出出来
28.     remainder-list-stats))
29.
30. ; 函数调用, 正常执行时, 应输出一个整数的列表
31. ; 此列表的展现形式类似于 '(7 11 13 10 5 11 6 11 12 14)
32. (random-list-100-remainder-stats)
```

删除list中的重复元素

问题

统计一个一维数组中的各个元素的个数，然后删除多出来的重复元素，并输出结果。

例如：[1,2,2,2,3,3,3,3,3]—>[1,2,3]

解决思路

将重复元素的列表中的重复元素进行统计，并将统计结果放在dictionary中，key为元素，value为该元素的个数

更新此步方法：上述步骤的功能，能够通过另外一个方法实现，即collections.Counter()

然后通过for获取key，得到一个新的列表，就是没有重复元素的列表

解决（Python）

```
1. #!/usr/bin/env python
2. #coding:utf-8
3.
4.
5. def count_element(one_list):
```

```
    element_number = {}
    for e in one_list:
        number = one_list.count(e)      #数出某个元素的个数
        element_number[e] = number     #生成类似：{1:1,2:3,3:5}的结果，key-element,value-元素的个数
    return element_number

#应用collections.Counter()实现count_element(one_list)函数功能，为了便于调试和说明，在另外一个函数里面使用

from collections import Counter

def count_element2(one_list):
    return Counter(one_list)


def no_repeat_element(element_number):      #element_number是count_element(one_list)的返回值
    no_repeat_list = [key for key in element_number]
    return no_repeat_list

if __name__=="__main__":

    ls = ["a","a","b","b",'b','c','c']
    el_num=count_element(ls)
    print el_num
    no_repeat = no_repeat_element(el_num)
    print no_repeat
```

解决（python）

无重复元素个数统计，只有新数组输出

```
list_a = [1,1,2,2,2,3,3,3,3,3,]
list_b = list(set(list_a))
```

qiwsir#gmail.com (# to @)

将字符串写成驼峰样式

问题

请写一个字符串转成驼峰的方法？

例如: border-bottom-color -> borderBottomColor

解决

python解决方法：

```
1. def convert(one_string,space_character):    #one_string:输入的字符串；space_character:字符串的间隔符，以其做为分隔标志
2. 
3.     string_list = str(one_string).split(space_character)    #将字符串转化为list
4.     first = string_list[0].lower()
5.     others = string_list[1:]
6. 
7.     others_capital = [word.capitalize() for word in others]    #str.capitalize():将字符串的首字母转化为大写
8. 
9.     others_capital[0:0] = [first]
```

```
10.
11.     hump_string = ''.join(others_capital)      #将list组合成为字符串,中间无连接符。
12.
13.     return hump_string
14.
15. if __name__=='__main__':
16.     print "the string is:ab-cd-ef"
17.     print "convert to hump:"
18.     print convert("ab-cd-ef","-")
```

欢迎补充其它语言的解决方法

racket解决方法 (racket 5.2.1)

```
1.  #lang racket
2.
3.  ; 定义字符串转换函数 train-to-camel
4.  (define (train-to-camel train-str separator-char)
5.    (let
6.      [(splited-str (regexp-split separator-char train-str))]      ; 把原始字符串用 '-' 分成多个单独的单词
7.      (string-append
8.        (first splited-str)
9.        (apply
10.         string-append
11.         (map
12.           string-titlecase
13.           (rest splited-str))))))
14.
15. ; 调用字符串转换函数 train-to-camel
16. (train-to-camel "this-is-a-var" "-")  ; 正常运行的情况下, 应输出 "thisIsAVar"
```

Ruby 解决方法

```
str.split('-').map{|x| x.capitalize}.join
```

```
str = 'border-bottom-color'
str.split('-').map{|x| x.capitalize}.join
# => "BorderBottomColor"
```

联系我: 老齐 qiwsir#gmail.com (# to @)

九宫格问题

问题

九宫格问题，要求：

- 行列必须为相等的奇数
- 每行数字之和、每列数字之和、两个对角线数字之和，都相等

思路说明

按照下面的方式排列

横向为x(从0到n-1)，纵向为y方向（从0到n-1)

1、第一个数放在X方向的中间位置

2、其它数顺次放置各个位置，并依据如下原则：（假设第一个数是a，第二个数是b）

以a为中心的位置关系分别为：

左上|上|右上

左 |a |右

左下|下|右下

（1）b放在a的右上位置。a(x,y)—>b(x+1,y-1)

（2）如果仅有“右”位置超过边界，即x+1>n，则b(1,y-1)

（3）如果仅有“上”位置超过边界，即y-1<0，则b(x+1,n)

（4）如果“右”“上”位置都超过边界，即x+1>n,y-1<o,则b(x,y+1)

（5）如果“右上”已经有值，则b(x,y+1)

解决(Python)

```python
#! /usr/bin/env python
#coding:utf-8

#判断输入的九宫格的格数是否为奇数(此处未使用，目的是对所输入的数进行判断)
def if_odd(n):
    if n%2==1:
        return True
    else:
        return False

#九宫格填写数的法则
def sudoku_rule(n,sudoku):

    tx = n/2
    ty = 0
    for i in range(n*n):
        sudoku[ty][tx] = i+1
        tx = tx+1
        ty = ty-1
        if ty<0 and tx>=n:      #条件（4）
            tx = tx-1
            ty = ty+2
        elif ty<0:              #条件(3)
            ty = n-1
        elif tx>=n:             #条件(2)
            tx = 0
        elif sudoku[ty][tx]!=0: #条件(5)
```

```
            tx = tx-1
            ty = ty+2
    return sudoku

if __name__=="__main__":
    n = 5
    sudoku = [[0 for i in range(n)] for i in range(n)]
    s = sudoku_rule(n,sudoku)
    for line in s:
        print line
```

说明：最后打印的结果和输入的九宫格格数，都可以进一步修改。

qiwsir#gmail.com (# to @)