

python模块学习：anydbm, shelve

原文地址：[DarkBull](#)

好久没写这系列的文章了，我越来越喜欢用python了，它在我的工作中占据的比例越来越大。废话少说，直接进入主题。

anydbm允许我们将一个磁盘上的文件与一个“dict-like”对象关联起来，操作这个“dict-like”对象，就像操作dict对象一样，最后可以将“dict-like”的数据持久化到文件。对这个“dict-like”对象进行操作的时候，key和value的类型必须是字符串。下面是使用anydbm的例子：

Python

```
#coding=utf-8
1 #coding=utf-8
2
3 import anydbm
4
5 def CreateData():
6     try:
7         db = anydbm.open('db.dat', 'c')
8         # key与value必须是字符串
9         # db['int'] = 1
10        # db['float'] = 2.3
11        db['string'] = "I like python."
12        db['key'] = 'value'
13    finally:
14        db.close()
15
16 def LoadData():
17     db = anydbm.open('db.dat', 'r')
18     for item in db.items():
19         print item
20     db.close()
21
22 if __name__ == '__main__':
23     CreateData()
24     LoadData()
```

anydbm.open(filename[, flag[, mode]])，filename是关联的文件路径，可选参数flag可以是：'r': 只读，'w': 可读写，'c': 如果数据文件不存在，就创建，允许读写；'n': 每次调用open()都重新创建一个空的文件。mode是unix下文件模式，如0666表示允许所有用户读写。

shelve模块是anydbm的增强版，它支持在“dict-like”对象中存储任何可以被pickle序列化的对象，但key也必须是字符串。同样的例子，与shelve来实现：

Python

```
import shelve
1 import shelve
2
3 def CreateData():
4     try:
```

```
5     db = shelve.open('db.dat', 'c')
6     # key与value必须是字符串
7     db['int'] = 1
8     db['float'] = 2.3
9     db['string'] = "I like python."
10    db['key'] = 'value'
11    finally:
12        db.close()
13
14 def LoadData():
15     db = shelve.open('db.dat', 'r')
16     for item in db.items():
17         print item
18     db.close()
19
20 if __name__ == '__main__':
21     CreateData()
22     LoadData()
```

1 赞 收藏 [评论](#)

Python模块学习：fileinput

原文地址：[DarkBull](#)

这几天有这样一个需求，要将用户登陆系统的信息统计出来，做成一个报表。当用户登陆成功的时候，服务器会往日志文件里写一条像下面这种格式的记录：“日期时间@用户名@IP”，这样的日志文件每天生成一个。所以，我们只要遍历这些日志文件，将所有的登陆信息提取出来，并重新组织数据格式就可以了。用python写一个分析工具非常简单，你会说，用glob获取所有的日志文件，然后对每个日志文件都open(logfile)，再一行一行的读取；或者用os.walk，也很简单。其实，标准库提供了另一个辅助模块，我们可以非常方便的完成这个工作，那就是fileinput。下面我们就通过fileinput来遍历所有的D盘下的文本文件，将每一行的长度打印出来：

Python

```
import fileinput  
from glob import glob  
  
1 import fileinput  
2 from glob import glob  
3  
4 for line in fileinput.input(glob(r'd:/*.txt')):  
5     print fileinput.lineno(), u'文件:', fileinput.filename(), /  
6         u'行号:', fileinput.filelineno(), u'长度:', len(line.strip('/n'))  
7 fileinput.close()
```

代码非常简单明了。input()接受要遍历的所有文件路径的列表，通过filename()返回当前正在读取的文件的文件名，filelineno()返回当前读取的行的行号，而lineno()返回当前已经读取的行的数量（或者序号）。其实，模块内部通过FileInput类来实现文件的遍历读取，input()在内部创建了该类的一个对象，当处理完数据行之后，通过fileinput.close()来关闭这个内部对象。

模块非常简单，详细的内容可以参考标准库手册。

1 赞 收藏 [评论](#)

Python算法：图

原文出处：[hujiawei \(@五道口宅男\)](#)

本节主要介绍图算法中的各种最短路径算法，从不同的角度揭示它们的内核以及它们的异同

在前面的内容里我们已经介绍了图的表示方法(邻接矩阵和“各种”邻接表)、图的遍历(DFS和BFS)、图中的一些基本算法(基于DFS的拓扑排序和有向无环图的强连通分量、最小生成树的Prim和Kruskal算法等)，剩下的就是图算法中的各种最短路径算法，也就是本节的主要内容。

[The shortest path problem comes in several varieties. For example, you can find shortest paths (just like any other kinds of paths) in both directed and undirected graphs. The most important distinctions, though, stem from your starting points and destinations. Do you want to find the shortest from one node to all others (single source)? From one node to another (single pair, one to one, point to point)? From all nodes to one (single destination)? From all nodes to all others (all pairs)? Two of these—single source and all pairs—are perhaps the most important. Although we have some tricks for the single pair problem (see “Meeting in the middle” and “Knowing where you’re going,” later), there are no guarantees that will let us solve that problem any faster than the general single source problem. The single destination problem is, of course, equivalent (just flip the edges for the directed case). The all pairs problem can be tackled by using each node as a single source (and we’ll look into that), but there are special-purpose algorithms for that problem as well.]

最短路径问题有很多的变种，比如我们是处理有向图还是无向图上的最短路径问题呢？此外，各个问题之间最大的区别在于起点和终点。这个问题是从一个节点到所有其他节点的最短路径吗(单源最短路径)？还是从一个节点到另一个节点的最短路径(单对节点间最短路径)？还是从所有其他节点到某一个节点(多源最短路径)？还是求任何两个节点之间的最短路径(所有节点对最短路径)？

其中单源最短路径和所有节点对最短路径是最常见的问题类型，其他问题大致可以将其转化成这两类问题。虽然单对节点间最短路径问题有一些求解的技巧(“Meeting in the middle” and “Knowing where you’re going,”)，但是该问题并没有比单源最短路径问题的解法快到哪里去，所以单对节点间最短路径问题可以就用单源最短路径问题的算法去求解；而多源点单终点的最短路径问题可以将边反转过来当成是单源最短路径问题；至于所有节点对最短路径问题，可以对图中的每个节点使用单源最短路径来求解，但是对于这个问题还有一些特殊的更好的算法可以解决。

在开始介绍各种算法之前，作者给出了图中的几个重要结论或者性质，此处附上原文

assume that we start in node s and that we initialize $D[s]$ to zero, while all other distance estimates are set to infinity. Let $d(u,v)$ be the length of the shortest path from u to v .

- $d(s,v) \leq d(s,u) + W[u,v]$. This is an example of the triangle inequality.
- $d(s,v) \leq D[v]$. For v other than s , $D[v]$ is initially infinite, and we reduce it only when we find actual shortcuts. We never “cheat,” so it remains an upper bound.
- If there is no path to node v , then relaxing will never get $D[v]$ below infinity. That’s because we’ll never find any shortcuts to improve $D[v]$.

- Assume a shortest path to v is formed by a path from s to u and an edge from u to v . Now, if $D[u]$ is correct at any time before relaxing the edge from u to v , then $D[v]$ is correct at all times afterward. The path defined by $P[v]$ will also be correct.
- Let $[s, a, b, \dots, z, v]$ be a shortest path from s to v . Assume all the edges $(s, a), (a, b), \dots, (z, v)$ in the path have been relaxed in order. Then $D[v]$ and $P[v]$ will be correct. It doesn't matter if other relax operations have been performed in between.

[最后这个是路径松弛性质，也就是后面的Bellman-Ford算法的核心]

对于单对节点间最短路径问题，如果每条边的权值都一样(或者说边一样长)的话，使用前面的BFS就可以得到结果了([第5节遍历中介绍了](#))；如果图是有向无环图，那么我们还可以用前面动规中的DAG最短路径算法来求解([第8节动态规划中介绍了](#))，但是，现实中的图总是有环的，边的权值也总是不同，而且可能有负权值，所以我们还需要其他的算法！

首先我们来实现下[之前学过的松弛技术relaxtion](#)，代码中 D 保存各个节点到源点的距离值估计(上界值)， P 保存节点的最短路径上的前驱节点， W 保存边的权值，其中不存在的边的权值为 inf 。松弛就是说，假设节点 u 和节点 v 事先都有一个最短距离的估计(例如测试代码中的7和13)，如果现在要松弛边 (u, v) ，也就是对从节点 u 通过边 (u, v) 到达节点 v ，将这条路径得到节点 v 的距离估计值 $(7+3=10)$ 和原来的节点 v 的距离估计值(13)进行比较，如果前者更小的话，就表示我们可以放弃在这之前确定的从源点到节点 v 的最短路径，改成从源点到节点 u ，然后节点 u 再到节点 v ，这条路线距离会更短些，这也就是发生了一次松弛！(测试代码中 $10 < 13$ ，所以要进行松弛，此时 $D[v]$ 变成10，而它的前驱节点也变成了 u)

Python

```
#relaxtion
inf = float('inf')

1 #relaxtion
2 inf = float('inf')
3 def relax(W, u, v, D, P):
4     d = D.get(u,inf) + W[u][v]           # Possible shortcut estimate
5     if d < D.get(v,inf):                 # Is it really a shortcut?
6         D[v], P[v] = d, u               # Update estimate and parent
7         return True                     # There was a change!
8
9 #测试代码
10 u = 0; v = 1
11 D, W, P = {}, {u:{v:3}}, {}
12 D[u] = 7
13 D[v] = 13
14 print D[u] # 7
15 print D[v] # 13
16 print W[u][v] # 3
17 relax(W, u, v, D, P) # True
18 print D[v] # 10
19 D[v] = 8
20 relax(W, u, v, D, P)
21 print D[v] # 8
```

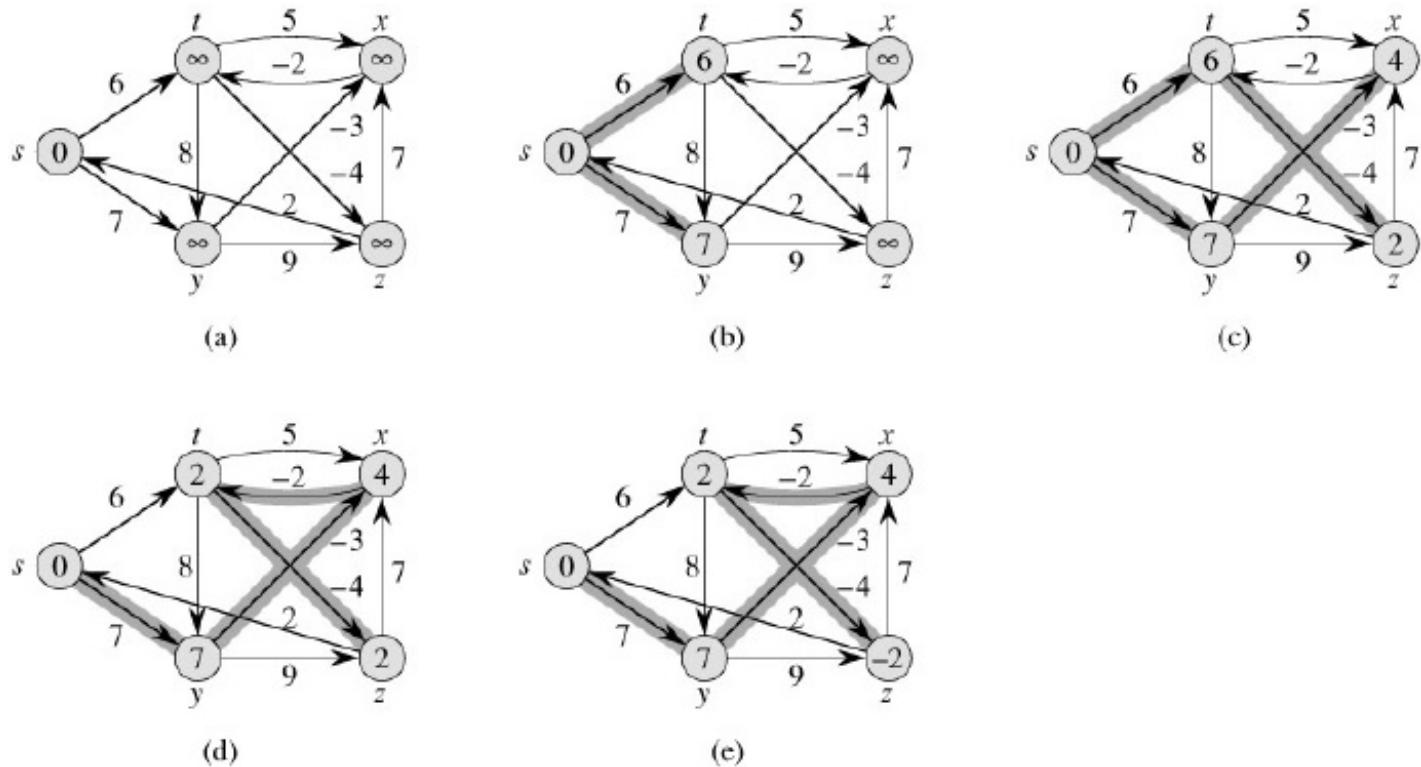
显然，如果你随机地对边进行松弛，那么与该边有关的节点的距离估计值就会慢慢地变得更加准确，这样的改进会在整个图中进行传播，如果一直这么松弛下去的话，最终整个图所有节点的距离值都不会发生变化的时候我们就得到了从源点到所有节点的最短路径值。

每次松弛可以看作是向最终解前进了“一步”，我们的目标自然是希望松弛的次数越少越好，关键就是要确定松弛的次数和松弛的顺序(好的松弛顺序可以让我们直接朝着最优解前进，缩短算法运行时间)，后面要介绍的图中的Bellman-Ford算法、Dijkstra算法以及DAG上的最短路径问题都是如此。

现在我们考虑一个问题，如果我们对图中的所有边都松弛一遍会怎样？可能部分顶点的距离估计值有所减小对吧，那如果再对图中的所有边都松弛一遍又会怎样呢？可能又有部分顶点的距离估计值有所减小对吧，那到底什么时候才会没有改进呢？到底什么时候可以停止了呢？

这个问题可以这么想，假设从源点 s 到节点 v 的最短路径是 $p = \langle v_0, v_1, v_2, v_3 \dots v_k \rangle$ ，此时 $v_0=s, v_k=v$ ，那除了源点 s 之外，这条路径总共经过了其他 k 个顶点对吧， k 肯定小于 $(V-1)$ 对吧，也就是说从节点 s 到节点 v 要经过一条最多只有 $(V-1)$ 条边的路径，因为每遍松弛都是松弛所有边，那么肯定会松弛路径 p 中的所有边，我们可以保险地认为第 i 次循环松弛了边 $\langle v_{i-1}, v_i \rangle$ ，这样的话经过 k 次松弛遍历，我们肯定能够得到节点 v 的最短路径值，再根据这条路径最多只有 $(V-1)$ 条边，也就说明了我们最多只要循环地对图中的所有边都松弛 $(V-1)$ 遍就可以得到所有节点的最短路径值！上面的思路就是 Bellman-Ford 算法了，时间复杂度是 $O(VE)$ 。

下面看下算法导论上的Bellman-Ford算法的示例图



[上图的解释，需要注意的是，如果边的松弛顺序不同，可能中间得到的结果不同，但是最后的结果都是一样的：The execution of the Bellman-Ford algorithm. The source is vertex s . The d values are shown within the vertices, and shaded edges indicate predecessor values: if edge (u, v) is shaded, then $\pi[v] = u$. In this particular example, each pass relaxes the edges in the order (t, x) , (t, y) , (t, z) , (x, t) , (y, x) , (y, z) , (z, x) , (z, s) , (s, t) , (s, y) . (a) The situation just before the first pass over the edges. (b)-(e) The situation after each successive pass over the edges. The d and π values in part (e) are the final values. The Bellman-Ford algorithm returns TRUE in this example.]

上面的分析很好，但是我们漏考虑了一个关键问题，那就是如果图中存在负权回路的话不论我们松弛多少遍，图中有些节点的最短路径值都还是会减小，所以我们在 $(V-1)$ 次松弛遍历之后再松弛遍历一

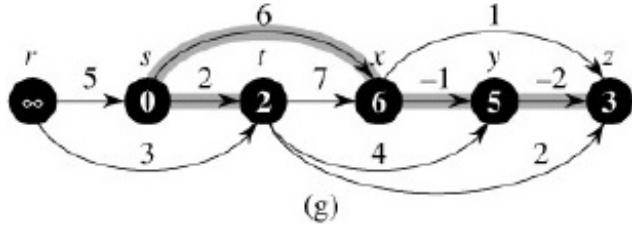
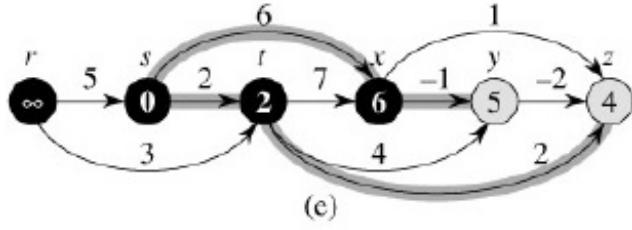
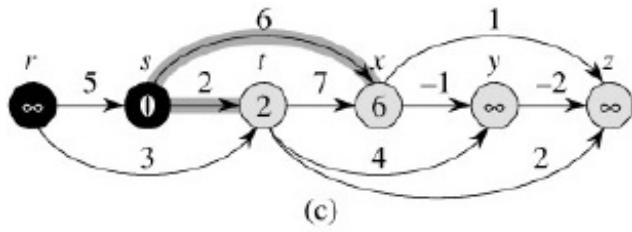
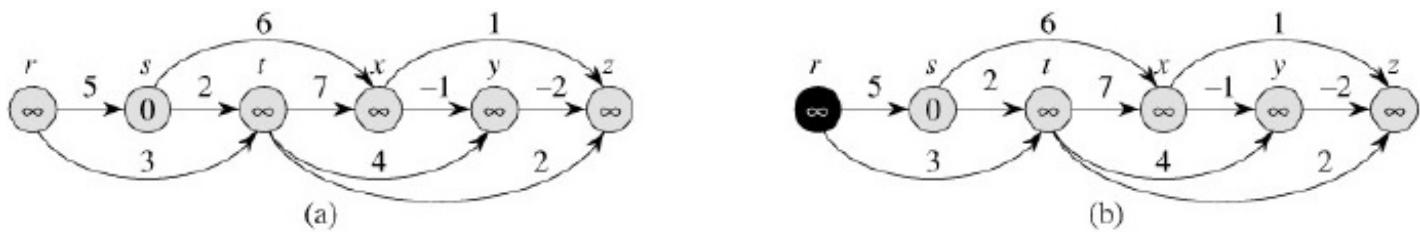
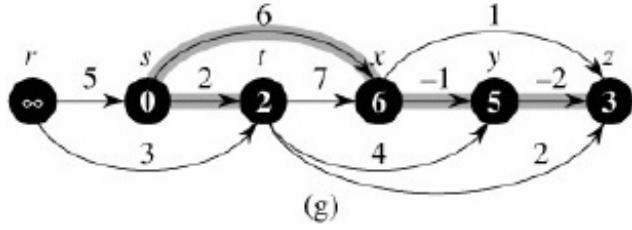
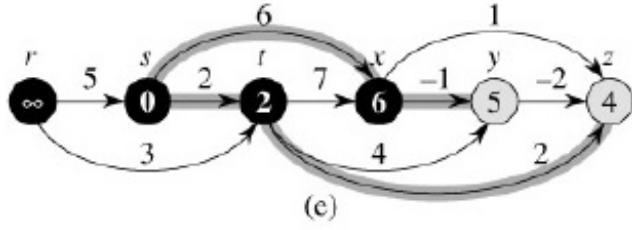
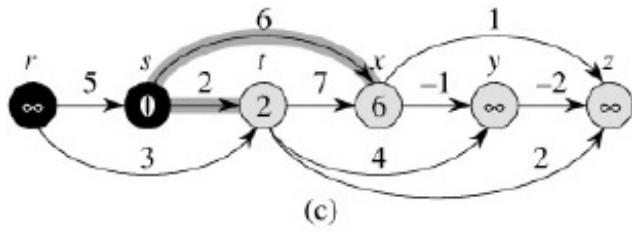
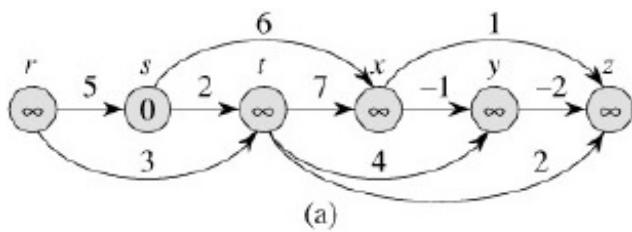
次，如果还有节点的最短路径减小的话就说明图中存在负权回路！这就引出了Bellman-Ford算法的一个重要作用：判断图中是否存在负权回路。

Python

```
#Bellman-Ford算法
def bellman_ford(G, s):
    1 #Bellman-Ford算法
    2 def bellman_ford(G, s):
    3     D, P = {s:0}, {}          # Zero-dist to s; no parents
    4     for rnd in G:            # n = len(G) rounds
    5         changed = False      # No changes in round so far
    6         for u in G:          # For every from-node...
    7             for v in G[u]:    # ... and its to-nodes...
    8                 if relax(G, u, v, D, P): # Shortcut to v from u?
    9                     changed = True # Yes! So something changed
   10                if not changed: break # No change in round: Done
   11            else:               # Not done before round n?
   12                raise ValueError('negative cycle') # Negative cycle detected
   13        return D, P           # Otherwise: D and P correct
   14
   15 #测试代码
   16 s, t, x, y, z = range(5)
   17 W = {
   18     s: {t:6, y:7},
   19     t: {x:5, y:8, z:-4},
   20     x: {t:-2},
   21     y: {x:-3, z:9},
   22     z: {s:2, x:7}
   23 }
   24 D, P = bellman_ford(W, s)
   25 print [D[v] for v in [s, t, x, y, z]] # [0, 2, 4, 7, -2]
   26 print s not in P # True
   27 print [P[v] for v in [t, x, y, z]] == [x, y, s, t] # True
   28 W[s][t] = -100
   29 print bellman_ford(W, s)
   30 # Traceback (most recent call last):
   31 # ...
   32 # ValueError: negative cycle
```

前面我们在动态规划中介绍了一个DAG图中的最短路径算法，它的时间复杂度是 $O(V+E)$ 的，下面我们用松弛的思路来快速回顾一下那个算法的迭代版本。因为它先对顶点进行了拓扑排序，所以它是一个典型的通过修改边松弛的顺序来提高算法运行速度的算法，也就是说，我们不是随机松弛，也不是所有边来松弛一遍，而是沿着拓扑排序得到的节点的顺序来进行松弛，怎么松弛呢？当我们到达一个节点时我们就松弛这个节点的出边，为什么这种方式能够奏效呢？

这里还是假设从源点 s 到节点 v 的最短路径是 $p = \langle v_0, v_1, v_2, v_3 \dots v_k \rangle$ ，此时 $v_0=s, v_k=v$ ，如果我们到达了节点 v ，那么说明源点 s 和节点 v 之间的那些点都已经经过了（节点是经过了拓扑排序的哟），而且它们的边也都已经松弛过了，所以根据路径松弛性质可以知道当我们到达节点 v 时我们能够直接得到源点 s 到节点 v 的最短路径值。



[上图的解释：The execution of the algorithm for shortest paths in a directed acyclic graph. The vertices are topologically sorted from left to right. The source vertex is s . The d values are shown within the vertices, and shaded edges indicate the π values. (a) The situation before the first iteration of the for loop of lines 3-5. (b)-(g) The situation after each iteration of the for loop of lines 3-5. The newly blackened vertex in each iteration was used as u in that iteration. The values shown in part (g) are the final values.]

接下来我们看下Dijkstra算法，它看起来非常像Prim算法，同样是基于贪心策略，每次贪心地选择松弛距离最近的“边缘节点”所在的那条边(另一个节点在已经包含的节点集合中)，那为什么这种方式也能奏效呢？因为算法导论给出了完整的证明，不信你去看看！呵呵，开玩笑的啦，如果光说有证明就用不着我来写文章咯，其实是因为Dijkstra算法隐藏了一个DAG最短路径算法，而DAG的最短路径问题我们上面已经介绍过了，仔细想也不难发现，它们的区别就是松弛的顺序不同，DAG最短路径算法是先进行拓扑排序然后松弛，而Dijkstra算法是每次直接贪心地选择一条边来松弛。那为什么Dijkstra算法隐藏了一个DAG？

这里我想了好久怎么解释，但是还是觉得原文实在太精彩，我想我这有限的水平很难讲明白，故这里附上原文，前面部分作者解释了为什么DAG最短路径算法中边松弛的顺序和拓扑排序有关，然后作者继续解释(Dijkstra算法中)下一个要加入(到已包含的节点集合)的节点必须有正确的距离估计值，最后作者解释了这个节点肯定是那个具有最小距离估计值的节点！一切顺风顺水，但是有一个重要前

提条件，那就是边不能有负权值！]

作者下面的解释中提到的图9-1

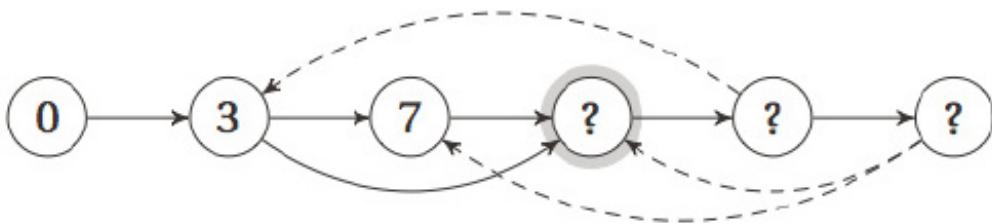


Figure 9-1. Gradually uncovering the hidden DAG. Nodes are labeled with their final distances. Because weights are positive, the backward edges (dashed) cannot influence the result and are therefore irrelevant.

To get things started, we can imagine that we already know the distances from the start node to each of the others. We don't, of course, but this imaginary situation can help our reasoning. Imagine ordering the nodes, left to right, based on their distance. What happens? For the general case—not much. However, we're assuming that we have no negative edge weights, and that makes all the difference.

Because all edges are positive, the only nodes that can contribute to a node's solution will lie to its left in our hypothetical ordering. It will be impossible to locate a node to the right that will help us find a shortcut, because this node is further away, and could only give us a shortcut if it had a negative back edge. The positive back edges are completely useless to us, and aren't part of the problem structure. What remains, then, is a DAG, and the topological ordering we'd like to use is exactly the hypothetical ordering we started with: nodes sorted by their actual distance. See Figure 9-1 for an illustration of this structure. (I'll get back to the question marks in a minute.)

Predictably enough, we now hit the major gap in the solution: it's totally circular. In uncovering the basic problem structure (decomposing into subproblems or finding the hidden DAG), we've assumed that we've already solved the problem. The reasoning has still been useful, though, because we now have something specific to look for. We want to find the ordering—and we can find it with our trusty workhorse, induction!

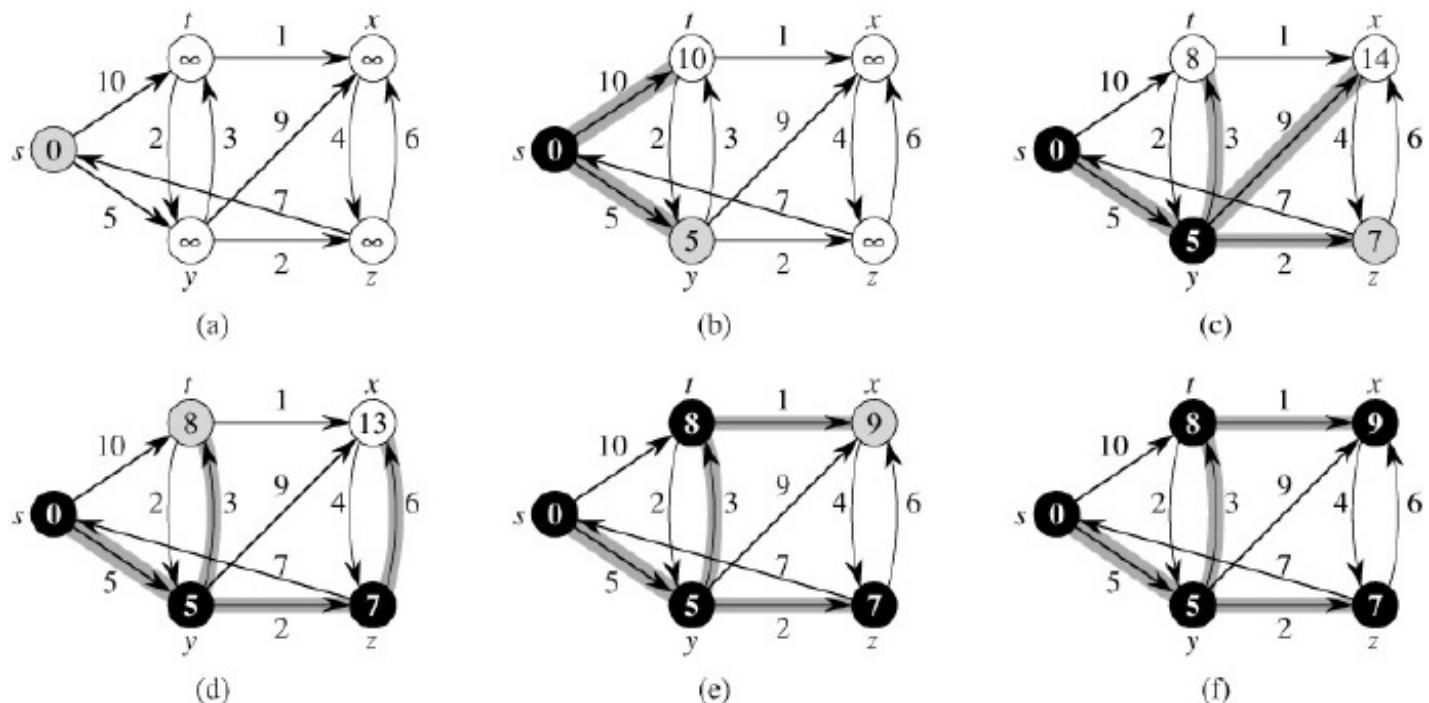
Consider, again, Figure 9-1. Assume that the highlighted node is the one we're trying to identify in our inductive step (meaning that the earlier ones have been identified and already have correct distance estimates). Just like in the ordinary DAG shortest path problem, we'll be relaxing all out-edges for each node, as soon as we've identified it and determined its correct distance. That means that we've relaxed the edges out of all earlier nodes. We haven't relaxed the out-edges of later nodes, but as discussed, they can't matter: the distance estimates of these later nodes are upper bounds, and the back-edges have positive weights, so there's no way they can contribute to a shortcut.

This means (by the earlier relaxation properties or the discussion of the DAG shortest path algorithm in Chapter 8) that the next node must have a correct distance estimate. That is, the highlighted node in Figure 9-1 must by now have received its correct distance estimate, because we've relaxed all edges out of the first three nodes. This is very good news, and all that remains is

to figure out which node it is. We still don't really know what the ordering is, remember? We're figuring out the topological sorting as we go along, step by step.

There is only one node that could possibly be the next one, of course:3 the one with the lowest distance estimate. We know it's next in the sorted order, and we know it has a correct estimate; because these estimates are upper bounds, none of the later nodes could possibly have lower estimates. Cool, no? And now, by induction, we've solved the problem. We just relax all out-edges of the nodes of each node in distance order—which means always taking the one with the lowest estimate next.

下图是算法导论中Dijkstra算法的示例图，可以参考下



[上图的解释：The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates are shown within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S , and white vertices are in the min-priority queue $Q = V - S$. (a) The situation just before the first iteration of the while loop of lines 4-8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. (b)-(f) The situation after each successive iteration of the while loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d and π values shown in part (f) are the final values.]

下面是Dijkstra算法的实现

Python

```
#Dijkstra算法
from heapq import heappush, heappop
1 #Dijkstra算法
2 from heapq import heappush, heappop
3
4 def dijkstra(G, s):
```

```

5   D, P, Q, S = {s:0}, {}, [(0,s)], set()    # Est., tree, queue, visited
6   while Q:                                     # Still unprocessed nodes?
7       _, u = heappop(Q)                         # Node with lowest estimate
8       if u in S: continue                       # Already visited? Skip it
9       S.add(u)                                 # We've visited it now
10      for v in G[u]:                           # Go through all its neighbors
11          relax(G, u, v, D, P)                  # Relax the out-edge
12          heappush(Q, (D[v], v))               # Add to queue, w/est. as pri
13      return D, P                             # Final D and P returned
14
15 #测试代码
16 s, t, x, y, z = range(5)
17 W = {
18     s: {t:10, y:5},
19     t: {x:1, y:2},
20     x: {z:4},
21     y: {t:3, x:9, z:2},
22     z: {x:6, s:7}
23 }
24 D, P = dijkstra(W, s)
25 print [D[v] for v in [s, t, x, y, z]] # [0, 8, 9, 5, 7]
26 print s not in P # True
27 print [P[v] for v in [t, x, y, z]] == [y, t, s, y] # True

```

Dijkstra算法和Prim算法的实现很像，也和BFS算法实现很像，其实，如果我们把每条权值为 w 的边 (u,v) 想象成节点 u 和节点 v 中间有 $(w-1)$ 个节点，且每条边都是权值为1的一条路径的话，BFS算法其实就和Dijkstra算法差不多了。 Dijkstra算法的时间复杂度和使用的优先队列有关，上面的实现用的是最小堆，所以时间复杂度是 $O(m\lg n)$ ，其中 m 是边数， n 是节点数。

下面我们来看看所有点对最短路径问题

对于所有点对最短路径问题，我们第一个想法肯定是对每个节点运行一遍Dijkstra算法就可以了嘛，但是，Dijkstra算法有个前提条件，所有边的权值都是正的，那些包含了负权边的图怎么办？那就想办法对图进行些预处理，使得所有边的权值都是正的就可以了，那怎么处理能够做到呢？此时可以看下前面的三角不等性质，内容如下：

$d(s,v) \leq d(s,u) + w[u,v]$. This is an example of the triangle inequality.

令 $h(u)=d(s,u)$, $h(v)=d(s,v)$ ，假设我们给边 (u,v) 重新赋权 $w'(u, v) = w(u, v) + h(u) - h(v)$ ，根据三角不等性质可知 $w'(u, v)$ 肯定非负，这样新图的边就满足Dijkstra算法的前提条件，但是，我们怎么得到每个节点的最短路径值 $d(s,v)$ ？

其实这个问题很好解决对吧，前面介绍的Bellman-Ford算法就干这行的，但是源点 s 是什么？这里的解决方案有点意思，我们可以向图中添加一个顶点 s ，并且让它连接图中的所有其他节点，边的权值都是0，完了之后我们就可以在新图上从源点 s 开始运行Bellman-Ford算法，这样就得到了每个节点的最短路径值 $d(s,v)$ 。但是，新的问题又来了，这么改了之后真的好吗？得到的最短路径对吗？

这里的解释更加有意思，想想任何一条从源点 s 到节点 v 的路径 $p = <s, v_1, v_2, v_3 \dots u, v>$ ，假设我们把路径上的边权值都加起来的话，你会发现下面的有意思的现象(telescoping sums)：

$$\text{sum} = [w(s, v_1) + h(s) - h(v_1)] + [w(v_1, v_2) + h(v_1) - h(v_2)] + \dots + [w(u, v) + h(u) - h(v)] = w(v_1, v_2) + w(v_2, v_3) + \dots + w(u, v) - h(v)$$

上面的式子说明，所有从源点 s 到节点 v 的路径都会减去 $h(v)$ ，也就说明对于新图上的任何一条最短

路径，它都是对应着原图的那条最短路径，只是路径的权值减去了 $h(v)$ ，这也就说明采用上面的策略得到的最短路径没有问题。

现在我们捋一捋思路，我们首先要使用Bellman-Ford算法得到每个节点的最短路径值，然后利用这些值修改图中边的权值，最后我们对图中所有节点都运行一次Dijkstra算法就解决了所有节点对最短路径问题，但是如果原图本来边的权值就都是正的话就直接运行Dijkstra算法就行了。这就是Johnson算法，一个巧妙地利用Bellman-Ford和Dijkstra算法结合来解决所有节点对最短路径问题的算法。它特别适合用于稀疏图，算法的时间复杂度是 $O(mn\lg n)$ ，比后面要介绍的Floyd-Warshall算法要好些。

还有一点需要补充的是，在运行完了Dijkstra算法之后，如果我们要得到准确的最短路径的权值的话，我们还需要做一定的修改，从前面的式子可以看出，新图上节点 u 和节点 v 之间的最短路径 $D'(u,v)$ 与原图上两个节点的最短路径 $D(u,v)$ 有如下左式的关系，那么经过右式的简单计算就能得到原图的最短路径值

$$D'(u,v) = D(u,v) + h(u) - h(v) \implies D(u,v) = D'(u,v) - h(u) + h(v)$$

基于上面的思路，我们可以得到下面的Johnson算法实现

Python

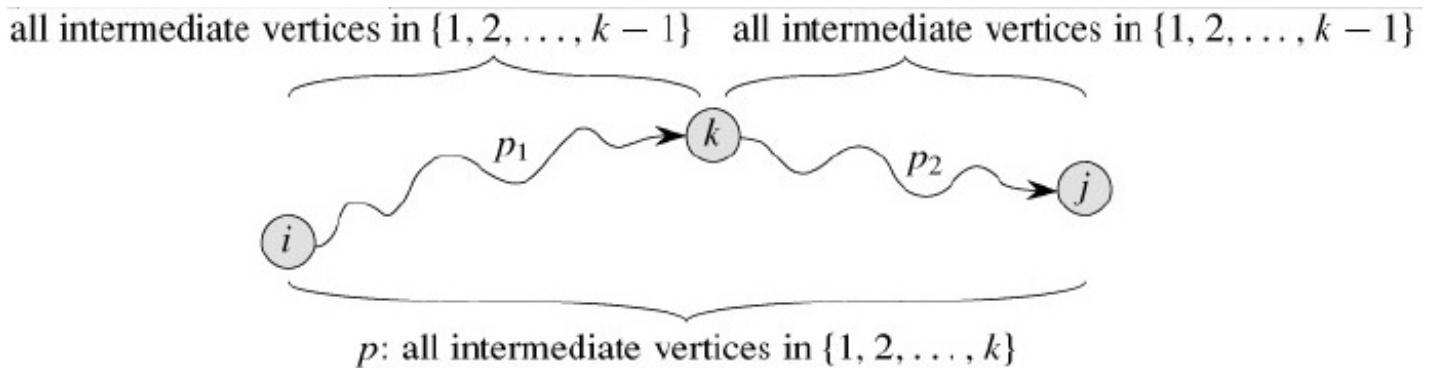
```
#Johnson's Algorithm
def johnson(G):
    1 #Johnson's Algorithm
    2 def johnson(G):
    3     G = deepcopy(G)
    4     s = object()
    5     G[s] = {v:0 for v in G}
    6     h, _ = bellman_ford(G, s)
    7     del G[s]
    8     for u in G:
    9         for v in G[u]:
   10             G[u][v] += h[u] - h[v]
   11     D, P = {}, {}
   12     for u in G:
   13         D[u], P[u] = dijkstra(G, u)
   14         for v in G:
   15             D[u][v] += h[v] - h[u]
   16     return D, P
   17
   18 a, b, c, d, e = range(5)
   19 W = {
   20     a: {c:1, d:7},
   21     b: {a:4},
   22     c: {b:-5, e:2},
   23     d: {c:6},
   24     e: {a:3, b:8, d:-4}
   25 }
   26 D, P = johnson(W)
   27 print [D[a][v] for v in [a, b, c, d, e]] # [0, -4, 1, -1, 3]
   28 print [D[b][v] for v in [a, b, c, d, e]] # [4, 0, 5, 3, 7]
   29 print [D[c][v] for v in [a, b, c, d, e]] # [-1, -5, 0, -2, 2]
   30 print [D[d][v] for v in [a, b, c, d, e]] # [5, 1, 6, 0, 8]
   31 print [D[e][v] for v in [a, b, c, d, e]] # [1, -3, 2, -4, 0]
```

下面我们看下Floyd-Warshall算法，这是一个基于动态规划的算法，时间复杂度是 $O(n^3)$ ， n 是图中节点数

假设所有节点都有一个数字编号(从1开始), 我们要把原来的问题reduce成一个个子问题, 子问题有三个参数: 起点 u、终点 v、能经过的节点的最大编号k, 也就是求从起点 u 到终点 v 只能够经过编号为(1,2,3,...,k)的节点的最短路径问题 (原文表述如下)

Let $d(u, v, k)$ be the length of the shortest path that exists from node u to node v if you're only allowed to use the k first nodes as intermediate nodes.

这个子问题怎么考虑呢? 当然还是采用之前动态规划中常用的选择还是不选择这种策略, 如果我们选择不经过节点 k 的话, 那么问题变成了求从起点 u 到终点 v 只能够经过编号为(1,2,3,...,k-1)的节点的最短路径问题; 如果我们选择经过节点 k 的话, 那么问题变成求从起点 u 到终点 k 只能够经过编号为(1,2,3,...,k-1)的节点的最短路径问题与求从起点 k 到终点 v 只能够经过编号为(1,2,3,...,k-1)的节点的最短路径问题之和, 如下图所示



经过上面的分析, 我们可以得到下面的结论

$$d(u, v, k) = \min(d(u, v, k-1), d(u, k, k-1) + d(k, v, k-1))$$

根据这个式子我们很快可以得到下面的递归实现

Python

```
#递归版本的Floyd-Warshall算法
# All shortest paths
# Store subsolutions
# u to v via 1..k
# Assumes v in G[u]
# Use k or not?
# D[u,v] = d(u,v,n)

1 #递归版本的Floyd-Warshall算法
2 from functools import wraps
3
4 def memo(func):
5     cache = {}          # Stored subproblem solutions
6     @wraps(func)        # Make wrap look like func
7     def wrap(*args):    # The memoized wrapper
8         if args not in cache: # Not already computed?
9             cache[args] = func(*args) # Compute & cache the solution
10            return cache[args] # Return the cached solution
11        return wrap # Return the wrapper
12
13 def rec_floyd_marshall(G): # All shortest paths
14     @memo # Store subsolutions
15     def d(u,v,k): # u to v via 1..k
16         if k==0: return G[u][v] # Assumes v in G[u]
17         return min(d(u,v,k-1), d(u,k,k-1) + d(k,v,k-1)) # Use k or not?
18     return {(u,v): d(u,v,len(G)) for u in G for v in G} # D[u,v] = d(u,v,n)
19
20 #测试代码
```

```

21 a, b, c, d, e = range(1,6) # One-based
22 W = {
23     a: {c:1, d:7},
24     b: {a:4},
25     c: {b:-5, e:2},
26     d: {c:6},
27     e: {a:3, b:8, d:-4}
28 }
29 for u in W:
30     for v in W:
31         if u == v: W[u][v] = 0
32         if v not in W[u]: W[u][v] = inf
33 D = rec_floyd_warshall(W)
34 print [D[a,v] for v in [a, b, c, d, e]] # [0, -4, 1, -1, 3]
35 print [D[b,v] for v in [a, b, c, d, e]] # [4, 0, 5, 3, 7]
36 print [D[c,v] for v in [a, b, c, d, e]] # [-1, -5, 0, -2, 2]
37 print [D[d,v] for v in [a, b, c, d, e]] # [5, 1, 6, 0, 8]
38 print [D[e,v] for v in [a, b, c, d, e]] # [1, -3, 2, -4, 0]

```

仔细看的话，不难发现这个解法和我们介绍动态规划时介绍的最长公共子序列的问题非常类似，[如果还没有阅读的话不妨看下最长公共子序列问题的5种实现这篇文章](#)，有了对最长公共子序列问题的理解，我们就很容易发现对于Floyd-Warshall算法我们也可以采用类似的方式来减小算法所需占用的空间，当然首先要将递归版本改成性能更好些的迭代版本。

Floyd-Warshall算法的递推公式

$$d_{ij}^k = \begin{cases} \omega_{ij} & \text{如果 } k=0 \\ \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}) & \text{如果 } k \geq 1 \end{cases}$$

从递推公式中可以看出，计算当前回合(k)只需要上一回合($k-1$)得到的结果，所以，如果应用对于中间结果不需要的话，那么可以只使用2个 $n \times n$ 的矩阵，一个保存当前回合(k)的结果 $D(k)$ ，另一个保存上一回合($k-1$)的结果 $D(k-1)$ ，待当前回合计算完了之后将其全部复制到 $D(k-1)$ 中，这样就仅需要 $O(n^2)$ 的空间。

Python

```

#空间优化后的Floyd-Warshall算法
1 #空间优化后的Floyd-Warshall算法
2 def floyd_warshall1(G):
3     D = deepcopy(G)           # No intermediates yet
4     for k in G:               # Look for shortcuts with k
5         for u in G:
6             for v in G:
7                 D[u][v] = min(D[u][v], D[u][k] + D[k][v])
8     return D
9
10 #测试代码
11 a, b, c, d, e = range(1,6) # One-based
12 W = {
13     a: {c:1, d:7},
14     b: {a:4},
15     c: {b:-5, e:2},
16     d: {c:6},
17     e: {a:3, b:8, d:-4}

```

```

18 }
19 for u in W:
20   for v in W:
21     if u == v: W[u][v] = 0
22     if v not in W[u]: W[u][v] = inf
23 D = floyd_warshall(W)
24 print [D[a][v] for v in [a, b, c, d, e]] # [0, -4, 1, -1, 3]
25 print [D[b][v] for v in [a, b, c, d, e]] # [4, 0, 5, 3, 7]
26 print [D[c][v] for v in [a, b, c, d, e]] # [-1, -5, 0, -2, 2]
27 print [D[d][v] for v in [a, b, c, d, e]] # [5, 1, 6, 0, 8]
28 print [D[e][v] for v in [a, b, c, d, e]] # [1, -3, 2, -4, 0]

```

当然啦，一般情况下求最短路径问题我们还需要知道最短路径是什么，这个时候我们只需要在进行选择的时候设置一个前驱节点就行了

Python

```
#最终版本的Floyd-Warshall算法
```

```

1 #最终版本的Floyd-Warshall算法
2 def floyd_warshall(G):
3   D, P = deepcopy(G), {}
4   for u in G:
5     for v in G:
6       if u == v or G[u][v] == inf:
7         P[u,v] = None
8       else:
9         P[u,v] = u
10  for k in G:
11    for u in G:
12      for v in G:
13        shortcut = D[u][k] + D[k][v]
14        if shortcut < D[u][v]:
15          D[u][v] = shortcut
16          P[u,v] = P[k,v]
17  return D, P
18
19 #测试代码
20 a, b, c, d, e = range(5)
21 W = {
22   a: {c:1, d:7},
23   b: {a:4},
24   c: {b:-5, e:2},
25   d: {c:6},
26   e: {a:3, b:8, d:-4}
27 }
28 for u in W:
29   for v in W:
30     if u == v: W[u][v] = 0
31     if v not in W[u]: W[u][v] = inf
32 D, P = floyd_warshall(W)
33 print [D[a][v] for v in [a, b, c, d, e]]#[0, -4, 1, -1, 3]
34 print [D[b][v] for v in [a, b, c, d, e]]#[4, 0, 5, 3, 7]
35 print [D[c][v] for v in [a, b, c, d, e]]#[-1, -5, 0, -2, 2]
36 print [D[d][v] for v in [a, b, c, d, e]]#[5, 1, 6, 0, 8]
37 print [D[e][v] for v in [a, b, c, d, e]]#[1, -3, 2, -4, 0]
38 print [P[a,v] for v in [a, b, c, d, e]]#[None, 2, 0, 4, 2]
39 print [P[b,v] for v in [a, b, c, d, e]]#[1, None, 0, 4, 2]
40 print [P[c,v] for v in [a, b, c, d, e]]#[1, 2, None, 4, 2]
41 print [P[d,v] for v in [a, b, c, d, e]]#[1, 2, 3, None, 2]
42 print [P[e,v] for v in [a, b, c, d, e]]#[1, 2, 3, 4, None]

```

[算法导论在介绍所有节点对最短路径问题时先介绍了另一个基于动态规划的解法，但是那个算法时间复杂度较高，即使是使用了重复平方技术还是比较差，所以这里不介绍了，但是有意思的是书中将这个算法和矩阵乘法运算进行了对比，发现两者之间惊人的相似，其实同理，我们开始介绍的Bellman-Ford算法和矩阵与向量的乘法运算也有很多类似的地方，感兴趣可以自己探索下，也可以阅读算法导论了解下]

本章节最后作者还提出了两个用来解最短路径问题的技巧：“Meeting in the middle” 和 “Knowing where you're going,”，这部分的内容又都比较难翻译和理解，感兴趣还是阅读原文较好

(1)Meeting in the middle

简单来说就是双向进行，Dijkstra算法是从节点 s 出发去找到达节点 t 的最短路径，但是，如果两个节点同时进行呢，当它们找到相同的节点时就得到一条路径了，这种方式比一个方向查找的效率要高些，下图是一个图示

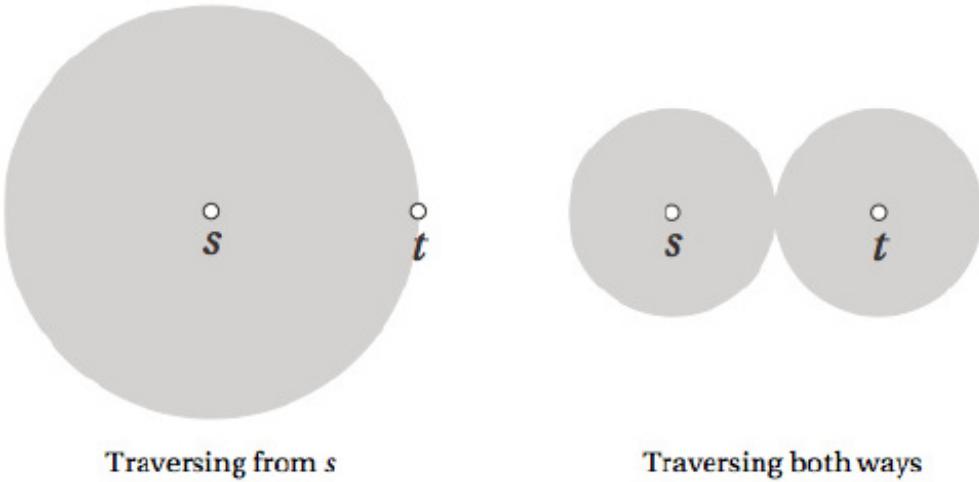


Figure 9-3. Unidirectional and bidirectional “ripples,” indicating the work needed to find a path from s to t by traversal.

(2)Knowing where you're going

这里作者介绍了大名鼎鼎的A*算法，实际上也就非常类似采用了分支限界策略的BFS算法(the best-first search used in the branch and bound strategy)。

By now you've seen that the basic idea of traversal is pretty versatile, and by simply using different queues, you get several useful algorithms. For example, for FIFO and LIFO queues, you get BFS and DFS, and with the appropriate priorities, you get the core of Prim's and Dijkstra's algorithms. The algorithm described in this section, called A*, extends Dijkstra's, by tweaking the priority once again.

As mentioned earlier, the A* algorithm uses an idea similar to Johnson's algorithm, although for a different purpose. Johnson's algorithm transforms all edge weights to ensure they're positive, while ensuring that the shortest paths are still shortest. In A*, we want to modify the edges in a similar fashion, but this time the goal isn't to make the edges positive—we're assuming they already are (as we're building on Dijkstra's algorithm). No, what we want is to guide the traversal in the right

direction, by using information of where we're going: we want to make edges moving away from our target node more expensive than those that take us closer to it.

练习题：来自算法导论24-3 货币兑换问题

Problems 24-3: Arbitrage

Arbitrage is the use of discrepancies in currency exchange rates to transform one unit of a currency into more than one unit of the same currency. For example, suppose that 1 U.S. dollar buys 46.4 Indian rupees, 1 Indian rupee buys 2.5 Japanese yen, and 1 Japanese yen buys 0.0091 U.S. dollars. Then, by converting currencies, a trader can start with 1 U.S. dollar and buy $46.4 \times 2.5 \times 0.0091 = 1.0556$ U.S. dollars, thus turning a profit of 5.56 percent.

Suppose that we are given n currencies c_1, c_2, \dots, c_n and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j .

- Give an efficient algorithm to determine whether or not there exists a sequence of currencies $(c_{i_1}, c_{i_2}, \dots, c_{i_k})$ such that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdot R[i_3, i_4] \cdots R[i_k, i_1] > 1.$$

Analyze the running time of your algorithm.

- Give an efficient algorithm to print out such a sequence if one exists. Analyze the running time of your algorithm.

简单来说就是在给定的不同货币的兑换率下是否存在一个货币兑换循环使得最终我们能够从中获利？

[提示：Bellman-Ford算法]

Solution to Problem 24-3

- a. We can use the Bellman-Ford algorithm on a suitable weighted, directed graph $G = (V, E)$, which we form as follows. There is one vertex in V for each currency, and for each pair of currencies c_i and c_j , there are directed edges (v_i, v_j) and (v_j, v_i) . (Thus, $|V| = n$ and $|E| = \binom{n}{2}$.)

To determine edge weights, we start by observing that

$$R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$$

if and only if

$$\frac{1}{R[i_1, i_2]} \cdot \frac{1}{R[i_2, i_3]} \cdots \frac{1}{R[i_{k-1}, i_k]} \cdot \frac{1}{R[i_k, i_1]} < 1.$$

Taking logs of both sides of the inequality above, we express this condition as

$$\lg \frac{1}{R[i_1, i_2]} + \lg \frac{1}{R[i_2, i_3]} + \lg \frac{1}{R[i_{k-1}, i_k]} + \cdots + \lg \frac{1}{R[i_k, i_1]} < 0.$$

Therefore, if we define the weight of edge (v_i, v_j) as

$$\begin{aligned} w(v_i, v_j) &= \lg \frac{1}{R[i, j]} \\ &= -\lg R[i, j], \end{aligned}$$

then we want to find whether there exists a negative-weight cycle in G with these edge weights.

We can determine whether there exists a negative-weight cycle in G by adding an extra vertex v_0 with 0-weight edges (v_0, v_i) for all $v_i \in V$, running BELLMAN-FORD from v_0 , and using the boolean result of BELLMAN-FORD (which is TRUE if there are no negative-weight cycles and FALSE if there is a

negative-weight cycle) to guide our answer. That is, we invert the boolean result of BELLMAN-FORD.

This method works because adding the new vertex v_0 with 0-weight edges from v_0 to all other vertices cannot introduce any new cycles, yet it ensures that all negative-weight cycles are reachable from v_0 .

It takes $\Theta(n^2)$ time to create G , which has $\Theta(n^2)$ edges. Then it takes $O(n^3)$ time to run BELLMAN-FORD. Thus, the total time is $O(n^3)$.

Another way to determine whether a negative-weight cycle exists is to create G and, without adding v_0 and its incident edges, run either of the all-pairs shortest-paths algorithms. If the resulting shortest-path distance matrix has any negative values on the diagonal, then there is a negative-weight cycle.

- b.** Assuming that we ran BELLMAN-FORD to solve part (a), we only need to find the vertices of a negative-weight cycle. We can do so as follows. First, relax all the edges once more. Since there is a negative-weight cycle, the d value of some vertex u will change. We just need to repeatedly follow the π values until we get back to u . In other words, we can use the recursive method given by the PRINT-PATH procedure of Section 22.2, but stop it when it returns to vertex u .

The running time is $O(n^3)$ to run BELLMAN-FORD, plus $O(n)$ to print the vertices of the cycle, for a total of $O(n^3)$ time.

1 赞 1 收藏 [评论](#)

Python算法：动态规划

原文出处：[hujiawei \(@五道口宅男\)](#)

本节主要结合一些经典的动规问题介绍动态规划的备忘录法和迭代法这两种实现方式，并对这两种方式进行对比

[这篇文章实际写作时间在这个系列文章之前，所以写作风格可能略有不同，嘿嘿]

大家都知道，动态规划算法一般都有下面两种实现方式，前者我称为递归版本，后者称为迭代版本，根据前面的知识可知，这两个版本是可以相互转换的

1.直接自顶向下实现递归式，并将中间结果保存，这叫备忘录法；

2.按照递归式自底向上地迭代，将结果保存在某个数据结构中求解。

编程有一个原则DRY=Don't Repeat Yourself，就是说你的代码不要重复来重复去的，这个原则同样可以用于理解动态规划，动态规划除了满足最优子结构，它还存在子问题重叠的性质，我们不能重复地去解决这些子问题，所以我们将子问题的解保存起来，类似缓存机制，之后遇到这个子问题时直接取出子问题的解。

举个简单的例子，斐波那契数列中的元素的计算，很简单，我们写下如下的代码：

Python

```
def fib(i):
    if i<2: return 1
1 def fib(i):
2     if i<2: return 1
3     return fib(i-1)+fib(i-2)
```

好，来测试下，运行`fib(10)`得到结果69，不错，速度也还行，换个大的数字，试试100，这时你会发现，这个程序执行不出结果了，为什么？递归太深了！要计算的子问题太多了！

所以，我们需要改进下，我们保存每次计算出来的子问题的解，用什么保存呢？用Python中的dict！那怎么实现保存子问题的解呢？用Python中的装饰器！

如果不是很了解Python的装饰器，可以快速看下[这篇总结中关于装饰器的解释：Python Basics](#)

修改刚才的程序，得到如下代码，定义一个函数`memo`返回我们需要的装饰器，这里用`cache`保存子问题的解，`key`是方法的参数，也就是数字`n`，值就是`fib(n)`返回的解。

Python

```
from functools import wraps
1 from functools import wraps
2
3 def memo(func):
4     cache={}
```

```

5     @wraps(func)
6     def wrap(*args):
7         if args not in cache:
8             cache[args]=func(*args)
9         return cache[args]
10    return wrap
11
12 @memo
13 def fib(i):
14     if i<2: return 1
15     return fib(i-1)+fib(i-2)

```

重新运行下`fib(100)`，你会发现这次很快就得到了结果573147844013817084101，这就是动态规划的威力，上面使用的是第一种带备忘录的递归实现方式。

带备忘录的递归方式的优点就是易于理解，易于实现，代码简洁干净，运行速度也不错，直接从需要求解的问题出发，而且只计算需要求解的子问题，没有多余的计算。但是，它也有自己的缺点，因为是递归形式，所以有限的栈深度是它的硬伤，有些问题难免会出现栈溢出了。

于是，迭代版本的实现方式就诞生了！

迭代实现方式有2个好处：1.运行速度快，因为没有用栈去实现，也避免了栈溢出的情况；2.迭代实现的话可以不使用dict来进行缓存，而是使用其他的特殊cache结构，例如多维数组等更为高效的数据结构。

那怎么把递归版本转变成迭代版本呢？

这就是递归实现和迭代实现的重要区别：递归实现不需要去考虑计算顺序，只要给出问题，然后自顶向下去解就行；而迭代实现需要考虑计算顺序，并且顺序很重要，算法在运行的过程中要保证当前要计算的问题中的子问题的解已经是求解好了的。

斐波那契数列的迭代版本很简单，就是按顺序来计算就行了，不解释，关键是你可以看到我们就用了3个简单变量就求解出来了，没有使用任何高级的数据结构，节省了大量的空间。

Python

```

def fib_iter(n):
    if n<2: return 1
1 def fib_iter(n):
2     if n<2: return 1
3     a,b=1,1
4     while n>=2:
5         c=a+b
6         a=b
7         b=c
8         n=n-1
9     return c

```

斐波那契数列的变种经常出现在上楼梯的走法问题中，每次只能走一个台阶或者两个台阶，广义上思考的话，动态规划也就是一个连续决策问题，到底当前这一步是选择它(走一步)还是不选择它(走两步)呢？

其他问题也可以很快地变相思考发现它们其实是一样的，例如求二项式系数 $C(n,k)$ ，杨辉三角(求从源

点到目标点有多少种走法)等等问题。

二项式系数 $C(n,k)$ 表示从 n 个中选 k 个，假设我们现在处理 n 个中的第1个，考虑是否选择它。如果选择它的话，那么我们还需要从剩下的 $n-1$ 个中选 $k-1$ 个，即 $C(n-1,k-1)$ ；如果不选择它的话，我们需要从剩下的 $n-1$ 中选 k 个，即 $C(n-1,k)$ 。所以， $C(n,k)=C(n-1,k-1)+C(n-1,k)$ 。

结合前面的装饰器，我们很快便可以实现求二项式系数的递归实现代码，其中的memo函数完全没变，只是在函数cnk前面添加了@memo而已，就这么简单！

Python

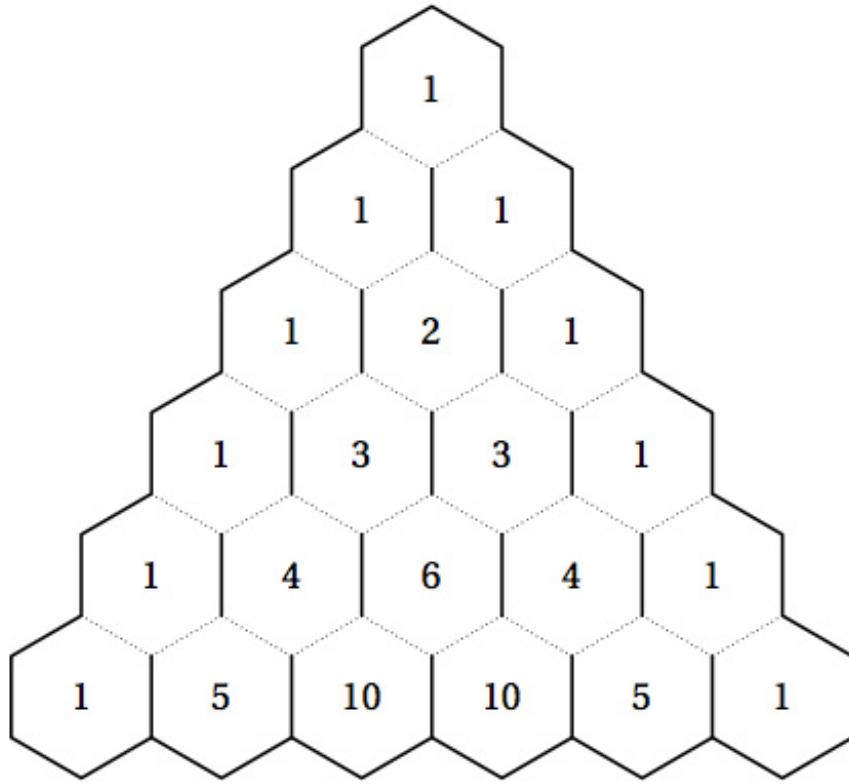
```
from functools import wraps
1 from functools import wraps
2
3 def memo(func):
4     cache={}
5     @wraps(func)
6     def wrap(*args):
7         if args not in cache:
8             cache[args]=func(*args)
9         return cache[args]
10    return wrap
11
12 @memo
13 def cnk(n,k):
14     if k==0: return 1 #the order of `if` should not change!!!
15     if n==0: return 0
16     return cnk(n-1,k)+cnk(n-1,k-1)
```

它的迭代版本也比较简单，这里使用了defaultdict，略高级的数据结构，和dict不同的是，当查找的key不存在对应的value时，会返回一个默认的值，这个很有用，下面的代码可以看到。如果不了解defaultdict的话可以看下[Python中的高级数据结构](#)

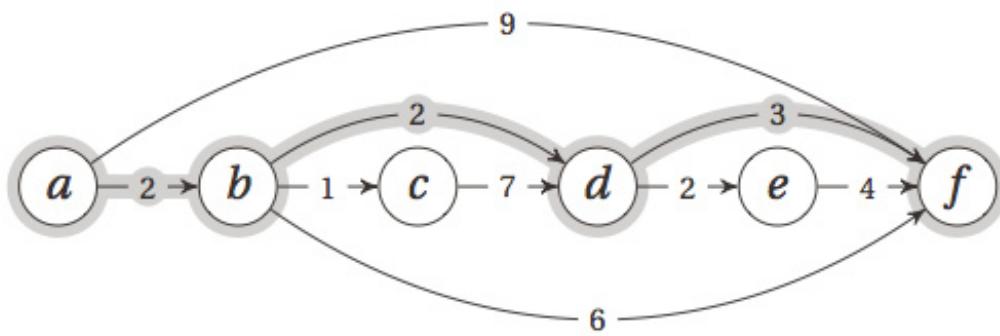
Python

```
from collections import defaultdict
1 from collections import defaultdict
2
3 n,k=10,7
4 C=defaultdict(int)
5 for row in range(n+1):
6     C[row,0]=1
7     for col in range(1,k+1):
8         C[row,col]=C[row-1,col-1]+C[row-1,col]
9
10 print(C[n,k]) #120
```

杨辉三角大家都熟悉，在国外这个叫Pascal Triangle，它和二项式系数特别相似，看下图，除了两边的数字之外，里面的任何一个数字都是由它上面相邻的两个元素相加得到，想想 $C(n,k)=C(n-1,k-1)+C(n-1,k)$ 不也就是这个含义吗？



所以说，顺序对于迭代版本的动态规划实现很重要，下面举个实例，用动态规划解决有向无环图的单源最短路径问题。假设有如下图所示的图，当然，我们看到的是这个有向无环图经过了拓扑排序之后的结果，从a到f的最短路径用灰色标明了。



好，怎么实现呢？

我们有两种思考方式：

1.”去哪里？”：我们顺向思维，首先假设从a点出发到所有其他点的距离都是无穷大，然后，按照拓扑排序的顺序，从a点出发，接着更新a点能够到达的其他的点的距离，那么就是b点和f点，b点的距离变成2，f点的距离变成9。因为这个有向无环图是经过了拓扑排序的，所以按照拓扑顺序访问一遍所有的点(到了目标点就可以停止了)就能够得到a点到所有已访问到的点的最短距离，也就是说，当到达哪个点的时候，我们就找到了从a点到该点的最短距离，拓扑排序保证了后面的点不会指向前面的点，所以访问到后面的点时不可能再更新它前面的点的最短距离！(这里的更新也就是[前面第4节介绍过的relaxtion](#))这种思维方式的代码实现就是迭代版本。

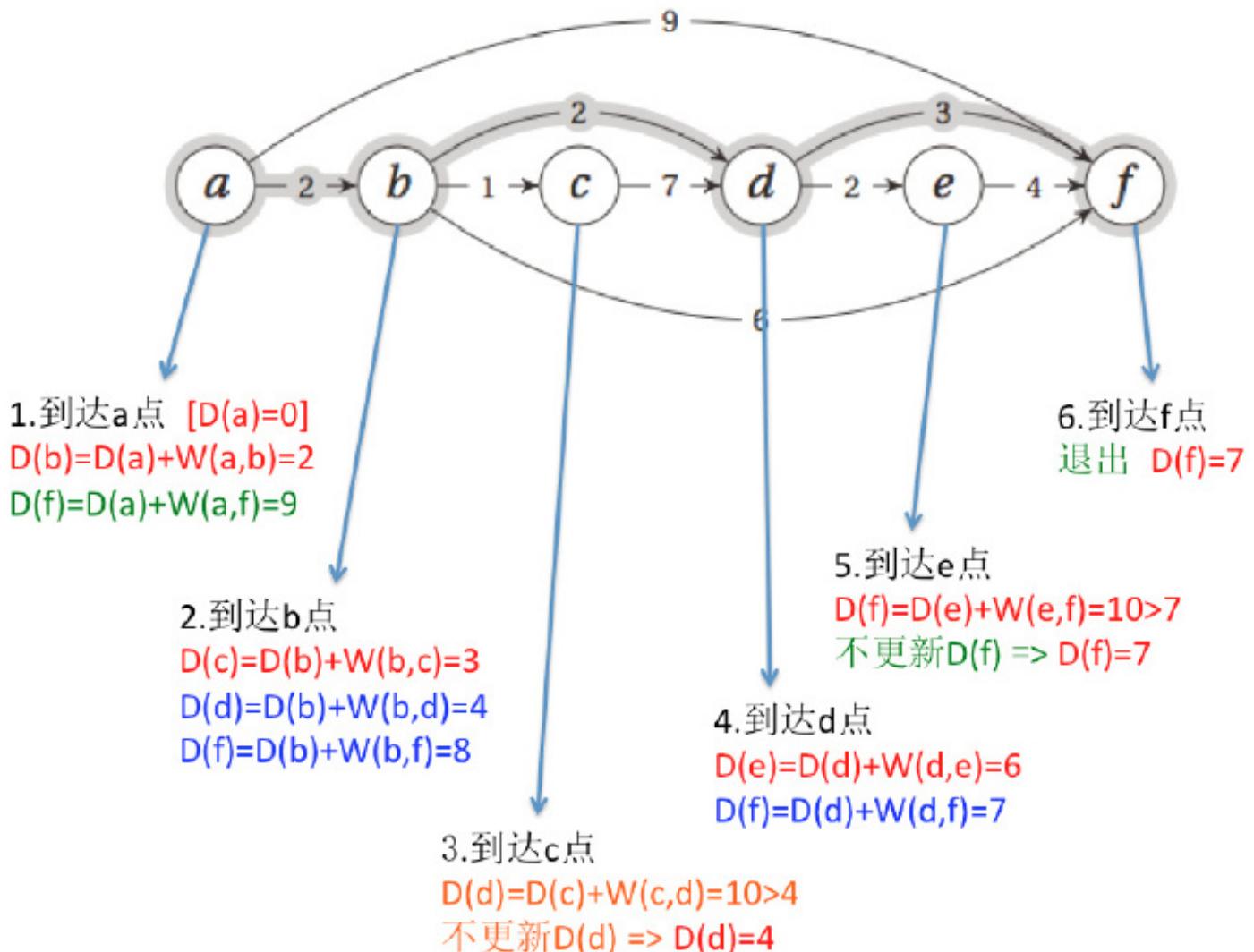
[这里涉及到了拓扑排序，前面第5节Traversal中介绍过了，这里为了方便没看前面的童鞋理解，W直接使用的是经过拓扑排序之后的结果。]

Python

```
def topsort(W):
    return W

1 def topsort(W):
2     return W
3
4 def dag_sp(W, s, t):
5     d = {u:float('inf') for u in W} #
6     d[s] = 0
7     for u in topsort(W):
8         if u == t: break
9         for v in W[u]:
10            d[v] = min(d[v], d[u] + W[u][v])
11    return d[t]
12
13 #邻接表
14 W={0:{1:2,5:9},1:{2:1,3:2,5:6},2:{3:7},3:{4:2,5:3},4:{5:4},5:{}}
15 s,t=0,5
16 print(dag_sp(W,s,t)) #7
```

用图来表示计算过程就是下面所示：



2.”从哪里来？”：我们逆向思维，目标是要到f，那从a点经过哪个点到f点会近些呢？只能是求解从a点出发能够到达的那些点哪个距离f点更近，这里a点能够到达b点和f点，f点到f点距离是0，但是a到f点的距离是9，可能不是最近的路，所以还要看b点到f点有多近，看b点到f点有多近就是求解从b点出发能够到达的那些点哪个距离f点更近，所以又绕回来了，也就是递归下去，直到我们能够回答从a点经过哪个点到f点会更近。这种思维方式的代码实现就是递归版本。

这种情况下，不需要输入是经过了拓扑排序的，所以你可以任意修改输入w中节点的顺序，结果都是一样的，而上面采用迭代实现方式必须要是拓扑排序了的，从中你就可以看出迭代版本和递归版本的区别了。

Python

```

from functools import wraps
1 from functools import wraps
2 def memo(func):
3     cache={}
4     @wraps(func)
5     def wrap(*args):
6         if args not in cache:
7             cache[args]=func(*args)
8             # print('cache {0} = {1}'.format(args[0],cache[args]))

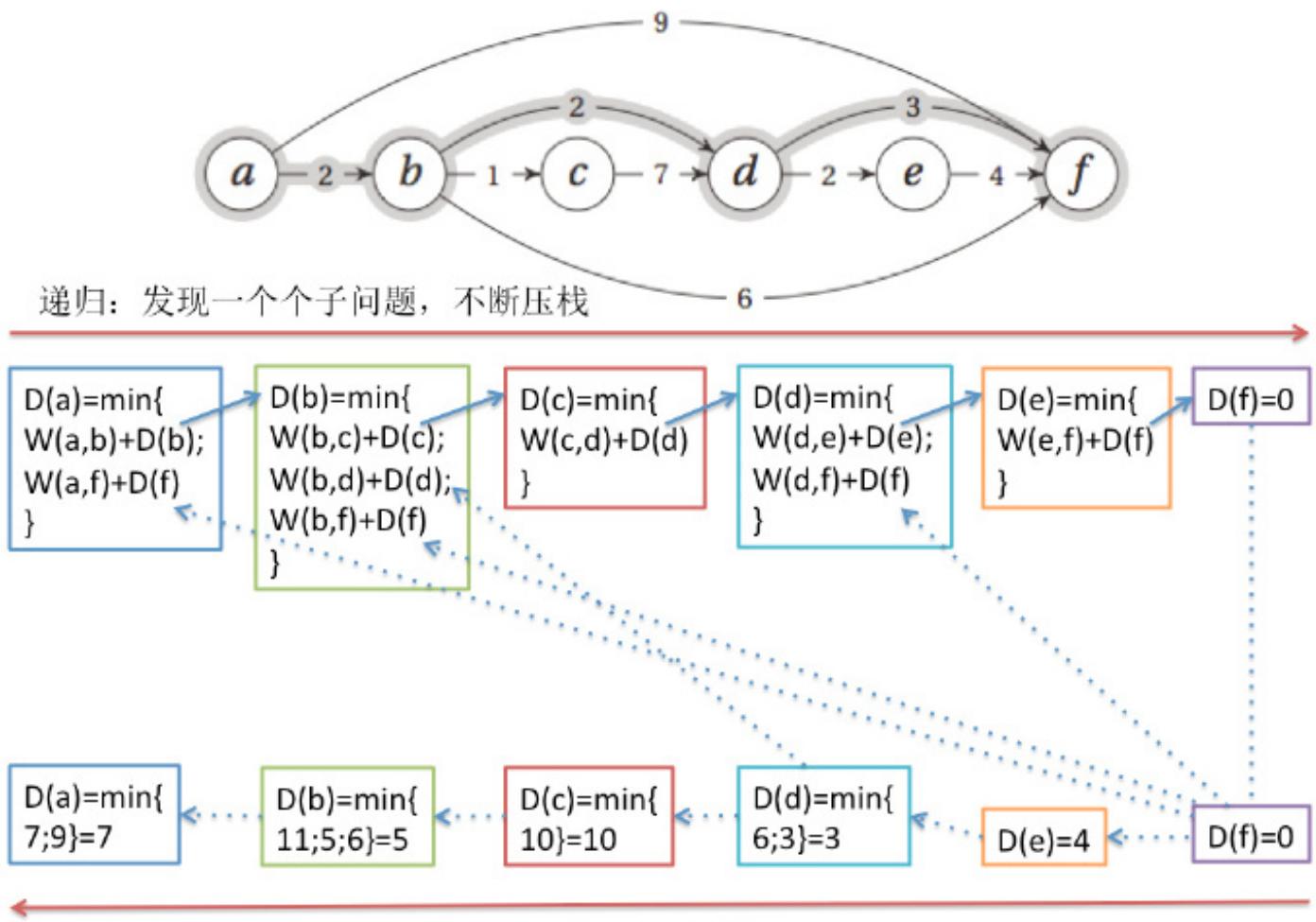
```

```

9     return cache[args]
10    return wrap
11
12 def rec_dag_sp(W, s, t):
13     @memo
14     def d(u):
15         if u == t: return 0
16         return min(W[u][v]+d(v) for v in W[u])
17     return d(s)
18
19 #邻接表
20 W={0:{1:2,5:9},1:{2:1,3:2,5:6},2:{3:7},3:{4:2,5:3},4:{5:4},5:{}}
21 s,t=0,5
22 print(rec_dag_sp(W,s,t)) #7

```

用图来表示计算过程就如下图所示：



递归：子问题一个个被求解出来，虚线表示子问题的解被再次引用

[扩展内容：对DAG求单源最短路径的动态规划问题的总结，比较难理解，附上原文]

Although the basic algorithm is the same, there are many ways of finding the shortest path in a DAG, and, by extension, solving most DP problems. You could do it recursively, with memoization, or you could do it iteratively, with relaxation. For the recursion, you could start at the first node, try various “next steps,” and then recurse on the remainder, or (if your graph representation permits) you could look at the last node and try “previous steps” and recurse on the initial part. The former is

usually much more natural, while the latter corresponds more closely to what happens in the iterative version.

Now, if you use the iterative version, you also have two choices: you can relax the edges out of each node (in topologically sorted order), or you can relax all edges into each node. The latter more obviously yields a correct result but requires access to nodes by following edges backward. This isn't as far-fetched as it seems when you're working with an implicit DAG in some nongraph problem. (For example, in the longest increasing subsequence problem, discussed later in this chapter, looking at all backward "edges" can be a useful perspective.)

Outward relaxation, called reaching, is exactly equivalent when you relax all edges. As explained, once you get to a node, all its in-edges will have been relaxed anyway. However, with reaching, you can do something that's hard in the recursive version (or relaxing in-edges): pruning. If, for example, you're only interested in finding all nodes that are within a distance r , you can skip any node that has distance estimate greater than r . You will still need to visit every node, but you can potentially ignore lots of edges during the relaxation. This won't affect the asymptotic running time, though (Exercise 8-6).

Note that finding the shortest paths in a DAG is surprisingly similar to, for example, finding the longest path, or even counting the number of paths between two nodes in a DAG. The latter problem is exactly what we did with Pascal's triangle earlier; the exact same approach would work for an arbitrary graph. These things aren't quite as easy for general graphs, though. Finding shortest paths in a general graph is a bit harder (in fact, Chapter 9 is devoted to this topic), while finding the longest path is an unsolved problem (see Chapter 11 for more on this).

好，我们差不多搞清楚了动态规划的本质以及两种实现方式的优缺点，下面我们就来实践一下，举最常用的例子：[矩阵链乘问题，内容较多，所以请点击链接过去阅读完了之后回来看总结！](#)

OK，希望我把动态规划讲清楚了，总结下：动态规划其实就是一个连续决策的过程，每次决策我们可能有多种选择(二项式系数和0-1背包问题中我们只有两个选择，DAG图的单源最短路径中我们的选择要看点的出边或者入边，矩阵链乘问题中就是矩阵链可以分开的位置总数...），我们每次选择最好的那个作为我们的决策。所以，动态规划的时间复杂度其实和这两者有关，也就是子问题的个数以及子问题的选择个数，一般情况下动态规划算法的时间复杂度就是两者的乘积。

动态规划有两种实现方式：一种是带备忘录的递归形式，这种方式直接从原问题出发，遇到子问题就去求解子问题并存储子问题的解，下次遇到的时候直接取出来，问题求解的过程看起来就像是先自顶向下地展开问题，然后自下而上的进行决策；另一个实现方式是迭代方式，这种方式需要考虑如何给定一个子问题的求解方式，使得后面求解规模较大的问题是需要求解的子问题都已经求解好了，它的缺点就是可能有些子问题不要算但是它还是算了，而递归实现方式只会计算它需要求解的子问题。

练习1：来试试写写最长公共子序列吧，[这篇文章中给出了Python版本的5种实现方式哟！](#)

练习2：算法导论问题 15-4: Planning a company party 计划一个公司聚会

Start example Professor Stewart is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; that is, the supervisor relation forms a

tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

Professor Stewart is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation described in Section 10.4. Each node of the tree holds, in addition to the pointers, the name of an employee and that employee's conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

原问题可以转换成：假设有一棵树，用左孩子右兄弟的表示方式表示，树的每个结点有个值，选了某个结点，就不能选择它的父结点，求整棵树选的节点值最大是多少。

假设如下：

$dp[i][0]$ 表示不选*i*结点时，*i*子树的最大价值

$dp[i][1]$ 表示选*i*结点时，*i*子树的最大价值

列出状态方程

$dp[i][0] = \max(\max(dp[u][0], dp[u][1]))$ (如果不选*i*结点，*u*为结点*i*的儿子)

$dp[i][1] = \sum(dp[u][0]) + val[i]$ (如果选*i*结点， $val[i]$ 表示*i*结点的价值)

最后就是求 $\max(dp[root][0], dp[root][1])$

1 赞 1 收藏 [评论](#)

Python算法：贪心策略

原文出处：[hujiawei \(@五道口宅男\)](#)

本节主要通过几个例子来介绍贪心策略，主要包括背包问题、哈夫曼编码和最小生成树

贪心算法顾名思义就是每次都贪心地选择当前最好的那个(局部最优解)，不去考虑以后的情况，而且选择了就不能够“反悔”了，如果原问题满足贪心选择性质和最优子结构，那么最后得到的解就是最优解。贪心算法和其他的算法比较有明显的区别，动态规划每次都是综合所有子问题的解得到当前的最优解(全局最优解)，而不是贪心地选择；回溯法是尝试选择一条路，如果选择错了的话可以“反悔”，也就是回过头来重新选择其他的试试。

这个算法想必大家也都很熟悉了，我觉得贪心法总是比较容易想到，但是很难证明它是正确的，所有对于一类问题，条件稍有不同也许就不能使用贪心策略了。这一节采用类似上节的形式，记录下原书中的一些重点难点内容

[果然贪心我领悟的不够，很多问题我貌似都讲不到点子上，大家将就着看下]

1. 匹配问题 matching problem (maximum-weight matching problem)

问题是这样的，有一群人打算一起跳探戈，跳之前要进行分组，一个男人和一个女人成为一组，而且任意一个异性组合都会一个相应的匹配值(compatibility)，目标是求使得匹配值之和达到最大的分组方式。

To be on the safe side, just let me emphasize that this greedy solution would not work in general, with an arbitrary set of weights. The distinct powers of two are key here.

一般情况下，如果匹配值是任意值的话，这个问题使用贪心法是不行的！但是如果匹配值都是2的整数幂的话，那么贪心法就能解决这个问题了！**[这点我不明白，这是此题的一个重点，避免误导，我附上原文，不解释了，如果读者有明白了的希望能留言告知，嘿嘿]**

In this case (or the bipartite case, for that matter), greed won't work in general. However, by some freak coincidence, all the compatibility numbers happen to be distinct powers of two. Now, what happens?

Let's first consider what a greedy algorithm would look like here and then see why it yields an optimal result. We'll be building a solution piece by piece—let the pieces be pairs and a partial solution be a set of pairs. Such a partial solution is valid only if no person in it participates in two (or more) of its pairs. The algorithm will then be roughly as follows:

1. List potential pairs, sorted by decreasing compatibility.
2. Pick the first unused pair from the list.
3. Is anyone in the pair already occupied? If so, discard it; otherwise, use it.
4. Are there any more pairs on the list? If so, go to 2.

As you'll see later, this is rather similar to Kruskal's algorithm for minimum spanning trees (although that works regardless of the edge weights). It also is a rather prototypical greedy algorithm. Its

correctness is another matter. Using distinct powers of two is sort of cheating, because it would make virtually any greedy algorithm work; that is, you'd get an optimal result as long as you could get a valid solution at all. Even though it's cheating (see Exercise 7-3), it illustrates the central idea here: making the greedy choice is safe. Using the most compatible of the remaining couples will always be at least as good as any other choice.

贪心解决的思路大致如下：首先列举出所有可能的组合，然后将它们按照匹配值进行降序排序，接着按顺序从中选择前面没有使用过而且人物没有在前面出现过的组合，遍历完整个序列就得到了匹配值之和最大的分组方式。

[原书关于稳定婚姻的扩展知识 EAGER SUITORS AND STABLE MARRIAGES]

There is, in fact, one classical matching problem that can be solved (sort of) greedily: the stable marriage problem. The idea is that each person in a group has preferences about whom he or she would like to marry. We'd like to see everyone married, and we'd like the marriages to be stable, meaning that there is no man who prefers a woman outside his marriage who also prefers him. (To keep things simple, we disregard same-sex marriages and polygamy here.)

There's a simple algorithm for solving this problem, designed by David Gale and Lloyd Shapley. The formulation is quite gender-conservative but will certainly also work if the gender roles are reversed. The algorithm runs for a number of rounds, until there are no unengaged men left. Each round consists of two steps:

1. Each unengaged man proposes to his favorite of the women he has not yet asked.
2. Each woman is (provisionally) engaged to her favorite suitor and rejects the rest.

This can be viewed as greedy in that we consider only the available favorites (both of the men and women) right now. You might object that it's only sort of greedy in that we don't lock in and go straight for marriage; the women are allowed to break their engagement if a more interesting suitor comes along. Even so, once a man has been rejected, he has been rejected for good, which means that we're guaranteed progress.

To show that this is an optimal and correct algorithm, we need to know that everyone gets married and that the marriages are stable. Once a woman is engaged, she stays engaged (although she may replace her fiance'). There is no way we can get stuck with an unmarried pair, because at some point the man would have proposed to the woman, and she would have (provisionally) accepted his proposal.

How do we know the marriages are stable? Let's say Scarlett and Stuart are both married but not to each other. Is it possible they secretly prefer each other to their current spouses? No: if so, Stuart would already have proposed to her. If she accepted that proposal, she must later have found someone she liked better; if she rejected it, she would already have a preferable mate.

Although this problem may seem silly and trivial, it is not. For example, it is used for admission to some colleges and to allocate medical students to hospital jobs. There have, in fact, been written entire books (such as those by Donald Knuth and by Dan Gusfield and Robert W. Irwing) devoted to

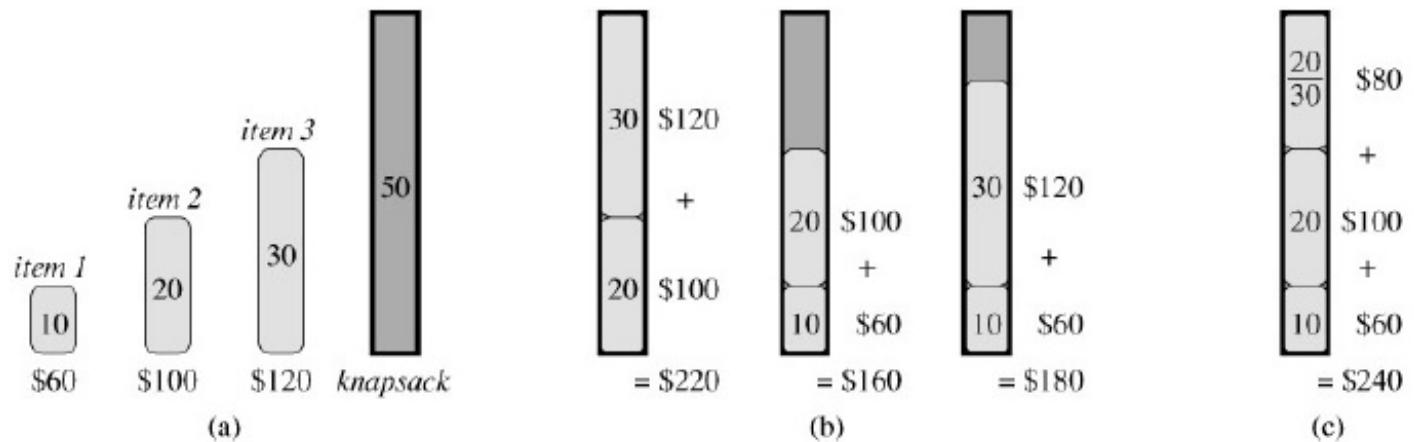
the problem and its variations.

2. 背包问题

这个问题大家很熟悉了，而且该问题的变种很多，常见的有整数背包和部分背包问题。问题大致是这样的，假设现在我们要装一些物品到一个书包里，每样物品都有一定的重量 w 和价值 v ，但是呢，这个书包承重量有限，所以我们要进行决策，如何选择物品才能使得最终的价值最大呢？整数背包是说一个物品要么拿要么不拿，比如茶杯或者台灯等等，而部分背包问题是说一个物品你可以拿其中的一部分，比如一袋子苹果放不下可以只装半袋子苹果。[更加复杂的版本是说每个物品都有一定的体积，同时书包还有体积的限制等等]

很显然，部分背包问题是可以用贪心法来求解的，我们计算每个物品的单位重量的价值，然后将它们降序排序，接着开始拿物品，只要装得下全部的该类物品那么就全装进去，如果不能全部装下就装部分进去直到书包载重量满了为止，这种策略肯定是正确的。

但是，整数背包问题就不能用贪心策略了。整数背包问题还可以分成两种：一种是每类物品数量都是有限的(bounded)，比如只有3个茶杯和2个台灯；还有一种是数量无限的(unbounded)，也就是你要多少有多少，这两种都不能使用贪心策略。0-1背包问题是典型的第一种整数背包问题，看下算法导论上的这个例子就明白了，在(b)中，虽然物品1单位重量的价值最大，但是任何包含物品1的选择都没有超过选择物品2和物品3得到的最优解220；而(c)中能达到最大的价值是240。



整数背包问题还没有能够在多项式时间内解决它的算法，下一节我们介绍的动态规划能够解决0-1背包问题，但是是一个伪多项式时间复杂度。[实际时间复杂度是 $O(nw)$ ， n 是物品数目， w 是书包载重量，严格意义上说这不是一个多项式时间复杂度]

There are two important cases of the integer knapsack problem—the bounded and unbounded cases. The bounded case assumes we have a fixed number of objects in each category,⁴ and the unbounded case lets us use as many as we want. Sadly, greed won't work in either case. In fact, these are both unsolved problems, in the sense that no polynomial algorithms are known to solve them. There is hope, however. As you'll see in the next chapter, we can use dynamic programming to solve the problems in pseudopolynomial time, which may be good enough in many important cases. Also, for the unbounded case, it turns out that the greedy approach ain't half bad! Or, rather, it's at least half good, meaning that we'll never get less than half the optimum value. And with a slight modification, you can get as good results for the bounded version, too. This concept of greedy

approximation is discussed in more detail in Chapter 11.

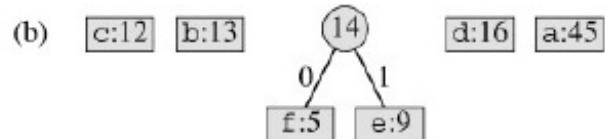
3. 哈夫曼编码

这个问题原始是用来实现一个可变长度的编码问题，但可以总结成这样一个问题，假设我们有很多的叶子节点，每个节点都有一个权值w(可以是任何有意义的数值，比如它出现的概率)，我们要用这些叶子节点构造一棵树，那么每个叶子节点就有一个深度d，我们的目标是使得所有叶子节点的权值与深度的乘积之和 $\sum w_i d_i$ 最小。

很自然的一个想法就是，对于权值大的叶子节点我们让它的深度小些(更加靠近根节点)，权值小的让它的深度相对大些，这样的话我们自然就会想着每次取当前权值最小的两个节点将它们组合出一个父节点，一直这样组合下去直到只有一个节点即根节点为止。如下图所示的示例

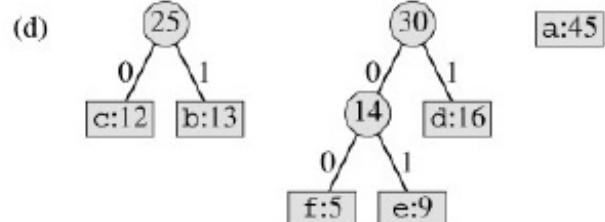
(a)

f:5	e:9	c:12	b:13	d:16	a:45
-----	-----	------	------	------	------



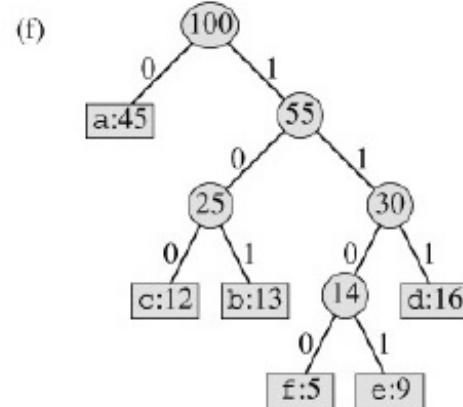
(c)

d:16	a:45
------	------



(e)

a:45



代码实现比较简单，使用了heapq模块，树结构是用list来保存的，有意思的是其中zip函数的使用，其中统计函数count作为zip函数的参数，[详情见python docs](#)

Python

```
from heapq import heapify, heappush, heappop
from itertools import count
def huffman(seq, frq):
    num = count()
```

```

6   trees = list(zip(frq, num, seq))      # num ensures valid ordering
7   heapify(trees)                      # A min-heap based on freq
8   while len(trees) > 1:                # Until all are combined
9       fa, _, a = heappop(trees)        # Get the two smallest trees
10      fb, _, b = heappop(trees)
11      n = next(num)
12      heappush(trees, (fa+fb, n, [a, b])) # Combine and re-add them
13  # print trees
14  return trees[0][-1]
15
16 seq = "abcdefghijklm"
17 frq = [4, 5, 6, 9, 11, 12, 15, 16, 20]
18 print(huffman(seq, frq))
19 # [[['i'], [['a', 'b'], 'e']], [['f', 'g'], [['c', 'd'], 'h']]]

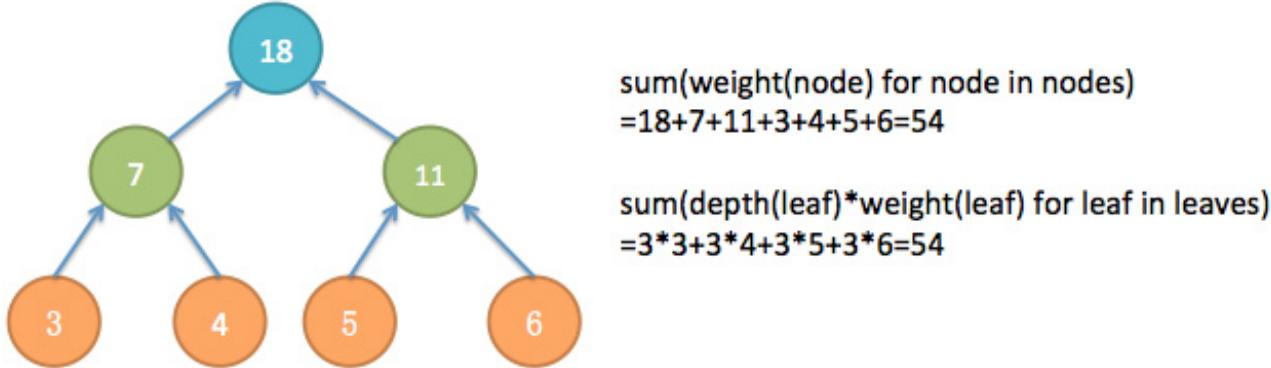
```

现在我们考虑另外一个问题，合并文件问题，假设我们将大小为 m 和大小为 n 的两个文件合并在一起需要 $m+n$ 的时间，现在给定一些文件，求一个最优的合并策略使得所需要的时间最小。

如果我们将上面哈夫曼树中的叶子节点看成是文件，两个文件合并得到的大文件就是树中的内部节点，假设每个节点上都有一个值表示该文件的大小，合并得到的大文件上的值是合并的两个文件的值之和，那我们的目标是就是使得内部节点的和最小的合并方案，因为叶子节点的大小是固定的，所以实际上也就是使得所有节点的和最小的合并方案！

consider how each leaf contributes to the sum over all nodes: the leaf weight occurs as a summand once in each of its ancestor nodes—which means that the sum is exactly the same! That is, $\text{sum}(\text{weight}(\text{node}) \text{ for } \text{node} \in \text{nodes})$ is exactly the same as $\text{sum}(\text{depth}(\text{leaf}) * \text{weight}(\text{leaf}) \text{ for } \text{leaf} \in \text{leaves})$.

细想也就有了一个叶子节点的所有祖先节点们都有一份该叶子节点的值包含在里面，也就是说所有叶子节点的深度与它的值的乘积之和就是所有节点的值之和！可以看下下面的示例图，最终我们知道哈夫曼树就是这个问题的解决方案。



[哈夫曼树问题的一个扩展就是最优二叉搜索树问题，后者可以用动态规划算法来求解，感兴趣的话可以阅读算法导论中动态规划部分内容]

4. 最小生成树

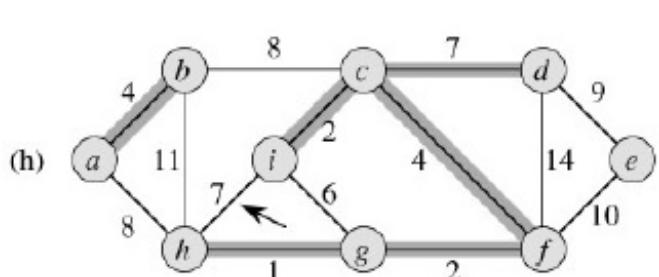
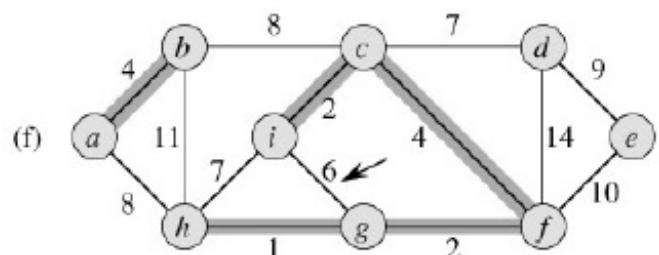
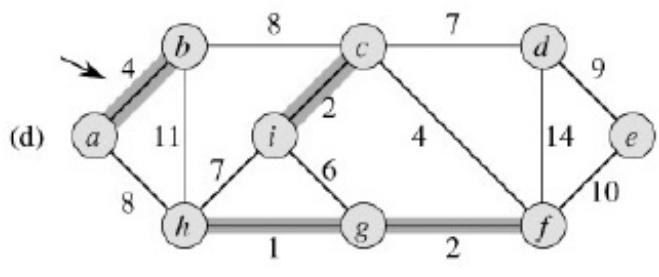
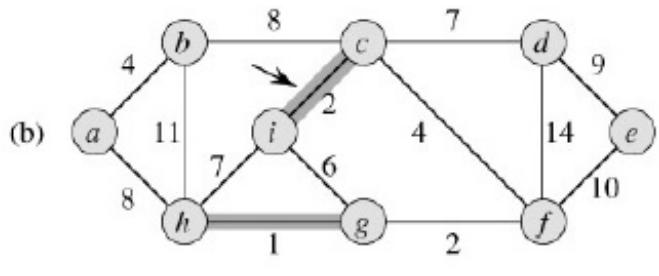
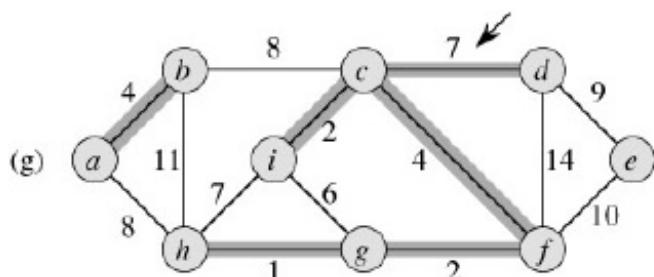
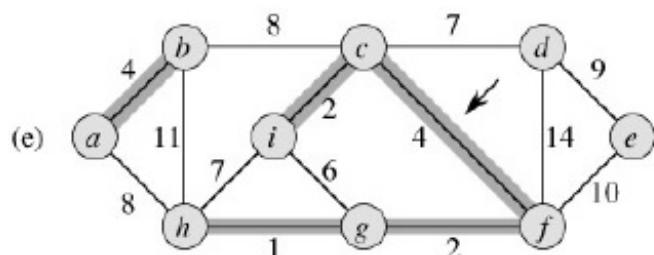
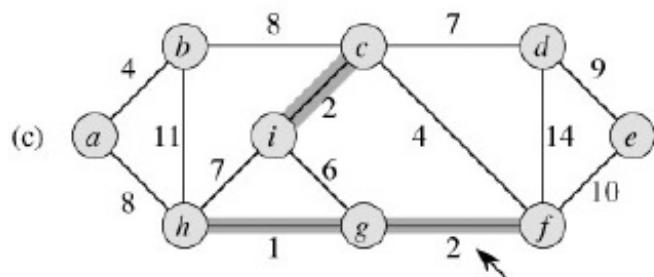
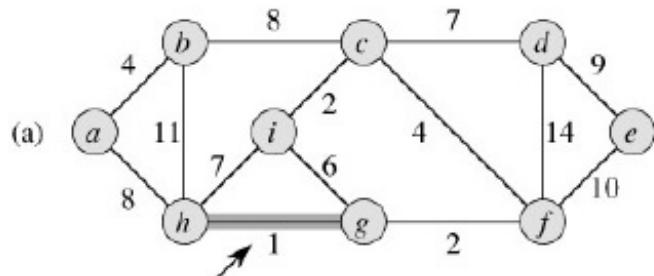
最小生成树是图中的重要算法，主要有两个大家耳熟能详的Kruskal和Prim算法，两个算法都是基于

贪心策略，不过略有不同。

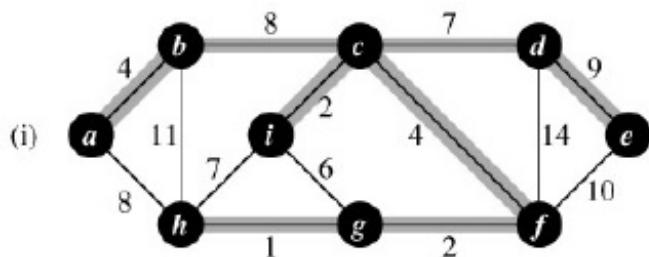
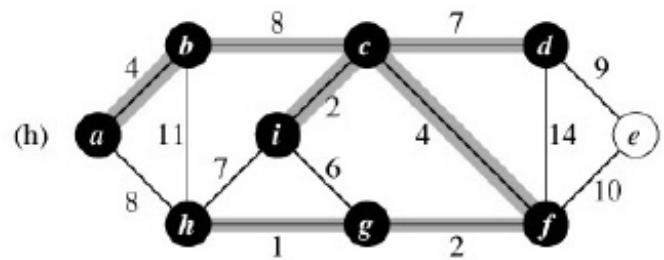
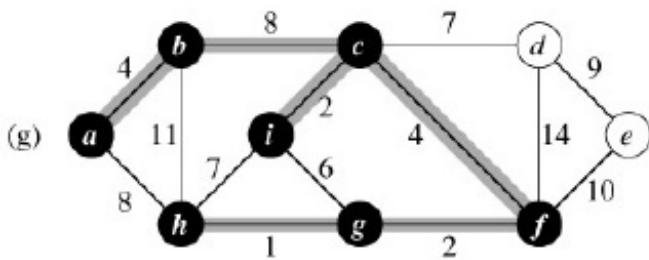
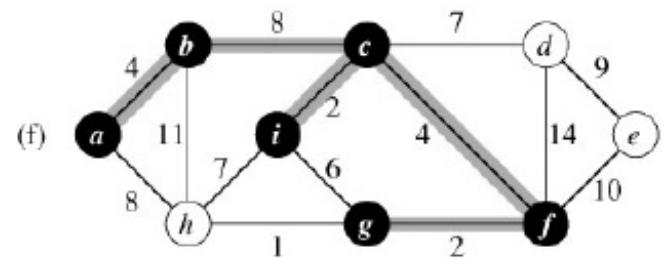
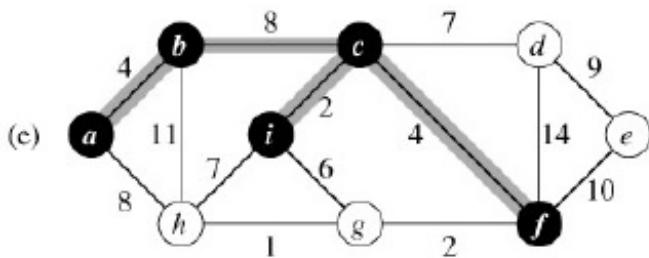
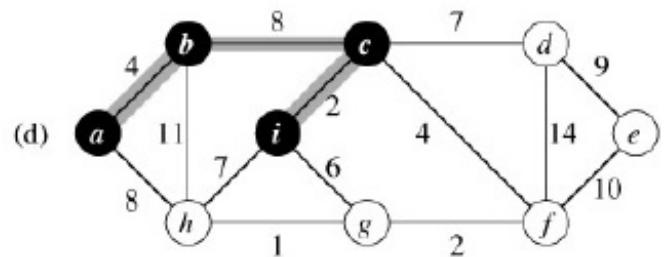
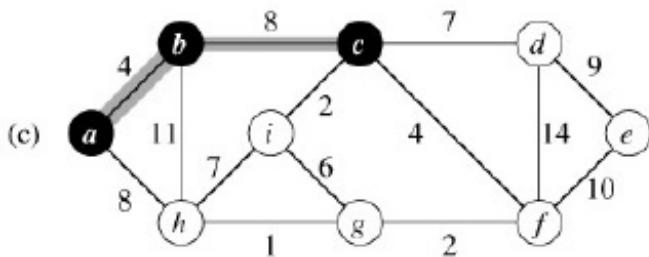
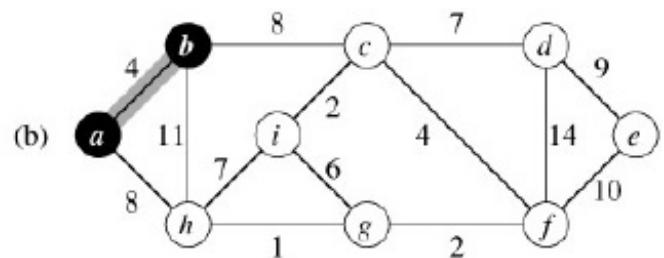
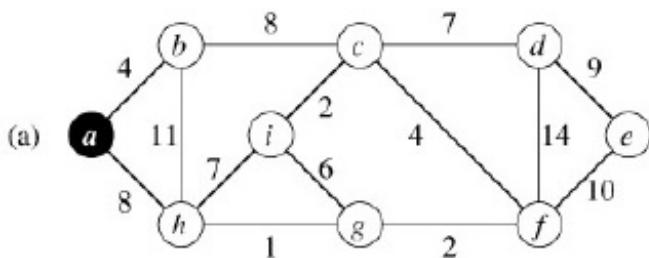
[如果对最小生成树问题的历史感兴趣的话作者推荐看这篇论文“On the History of the Minimum Spanning Tree Problem,” by Graham and Hell]

不了解Kruskal或者Prim算法的童鞋可以参考算法导论的示例图理解下面的内容

Kruskal算法



Prim算法



连通无向图G的生成树是指包含它所有顶点但是部分边的子图，假设每条边都有一个权值，那么权值之和最小的生成树就是最小生成树，它不一定是唯一的。如果图G是非连通的，那么它就没有生成树。

前面我们在介绍遍历的时候也得到过生成树，那里我们是一个顶点一个顶点进行遍历，下面我们通过每次添加一条边来得到最小生成树，而且每次我们贪心地选择剩下的边中权值最小的那条边，但是要保证不能形成环！

那怎么判断是否会出现环呢？

假设我们要考虑是否添加边(u, v)，一个最直接的想法就是遍历已生成的树，看是否能够从 u 到 v ，如果能，那么就舍弃这条边继续考虑后面的边，否则就添加这条边。很显然，采用遍历的方式太费时了。

再假设我们用一个集合来保存我们已经生成的树中的节点，如果我们要考虑是否添加边(u, v)，那么我们就看下集合中这两个节点是否都存在，如果都存在的话说明这条边加进来的话会形成环。这么做可以在常数时间内确定是否会形成环，但是...它是错误的！除非我们每次添加一条边之后得到的局部解一直都只有一棵树才对，如果之前加入的节点 u 和节点 v 在不同的分支上的话，上面的判断不能确定添加这条边之后会形成环！[后面的Prim算法采用的策略就能保证局部解一直都是同一棵树]

下面我们可以试着让每个加入的节点都知道自己处在哪个分支上，而且我们可以用分支中的某一个节点作为该分支的“代表”，该分支中的所有节点都指向这个“代表”，显然我们接下来会遇到分支合并的问题。如果两个分支因为某条边的加入而连通了，那么它们就要合并了，那怎么合并呢？我们让两个分支中的所有节点都指向同一个“代表”就行了，但是这是一个线性时间的操作，我们可以做得更快！假设我们改变下策略，让每个节点指向另一个节点(这个节点不一定是分支的“代表”)，如果我们顺着指向链一直找，就肯定能找到“代表”，因为“代表”是自己指向自己的。这样的话，如果两个分支要合并，只需要让其中的一个分支的“代表”指向另一个分支的“代表”就行啦！这就是一个常数时间的操作。

基于上面的思路我们就有了下面的实现

Python

```
#A Naive Implementation of Kruskal's Algorithm
1 def naive_find(C, u):                      # Find component rep.
2     while C[u] != u:                         # Rep. would point to itself
3         u = C[u]
4     return u
5
6
7 def naive_union(C, u, v):                   # Make one refer to the other
8     u = naive_find(C, u)                     # Find both reps
9     v = naive_find(C, v)
10    C[u] = v
11
12 def naive_kruskal(G):
13     E = [(G[u][v], u, v) for u in G for v in G[u]]
14     T = set()                                # Empty partial solution
15     C = {u: u for u in G}                    # Component reps
16     for _, u, v in sorted(E):                # Edges, sorted by weight
17         if naive_find(C, u) != naive_find(C, v):
18             T.add((u, v))                      # Different reps? Use it!
19             naive_union(C, u, v)              # Combine components
20     return T
21
22 G = {
23     0: {1:1, 2:3, 3:4},
24     1: {2:5},
25     2: {3:2},
26     3: set()
27 }
28 print list(naive_kruskal(G)) #[0, 1), (2, 3), (0, 2]
```

从上面的分析我们可以看到，虽然合并时修改指向的操作是常数时间的，但是通过指向链的方式找

到“代表”所花的时间是线性的，而这里还可以做些改进。

首先，在合并(union)的时候我们让“小”分支指向“大”分支，这样平衡了之后平均查找时间肯定有所下降，那么怎么确定分支的“大小”呢？这个可以用平衡树的方式来思考，假设我们给每个节点都设置一个权重(rank or weight)，其实重要的还是“代表”的权重，如果要合并的两个分支的“代表”的权重相等的话，在将“小”分支指向“大”分支之后，还要将“大”分支的权重加1。

其次，在查找(find)的时候我们一边查找一边修正经过的点的指向，让它直接指向“代表”，这个怎么做呢？使用递归就行了，因为递归在找到了之后会回溯，回溯的时候就可以设置其他节点的“代表”了，这个叫做path compression技术，是Kruskal算法常用的一个技巧。

基于上面的改进就有了下面优化的Kruskal算法

Python

```
#Kruskal's Algorithm
def find(C, u):
    1 #Kruskal's Algorithm
    2 def find(C, u):
    3     if C[u] != u:
    4         C[u] = find(C, C[u])           # Path compression
    5     return C[u]
    6
    7 def union(C, R, u, v):
    8     u, v = find(C, u), find(C, v)
    9     if R[u] > R[v]:                 # Union by rank
   10        C[v] = u
   11    else:
   12        C[u] = v
   13    if R[u] == R[v]:               # A tie: Move v up a level
   14        R[v] += 1
   15
   16 def kruskal(G):
   17     E = [(G[u][v], u, v) for u in G for v in G[u]]
   18     T = set()
   19     C, R = {u: u for u in G}, {u: 0 for u in G} # Comp. reps and ranks
   20     for _, u, v in sorted(E):
   21         if find(C, u) != find(C, v):
   22             T.add((u, v))
   23             union(C, R, u, v)
   24     return T
   25
   26 G = {
   27     0: {1:1, 2:3, 3:4},
   28     1: {2:5},
   29     2: {3:2},
   30     3: set()
   31 }
   32 print list(kruskal(G)) #[[0, 1), (2, 3), (0, 2)]
```

接下来就是Prim算法了，它其实就是我们前面介绍的traversal算法中的一种，不同点是它对待办事项(to-do list，即前面提到的“边缘节点”，也就是我们已经包含的这些节点能够直接到达的那些节点)进行了一定的排序，我们在实现BFS时使用的是双端队列deque，此时我们只要把它改成一个优先队列(priority queue)就行了，这里选用heapq模块中的堆heap。

Prim算法不断地添加新的边(也可以说是一个新的顶点)，一旦我们加入了一条新的边，可能会导致某

些原来的边缘节点到生成树的距离更加近了，所以我们要更新一下它们的距离值，然后重新调整下排序，那怎么修改距离值呢？我们可以先找到原来的那个节点，然后再修改它的距离值接着重新调整堆，但是这么做实在是太麻烦了！这里有一个巧妙的技巧就是直接向堆中插入新的距离值的节点！为什么可以呢？因为插入的新节点B的距离值比原来的节点A的距离值小，那么Prim算法添加顶点的时候肯定是先弹出堆中的节点B，后面如果弹出节点A的话，因为这个节点已经添加进入了，直接忽略就行了，也就是说我们这么做不仅很简单，而且并没有把原来的问题搞砸了。下面是作者给出的详细解释，总共三点，第三点是重复的添加不会影响算法的渐近时间复杂度

- We're using a priority queue, so if a node has been added multiple times, by the time we remove one of its entries, it will be the one with the lowest weight (at that time), which is the one we want.
- We make sure we don't add the same node to our traversal tree more than once. This can be ensured by a constant-time membership check. Therefore, all but one of the queue entries for any given node will be discarded.
- The multiple additions won't affect asymptotic running time

[重新添加一次权值减小了的节点就相当于是松弛(或者说是隐含了松弛操作在里面)，Re-adding a node with a lower weight is equivalent to a relaxation，这两种方式是可以相互交换的，后面图算法中作者在实现Dijkstra算法时使用的是relax，那其实我们还可以实现带relax的Prim和不带relax的Dijkstra]

根据上面的分析就有了下面的Prim算法实现

Python

```
from heapq import heappop, heappush
1 from heapq import heappop, heappush
2
3 def prim(G, s):
4     P, Q = {}, [(0, None, s)]
5     while Q:
6         _, p, u = heappop(Q)
7         if u in P: continue
8         P[u] = p
9         for v, w in G[u].items():
10            heappush(Q, (w, u, v)) #weight, predecessor node, node
11    return P
12
13 G = {
14     0: {1:1, 2:3, 3:4},
15     1: {0:1, 2:5},
16     2: {0:3, 1:5, 3:2},
17     3: {2:2, 0:4}
18 }
19 print prim(G, 0) # {0: None, 1: 0, 2: 0, 3: 2}
```

[扩展知识，另一个角度来看最小生成树 A SLIGHTLY DIFFERENT PERSPECTIVE]

In their historical overview of minimum spanning tree algorithms, Ronald L. Graham and Pavol Hell outline three algorithms that they consider especially important and that have played a central role

in the history of the problem. The first two are the algorithms that are commonly attributed to Kruskal and Prim (although the second one was originally formulated by Vojteˇch Jarník in 1930), while the third is the one initially described by Boruˇvka. Graham and Hell succinctly explain the algorithms as follows. A partial solution is a spanning forest, consisting of a set of fragments (components, trees). Initially, each node is a fragment. In each iteration, edges are added, joining fragments, until we have a spanning tree.

Algorithm 1: Add a shortest edge that joins two different fragments.

Algorithm 2: Add a shortest edge that joins the fragment containing the root to another fragment.

Algorithm 3: For every fragment, add the shortest edge that joins it to another fragment.

For algorithm 2, the root is chosen arbitrarily at the beginning. For algorithm 3, it is assumed that all edge weights are different to ensure that no cycles can occur. As you can see, all three algorithms are based on the same fundamental fact—that the shortest edge over a cut is safe. Also, in order to implement them efficiently, you need to be able to find shortest edges, detect whether two nodes belong to the same fragment, and so forth (as explained for algorithms 1 and 2 in the main text). Still, these brief explanations can be useful as a memory aid or to get the bird’s-eye perspective on what’s going on.

5.Greed Works. But When?

还是老话题，贪心算法真的很好，有时候也比较容易想到，但是它什么时候是正确的呢？

针对这个问题，作者提出了些建议和方法[都比较难翻译和理解，感兴趣还是阅读原文较好]

(1)Keeping Up with the Best

This is what Kleinberg and Tardos (in Algorithm Design) call staying ahead. The idea is to show that as you build your solution, one step at a time, the greedy algorithm will always have gotten at least as far as a hypothetical optimal algorithm would have. Once you reach the finish line, you’ve shown that greed is optimal.

(2)No Worse Than Perfect

This is a technique I used in showing the greedy choice property for Huffman’s algorithm. It involves showing that you can transform a hypothetical optimal solution to the greedy one, without reducing the quality. Kleinberg and Tardos call this an exchange argument.

(3)Staying Safe

This is where we started: to make sure a greedy algorithm is correct, we must make sure each greedy step along the way is safe. One way of doing this is the two-part approach of showing (1) the greedy choice property, that is, that a greedy choice is compatible with optimality, and (2) optimal substructure, that is, that the remaining subproblem is a smaller instance that must also be solved optimally.

[扩展知识：算法导论中还介绍了贪心算法的内在原理，也就是拟阵，贪心算法一般都是求这个拟阵的最大独立子集，方法就是从一个空的独立子集开始，从一个已经经过排序的序列中依次取出一个元素，尝试添加到独立子集中，如果新元素加入之后的集合仍然是一个独立子集的话那就加入进去，这样就形成了一个更大的独立子集，待遍历完整个序列时我们就得到最大的独立子集。拟阵的内容比较难，感兴趣不妨阅读下算法导论然后证明一两道练习题挑战下，嘻嘻]

用Python代码来形容上面的过程就是

Python

```
#贪心算法的框架 [拟阵  
的思想]  
#  
1 #贪心算法的框架 [拟阵的思想]  
2 def greedy(E, S, w):  
3     T = []          # Emtpy, partial solution  
4     for e in sorted(E, key=w):      # Greedily consider elements  
5         TT = T + [e]            # Tentative solution  
6         if TT in S: T = TT      # Is it valid? Use it!  
7     return T
```

练习：[试试这道删数问题吧，这里对比了贪心算法和动态规划算法两种解法](#)

1 赞 1 收藏 [评论](#)

Python算法：分治法

原文出处：[hujiawei \(@五道口宅男\)](#)

本节主要介绍分治法策略，提到了树形问题的平衡性以及基于分治策略的排序算法

本节的标题写全了就是：**divide the problem instance, solve subproblems recursively, combine the results, and thereby conquer the problem**

简言之就是将原问题划分成几个小问题，然后递归地解决这些小问题，最后综合它们的解得到问题的解。分治法的思想我想大家都已经很清楚了，所以我就不过多地介绍它了，下面摘录些原书中的重点内容。

1. 平衡性是树形问题的关键

如果我们将子问题看做节点，将问题之间的依赖关系(dependencies or reductions)看做边，那么我们就得到了子问题图(subproblem graph)，最简单的子问题图就是树形结构问题，例如我们之前提到过的递归树的形式。也许子问题之间有依赖关系，但是对于每个子问题我们都是可以独立求解的，根据我们前面学的内容，只要我们能够找到合适的规约，我们就可以直接使用递归形式的算法将这个问题解决。[至于子问题间有重叠的话我们后面会详细介绍动态规划的方法来解决这类问题，这里我们不考虑]

前面我们学的内容已经完全足够我们理解分治法了，第3节的Divide-and-conquer recurrences，第4节的Strong induction，还有第5节的Recursive traversal

The recurrences tell you something about the performance involved, the induction gives you a tool for understanding how the algorithms work, and the recursive traversal (DFS in trees) is a raw skeleton for the algorithms.

但是，我们前面介绍Induction时总是从 $n-1$ 到 n ，这节我们要考虑平衡性，我们希望从 $n/2$ 到 n ，也就是说我们假设我们能够解决规模为原问题一半的子问题。

假设对于同一个问题，我们有下面两个解决方案，哪个方案更好些呢？

$$(1) T(n)=T(n-1)+T(1)+n$$

$$(2) T(n)=2T(n/2)+n$$

如果从时间复杂度来评价的话，前者是 $O(n^2)$ 的，而后者是 $O(n\lg n)$ 的，所以是后者更好些。下图以递归树的形式显示了两种方案的不同

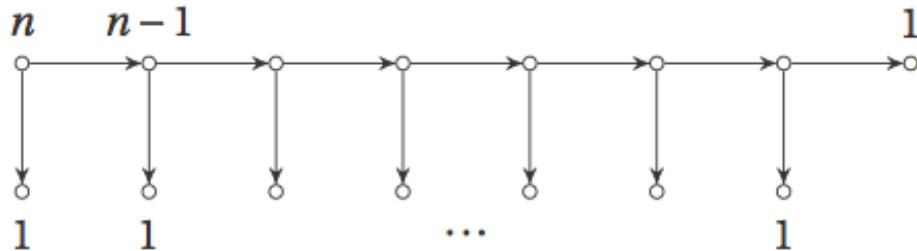


Figure 6-1. An unbalanced decomposition, with linear division/combination cost and quadratic running time in total

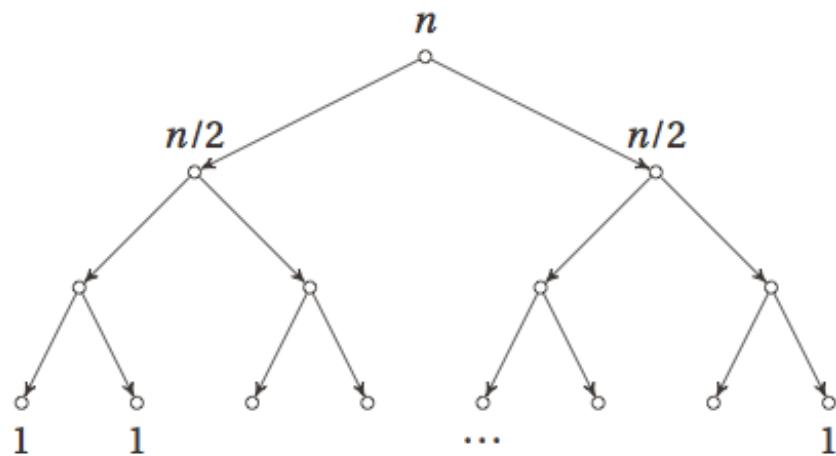


Figure 6-2. Divide and conquer: a balanced decomposition, with linear division/combination cost and loglinear running time in total

2. 典型的分治法

下面是典型分治法的伪代码，很容易理解对吧

Python

```
# Pseudocode(ish)
def divide_and_conquer(S, divide, combine):
    # Pseudocode(ish)
    if len(S) == 1: return S
    L, R = divide(S)
    A = divide_and_conquer(L, divide, combine)
    B = divide_and_conquer(R, divide, combine)
    return combine(A, B)
```

用图形来表示如下，上面部分是分(dvision)，下面部分是合(combination)

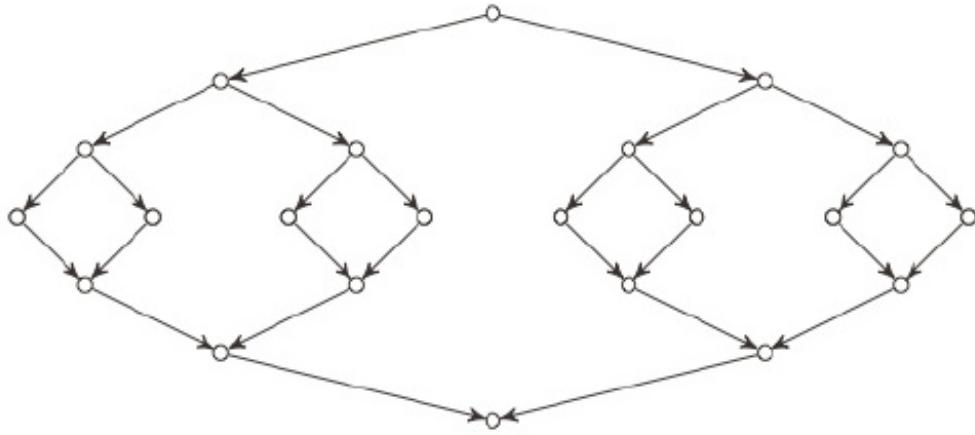


Figure 6-5. Dividing, recursing, and combining in a divide-and-conquer algorithm

二分查找是最常用的采用分治策略的算法，我们经常使用的版本控制系统(evision control systems=RCSS)查找代码中发生某个变化是在哪个版本时采用的正是二分查找策略。

Python中**bisect**模块也正是利用了二分查找策略，其中方法**bisect**的作用是返回要找到元素的位置，**bisect_left**是其左边的那个位置，而**bisect_right**和**bisect**的作用是一样的，函数**insort**也是这样设计的。

Python

```
from bisect import
bisect
1 from bisect import bisect
2 a = [0, 2, 3, 5, 6, 7, 8, 8, 9]
3 print bisect(a, 5) #4
4 from bisect import bisect_left, bisect_right
5 print bisect_left(a, 5) #3
6 print bisect_right(a, 5) #4
```

二分查找策略很好，但是它有个前提，序列必须是有序的才可以这样做，为了高效地得到中间位置的元素，于是就有了二叉搜索树，这个我们在[数据结构篇中已经详细介绍过了](#)，下面给出一份完整的二叉搜索树的实现，不过多介绍了。

Python

```
class Node:
    lft = None
1 class Node:
2     lft = None
3     rgt = None
4     def __init__(self, key, val):
5         self.key = key
6         self.val = val
7
8     def insert(node, key, val):
9         if node is None: return Node(key, val)    # Empty leaf: Add node here
10        if node.key == key: node.val = val       # Found key: Replace val
11        elif key < node.key:                  # Less than the key?
12            node.lft = insert(node.lft, key, val) # Go left
13        else:                                # Otherwise...
```

```

14     node.rgt = insert(node.rgt, key, val) # Go right
15     return node
16
17 def search(node, key):
18     if node is None: raise KeyError      # Empty leaf: It's not here
19     if node.key == key: return node.val # Found key: Return val
20     elif key < node.key:             # Less than the key?
21         return search(node.lft, key)   # Go left
22     else:                          # Otherwise...
23         return search(node.rgt, key) # Go right
24
25 class Tree:                      # Simple wrapper
26     root = None
27     def __setitem__(self, key, val):
28         self.root = insert(self.root, key, val)
29     def __getitem__(self, key):
30         return search(self.root, key)
31     def __contains__(self, key):
32         try: search(self.root, key)
33         except KeyError: return False
34         return True

```

比较：二分法，二叉搜索树，字典

三者都是用来提高搜索效率的，但是各有区别。二分法只能作用于有序数组(例如排序后的Python的list)，但是有序数组较难维护，因为插入需要线性时间；二叉搜索树有些复杂，动态变化着，但是插入和删除效率高了些；字典的效率相比而言就比较好，插入删除操作的平均时间都是常数的，只不过它还需要计算下hash值才能确定元素的位置。

3.顺序统计量

在算法导论中一组序列中的第 k 大的元素定义为顺序统计量

如果我们想要在线性时间内找到一组序列中的前 k 大的元素怎么做呢？很显然，如果这组序列中的数字范围比较大的话，我们就不能使用线性排序算法，而其他的基于比较的排序算法的最好的平均时间复杂度($O(nlgn)$)都超过了线性时间，怎么办呢？

[扩展知识：在Python中如果需要求前 k 小或者前 k 大的元素，可以使用heapq模块中的nsmallest或者nlargest函数，如果 k 很小的话这种方式会好些，但是如果 k 很大的话，不如直接去调用sort函数]

要想解决这个问题，我们还是要用分治法，采用类似快排中的partition将序列进行划分(divide)，也就是说找一个主元(pivot)，然后用主元作为基准将序列分成两部分，一部分小于主元，另一半大于主元，比较下主元最终的位置值和 k 的大小关系，然后确定后面在哪个部分继续进行划分。如果这里不理解的话请移步阅读前面[数据结构篇之排序中的快速排序](#)

基于上面的想法就有了下面的实现，需要注意的是下面的partition函数不是就地划分的哟

Python

#A Straightforward Implementation of Partition and Select	
---	---

```

1 #A Straightforward Implementation of Partition and Select
2 def partition(seq):
3     pi, seq = seq[0], seq[1:]           # Pick and remove the pivot
4     lo = [x for x in seq if x <= pi]    # All the small elements

```

```

5     hi = [x for x in seq if x > pi]      # All the large ones
6     return lo, pi, hi                      # pi is "in the right place"
7
8 def select(seq, k):
9     lo, pi, hi = partition(seq)           # [<= pi], pi, [> pi]
10    m = len(lo)
11    if m == k: return pi                 # We found the kth smallest
12    elif m < k:                         # Too far to the left
13        return select(hi, k-m-1)          # Remember to adjust k
14    else:                               # Too far to the right
15        return select(lo, k)             # Just use original k here
16
17 seq = [3, 4, 1, 6, 3, 7, 9, 13, 93, 0, 100, 1, 2, 2, 3, 3, 2]
18 print partition(seq) #([1, 3, 0, 1, 2, 2, 3, 3, 2], 3, [4, 6, 7, 9, 13, 93, 100])
19 print select([5, 3, 2, 7, 1], 3) #5
20 print select([5, 3, 2, 7, 1], 4) #7
21 ans = [select(seq, k) for k in range(len(seq))]
22 seq.sort()
23 print ans == seq #True

```

细读上面的代码发现主元默认就是第一个元素，你也许会想这么选科学吗？事实证明这种随机选择的期望运行时间的确是线性的，但是如果每次都选择的不好，导致划分的时候每次都特别不平衡将会导致运行时间变成平方时间，那有没有什么选主元的办法能够保证算法的运行时间是线性的？的确有！但是比较麻烦，实际使用的并不多，感兴趣可以看下面的内容

[我还未完全理解，算法导论上也有相应的介绍，感兴趣不妨去阅读下]

It turns out guaranteeing that the pivot is even a small percentage into the sequence (that is, not at either end, or a constant number of steps from it) is enough for the running time to be linear. In 1973, a group of algorists (Blum, Floyd, Pratt, Rivest, and Tarjan) came up with a version of the algorithm that gives exactly this kind of guarantee.

The algorithm is a bit involved, but the core idea is simple enough: first divide the sequence into groups of five (or some other small constant). Find the median in each, using (for example) a simple sorting algorithm. So far, we've used only linear time. Now, find the median among these medians, using the linear selection algorithm recursively. (This will work, because the number of medians is smaller than the size of the original sequence—still a bit mind-bending.) The resulting value is a pivot that is guaranteed to be good enough to avoid the degenerate recursion—use it as a pivot in your selection.

In other words, the algorithm is used recursively in two ways: first, on the sequence of medians, to find a good pivot, and second, on the original sequence, using this pivot.

While the algorithm is important to know about for theoretical reasons (because it means selection can be done in guaranteed linear time), you'll probably never actually use it in practice.

3.二分排序

前面我们介绍了二分查找，下面看看如何进行二分排序，这里不再详细介绍快排和合并排序的思想了，如果不理解的话请移步阅读前面[数据结构篇之排序](#)

利用前面的partition函数快排代码呼之欲出

Python

```
def quicksort(seq):
    if len(seq) <= 1:
        return seq
    1 def quicksort(seq):
    2     if len(seq) <= 1: return seq      # Base case
    3     lo, pi, hi = partition(seq)      # pi is in its place
    4     return quicksort(lo) + [pi] + quicksort(hi) # Sort lo and hi separately
    5
    6 seq = [7, 5, 0, 6, 3, 4, 1, 9, 8, 2]
    7 print(quicksort(seq)) #[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

合并排序是更加典型的采用分治法策略来进行的排序，注意后半部分是比较谁大然后调用append函数，最后reverse一下，因为如果是比较谁小的话就要调用insert函数，它的效率不如append

Python

```
# Mergesort, repeated from Chapter 3 (with some modifications)
from Chapter 3 (with
1 # Mergesort, repeated from Chapter 3 (with some modifications)
2 def mergesort(seq):
3     mid = len(seq)//2                  # Midpoint for division
4     lft, rgt = seq[:mid], seq[mid:]
5     if len(lft) > 1: lft = mergesort(lft)    # Sort by halves
6     if len(rgt) > 1: rgt = mergesort(rgt)
7     res = []
8     while lft and rgt:                 # Neither half is empty
9         if lft[-1] >= rgt[-1]:          # lft has greatest last value
10            res.append(lft.pop())       # Append it
11        else:                         # rgt has greatest last value
12            res.append(rgt.pop())       # Append it
13    res.reverse()                   # Result is backward
14    return (lft or rgt) + res         # Also add the remainder
```

[扩展知识：Python内置的排序算法TimSort，看起来好复杂的样子啊，我果断只是略读了一下下]

BLACK BOX: TIMSORT

The algorithm hiding in `list.sort` is one invented (and implemented) by Tim Peters, one of the big names in the Python community.⁸ The algorithm, aptly named *timsort*, replaces an earlier algorithm that had lots of tweaks to handle special cases such as segments of ascending and descending values, and the like. In *timsort*, these cases are handled by the general mechanism, so the performance is still there (and in some cases, it's much improved), but the algorithm is cleaner and simpler. The algorithm is still a bit too involved to explain in detail here; I'll try to give you a quick overview. For more details, take a look at the source.⁹

Timsort is a close relative to merge sort. It's an *in-place* algorithm, in that it merges segments and leaves the result in the original array (although it uses some auxiliary memory during the merging). Instead of simply sorting the array half-and-half and then merging those, though, it starts at the beginning, looking for segments that are *already sorted* (possibly in reverse), called *runs*. In random arrays, there won't be many, but in many kinds of real data, there may be a lot—giving the algorithm a clear edge over a plain merge sort and a *linear* running time in the best case (and that covers a lot of cases beyond simply getting a sequence that's already sorted).

As *timsort* iterates over the sequence, identifying runs and pushing their bounds onto a stack, it uses some heuristics to decide which runs are to be merged when. The idea is to avoid the kind of merge imbalance that would give you a quadratic running time while still exploiting the structure in the data (that is, the runs). First, any really short runs are artificially extended and sorted (using a stable insertion sort). Second, the following invariants are maintained for the three topmost runs on the stack, A, B, and C (with A on top): $\text{len}(A) > \text{len}(B) + \text{len}(C)$ and $\text{len}(B) > \text{len}(C)$. If the first invariant is violated, the smaller of A and C is merged with B, and the result replaces the merged runs in the stack. The second invariant may still not hold, and the merging continues until both invariants hold.

The algorithm uses some other tricks as well, to get as much speed as possible. If you're interested, I recommend you check out the source.¹⁰ If you'd rather not read C code, you could also take a look at the pure Python version of *timsort*, available as part of the PyPy project.¹¹ Their implementation has excellent comments and is very clearly written. (The PyPy project is discussed in Appendix A.)

[章节最后作者介绍了一些关于树平衡的内容，提到2-3树，我对树平衡不是特别感兴趣，也不是很明白，所以跳过不总结，感兴趣的不妨阅读下]

问题6-2. 三分查找

Binary search divides the sequence into two approximately equal parts in each recursive step. Consider ternary search, which divides the sequence into three parts. What would its asymptotic complexity be? What can you say about the number of comparisons in binary and ternary search?

题目就是说让我们分析下三分查找的时间复杂度，和二分查找进行下对比

The asymptotic running time would be the same. The number of comparison goes up, however. To see this, consider the recurrences $B(n) = B(n/2) + 1$ and $T(n) = T(n/3) + 2$ for binary and ternary search, respectively (with base cases $B(1) = T(1) = 0$ and $B(2) = T(2) = 1$). You can show (by induction) that $B(n) < \lg n + 1 < T(n)$.

1 赞 1 收藏 [评论](#)

Python算法：遍历

原文出处：[hujiawei \(@五道口宅男\)](#)

本节主要介绍图的遍历算法BFS和DFS，以及寻找图的(强)连通分量的算法

Traversal就是遍历，主要是对图的遍历，也就是遍历图中的每个节点。对一个节点的遍历有两个阶段，首先是发现(discover)，然后是访问(visit)。遍历的重要性自然不必说，图中有几个算法和遍历没有关系？！

[算法导论对于发现和访问区别的非常明显，对图的算法讲解地特别好，在遍历节点的时候给节点标注它的发现节点时间 $d[v]$ 和结束访问时间 $f[v]$ ，然后由这些时间的一些规律得到了不少实用的定理，本节后面介绍了部分内容，感兴趣不妨阅读下算法导论原书]

图的连通分量是图的一个最大子图，在这个子图中任何两个节点之间都是相互可达的(忽略边的方向)。我们本节的重点就是想想怎么找到一个图的连通分量呢？

一个很明显的做法是，我们从一个顶点出发，沿着边一直走，慢慢地扩大子图，直到子图不能再扩大了停止，我们就得到了一个连通分量对吧，我们怎么确定我们真的是找到了一个完整的连通分量呢？可以看下作者给出的解释，类似上节的Induction，我们思考从 $i-1$ 到 i 的过程，只要我们保证增加了这个节点后子图仍然是连通的就对了。

Let's look at the following related problem. Show that you can order the nodes in a connected graph, V_1, V_2, \dots, V_n , so that for any $i = 1 \dots n$, the subgraph over V_1, \dots, V_i is connected. If we can show this and we can figure out how to do the ordering, we can go through all the nodes in a connected component and know when they're all used up.

How do we do this? Thinking inductively, we need to get from $i-1$ to i . We know that the subgraph over the $i-1$ first nodes is connected. What next? Well, because there are paths between any pair of nodes, consider a node u in the first $i-1$ nodes and a node v in the remainder. On the path from u to v , consider the last node that is in the component we've built so far, as well as the first node outside it. Let's call them x and y . Clearly there must be an edge between them, so adding y to the nodes of our growing component keeps it connected, and we've shown what we set out to show.

经过上面的一番思考，我们就知道了如何找连通分量：从一个顶点开始，沿着它的边找到其他的节点（或者说站在这个节点上看，看能够发现哪些节点），然后就是不断地向已有的连通分量中添加节点，使得连通分量内部依然满足连通性质。如果我们按照上面的思路一直做下去，我们就得到了一棵树，一棵遍历树，它也是我们遍历的分量的一棵生成树。在具体实现这个算法时，我们要记录“边缘节点”，也就是那些和已得到的连通分量中的节点相连的节点，它们就像是一个个待办事项(to-do list)一样，而前面加入的节点就是标记为已完成的(checked off)待办事项。

这里作者举了一个很有意思的例子，一个角色扮演的游戏，如下图所示，我们可以将房间看作是节点，将房间的门看作是节点之间的边，走过的轨迹就是遍历树。这么看的话，房间就分成了三种：(1)我们已经经过的房间；(2)我们已经经过的房间附近的房间，也就是马上可以进入的房间；(3)“黑屋”，我们甚至都不知道它们是否存在，存在的话也不知道在哪里。

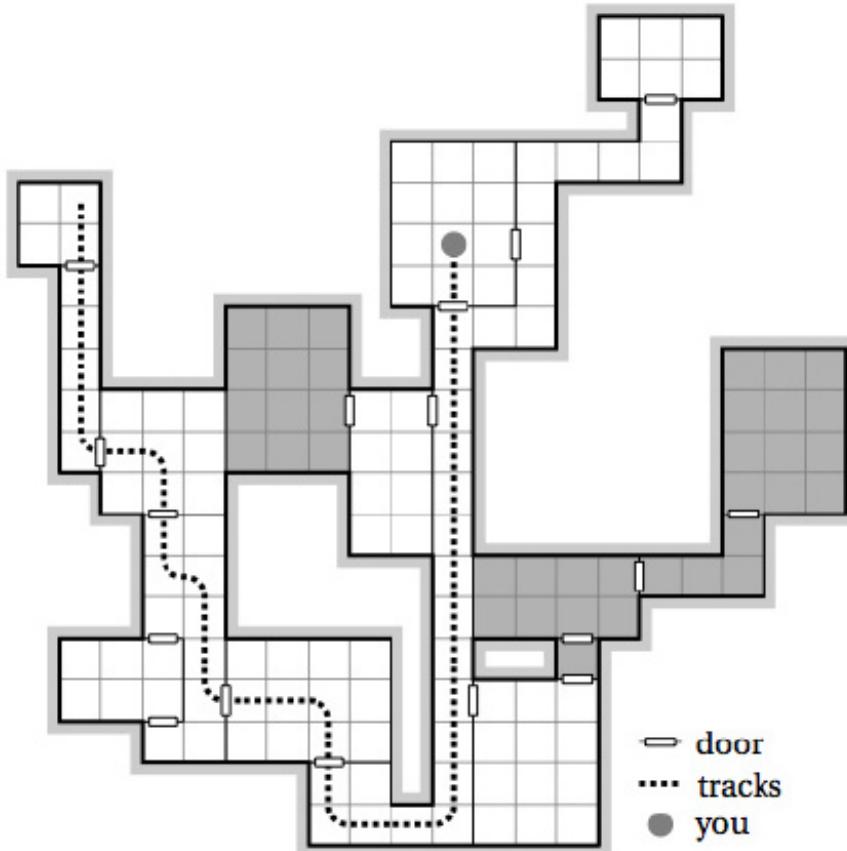


Figure 5-2. A partial traversal of a typical role-playing dungeon. Think of the rooms and nodes and the doors as edges. The traversal tree is defined by your tracks; the fringe (the traversal queue) consists of the neighboring rooms. The remaining (darkened) rooms haven't been discovered yet.

根据上面的分析可以写出下面的遍历函数walk，其中参数S暂时没有用，它在后面求强连通分量时需要，表示的是一个“禁区”(forbidden zone)，也就是不要去访问这些节点。

注意下面的difference函数的使用，参数可以是多个，也就是说调用后返回的集合中的元素在各个参数中都不存在，此外，参数也不一定是set，也可以是dict或者list，只要是可迭代的(iterables)即可。可以看下python docs

Python

```
# Walking Through a Connected Component
```

```

1 # Walking Through a Connected Component of a Graph Represented Using Adjacency Sets
2 def walk(G, s, S=set()):           # Walk the graph from node s
3     P, Q = dict(), set()           # Predecessors + "to do" queue
4     P[s] = None                   # s has no predecessor
5     Q.add(s)                      # We plan on starting with s
6     while Q:                     # Still nodes to visit
7         u = Q.pop()              # Pick one, arbitrarily
8         for v in G[u].difference(P, S):    # New nodes?
9             Q.add(v)                # We plan to visit them!
10            P[v] = u               # Remember where we came from
11

```

我们可以用下面代码来测试下，得到的结果没有问题

Python

```
def some_graph():
    a, b, c, d, e, f, g, h =
1  def some_graph():
2      a, b, c, d, e, f, g, h = range(8)
3      N = [
4          [b, c, d, e, f],  # a
5          [c, e],           # b
6          [d],             # c
7          [e],             # d
8          [f],             # e
9          [c, g, h],       # f
10         [f, h],          # g
11         [f, g]           # h
12     ]
13     return N
14
15 G = some_graph()
16 for i in range(len(G)): G[i] = set(G[i])
17 print list(walk(G,0)) #[0, 1, 2, 3, 4, 5, 6, 7]
```

上面的walk函数只适用于无向图，而且只能找到一个从参数s出发的连通分量，要想得到全部的连通分量需要修改下

Python

```
def components(G):
# The connected
1 def components(G):                                # The connected components
2     comp = []                                     # Nodes we've already seen
3     seen = set()                                  # Try every starting point
4     for u in G:                                 # Seen? Ignore it
5         if u in seen: continue                   # Traverse component
6         C = walk(G, u)                          # Add keys of C to seen
7         seen.update(C)                         # Collect the components
8         comp.append(C)
9     return comp
```

用下面的代码来测试下，得到的结果没有问题

Python

```
G = {
    0: set([1, 2]),
1  G = {
2      0: set([1, 2]),
3      1: set([0, 2]),
4      2: set([0, 1]),
5      3: set([4, 5]),
6      4: set([3, 5]),
7      5: set([3, 4])
8  }
9
10 print [list(sorted(C)) for C in components(G)] #[[0, 1, 2], [3, 4, 5]]
```

至此我们就完成了一个时间复杂度为 $\Theta(E+V)$ 的求无向图的连通分量的算法，因为每条边和每个顶点

都要访问一次。[这个时间复杂度会经常看到，例如拓扑排序，强连通分量都是它]

[接下来作者作为扩展介绍了欧拉回路和哈密顿回路：前者是经过图中的所有边一次，然后回到起点；后者是经过图中的所有顶点一次，然后回到起点。网上资料甚多，感兴趣自行了解]

下面我们看下迷宫问题，如下图所示，原始问题是一个人在公园中走路，结果走不出来了，即使是按照“左手准则”(也就是但凡遇到交叉口一直向左转)走下去，如果走着走着回到了原来的起点，那么就会陷入无限的循环中！有意思的是，左边的迷宫可以通过“左手准则”转换成右边的树型结构。

[注：具体的转换方式我还未明白，下面是作者给出的构造说明]

Here the “keep one hand on the wall” strategy will work nicely. One way of seeing why it works is to observe that the maze really has only one inner wall (or, to put it another way, if you put wallpaper inside it, you could use one continuous strip). Look at the outer square. As long as you’re not allowed to create cycles, any obstacles you draw have to be connected to the it in exactly one place, and this doesn’t create any problems for the left-hand rule. Following this traversal strategy, you’ll discover all nodes and walk every passage twice (once in either direction).

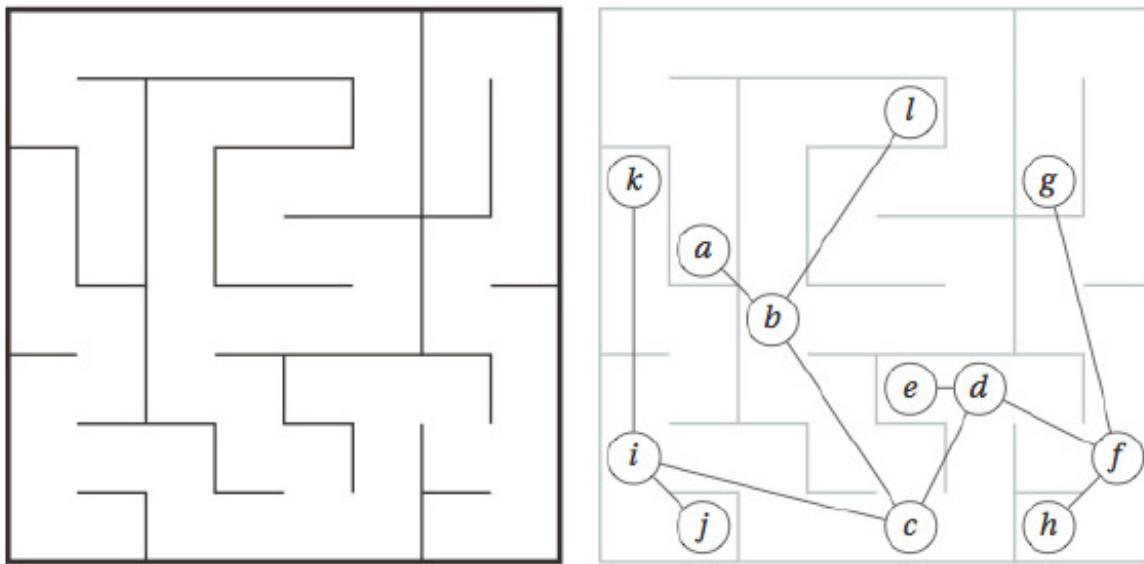


Figure 5-5. A tree, drawn as a maze and as a more conventional graph diagram, superimposed on the maze

上面的迷宫实际上就是为了引出深度优先搜索(DFS)，每次到了一个交叉口的时候，可能我们可以向左走，也可以向右走，选择是有不少，但是我们要向一直走下去的话就只能选择其中的一个方向，如果我们发现这个方向走不出去的话，我们就回溯回来，选择一个刚才没选过的方向继续尝试下去。

基于上面的想法可以写出下面递归版本的DFS

Python

```
def rec_dfs(G, s, S=None):  
    1 def rec_dfs(G, s, S=None):
```

```

2   if S is None: S = set()           # Initialize the history
3   S.add(s)                         # We've visited s
4   for u in G[s]:                  # Explore neighbors
5       if u in S: continue          # Already visited: Skip
6       rec_dfs(G, u, S)            # New: Explore recursively
7   return S # For testing
8
9 G = some_graph()
10 for i in range(len(G)): G[i] = set(G[i])
11 print list(rec_dfs(G, 0)) #[0, 1, 2, 3, 4, 5, 6, 7]

```

很自然的我们想到要将递归版本改成迭代版本的，下面的代码中使用了Python中的yield关键字，具体的用法可以看下[这里IBM Developer Works](#)

Python

```

def iter_dfs(G, s):
    S, Q = set(), []
1 def iter_dfs(G, s):
2     S, Q = set(), []                 # Visited-set and queue
3     Q.append(s)                      # We plan on visiting s
4     while Q:                        # Planned nodes left?
5         u = Q.pop()                 # Get one
6         if u in S: continue          # Already visited? Skip it
7         S.add(u)                     # We've visited it now
8         Q.extend(G[u])              # Schedule all neighbors
9         yield u                      # Report u as visited
10
11 G = some_graph()
12 for i in range(len(G)): G[i] = set(G[i])
13 print list(iter_dfs(G, 0)) #[0, 5, 7, 6, 2, 3, 4, 1]

```

上面迭代版本经过一点点的修改可以得到更加通用的遍历函数

Python

```

def traverse(G, s,
qtype=set):
1 def traverse(G, s, qtype=set):
2     S, Q = set(), qtype()
3     Q.add(s)
4     while Q:
5         u = Q.pop()
6         if u in S: continue
7         S.add(u)
8         for v in G[u]:
9             Q.add(v)
10        yield u

```

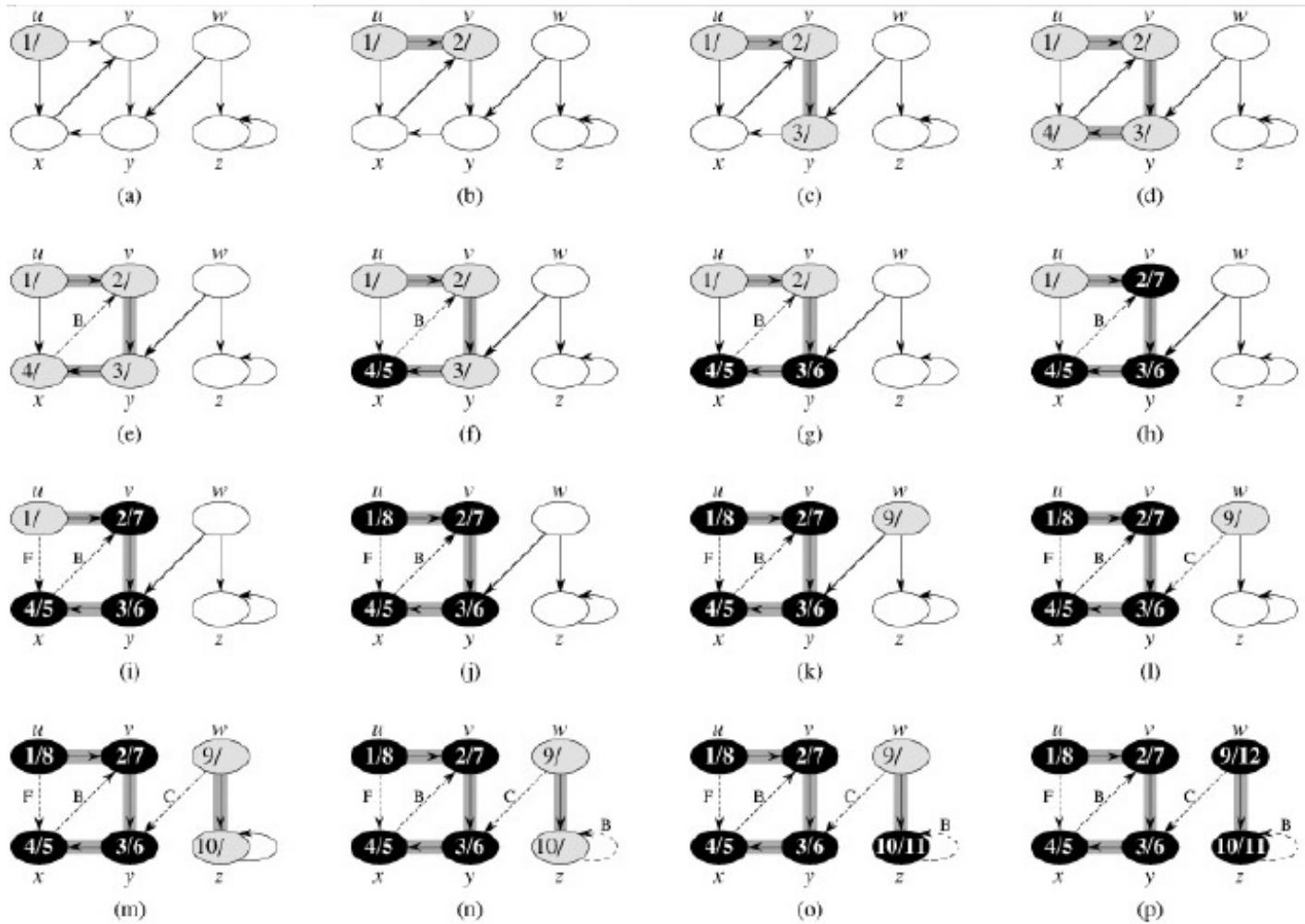
函数traverse中的参数qtype表示队列类型，例如栈stack，下面的代码给出了如何自定义一个stack，以及测试traverse函数

Python

```
class stack(list):
    add = list.append
```

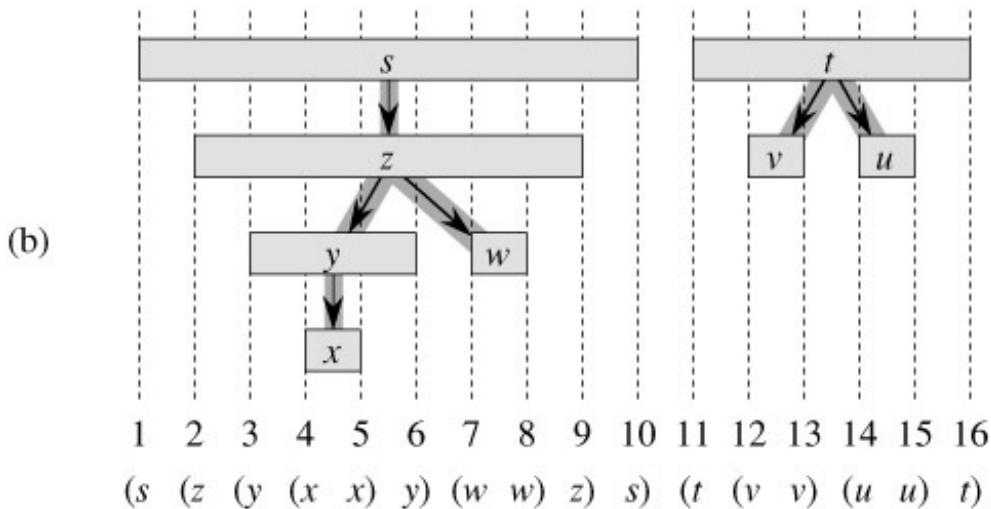
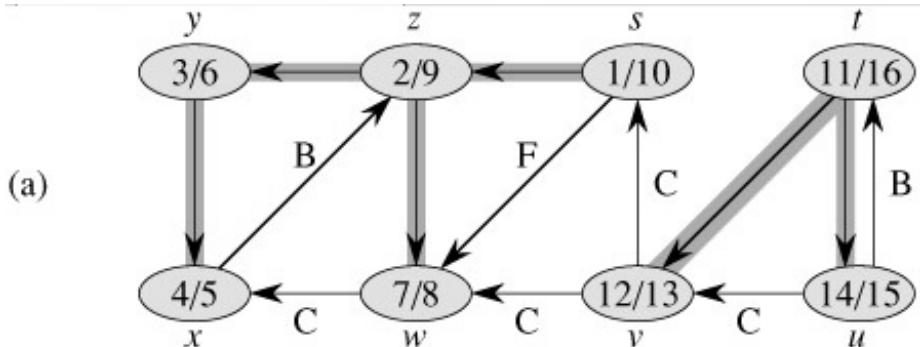
```
1 class stack(list):
2     add = list.append
3
4 G = some_graph()
5 print list(traverse(G, 0, stack)) #[0, 5, 7, 6, 2, 3, 4, 1]
```

如果还不清楚的话可以看下算法导论中的这幅DFS示例图，节点的颜色后面有介绍



上图在DFS时给节点加上了时间戳，这有什么作用呢？

前面提到过，在遍历节点的时候如果给节点标注它的发现节点时间 $d[v]$ 和结束访问时间 $f[v]$ 的话，从这些时间我们就能够发现一些信息，比如下图，(a)是图的一个DFS遍历加上时间戳后的结果；(b)是如果给每个节点的 $d[v]$ 到 $f[v]$ 区间加上一个括号的话，可以看出在DFS遍历中(也就是后来的深度优先树/森林)中所有的节点 u 的后继节点 v 的区间都在节点 u 的区间内部，如果节点 v 不是节点 u 的后继，那么两个节点的区间不相交，这就是“括号定理”。



加上时间戳的DFS遍历还算比较好写对吧

Python

```
#Depth-First Search
with Timestamps
```

- 1 #Depth-First Search with Timestamps
- 2 def dfs(G, s, d, f, S=None, t=0):
- 3 if S is None: S = set() # Initialize the history
- 4 d[s] = t; t += 1 # Set discover time
- 5 S.add(s) # We've visited s
- 6 for u in G[s]: # Explore neighbors
- 7 if u in S: continue # Already visited. Skip
- 8 t = dfs(G, u, d, f, S, t) # Recurse; update timestamp
- 9 f[s] = t; t += 1 # Set finish time
- 10 return t # Return timestamp

除了给节点加上时间戳之外，算法导论在介绍DFS的时候还给节点进行着色，在节点被发现之前是白色的，在发现之后先是灰色的，在结束访问之后才是黑色的，详细的流程可以参考上面给出的算法导论中的那幅DFS示例图。有了颜色有什么用呢？作用大着呢！根据节点的颜色，我们可以对边进行分类！大致可以分为下面四种：

根据在图 G 上进行深度优先搜索所产生的深度优先森林 G_π ，可以把图的边分为四种类型：

1) 树边(tree edge)，是深度优先森林 G_π 中的边。如果顶点 v 是在探寻边 (u, v) 时被首次发现的，那么 (u, v) 就是一条树边。

2) 反向边(back edge)是深度优先树中，连接顶点 u 到它的某一祖先顶点 v 的那些边。有向图中可能出现的自环也被认为是反向边。

3) 正向边(forward edge)是指深度优先树中，连接顶点 u 到它的某个后裔 v 的非树边 (u, v) 。

4) 交叉边(cross edge)是其他类型的边，存在于同一棵深度优先树中的两个顶点之间，条件是其中一个顶点不是另一个顶点的祖先。交叉边也可以在不同的深度优先树的顶点之间。

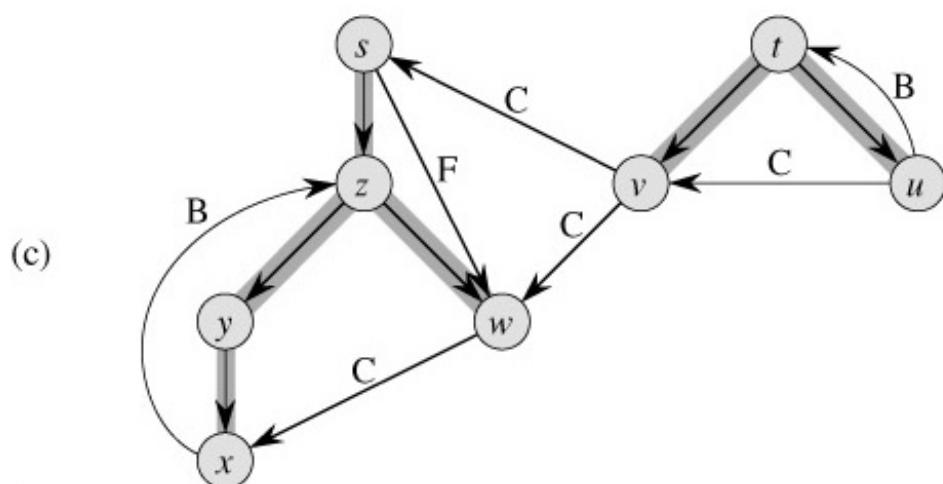
使用DFS对图进行遍历时，对于每条边 (u, v) ，当该边第一次被发现时，根据到达节点 v 的颜色来对边进行分类(正向边和交叉边不做细分)：

(1)白色表示该边是一条树边；

(2)灰色表示该边是一条反向边；

(3)黑色表示该边是一条正向边或者交叉边。

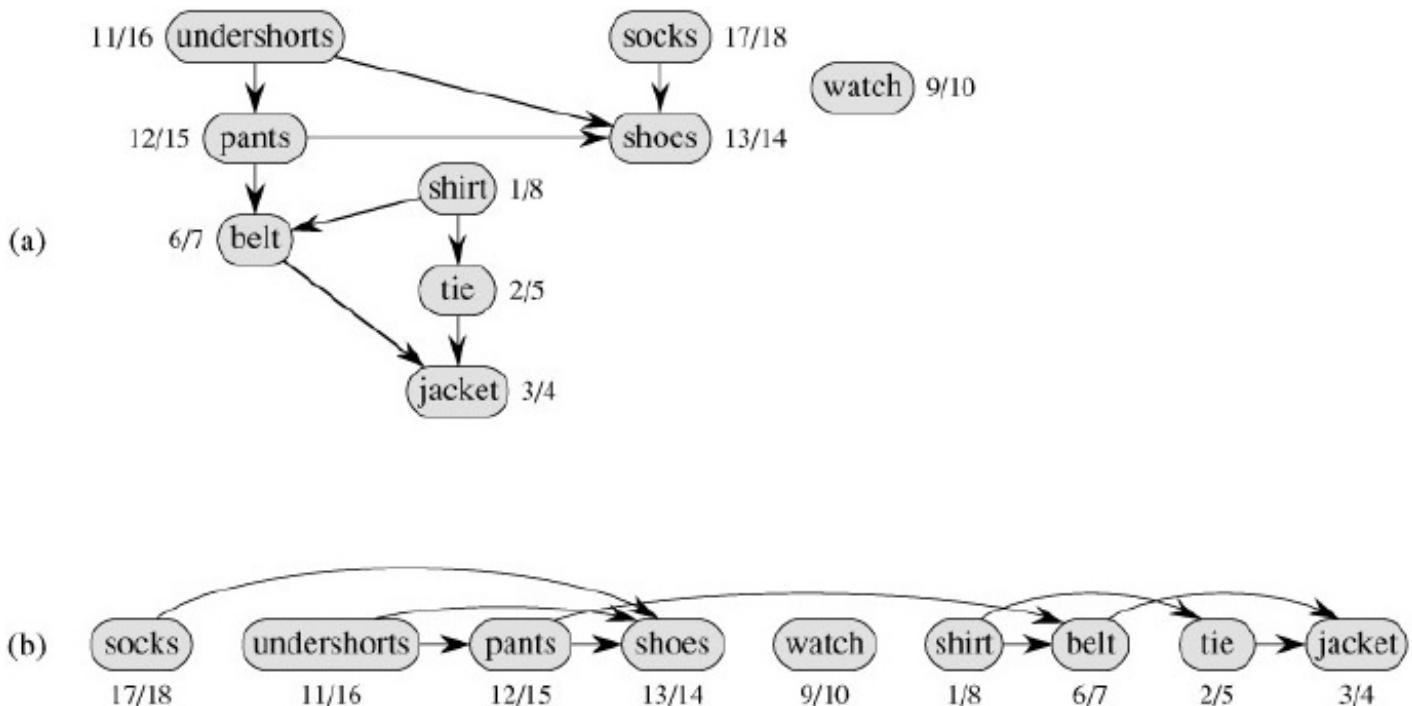
下图显示了上面介绍括号定理用时的那个图的深度优先树中的所有边的类型，灰色标记的边是深度优先树的树边



那对边进行分类有什么作用呢？作用多着呢！最常见的作用的是判断一个有向图是否存在环，如果对有向图进行DFS遍历发现了反向边，那么一定存在环，反之没有环。此外，对于无向图，如果对它进行DFS遍历，肯定不会出现正向边或者交叉边。

那对节点标注时间戳有什么用呢？其实，除了可以发现上面提到的那些很重要的性质之外，时间戳对于接下来要介绍的拓扑排序的另一种解法和强连通分量很重要！

我们先看下摘自算法导论的这幅拓扑排序示例图，这是某个教授早上起来后要做的事情，嘿嘿



不难发现，最终得到的拓扑排序刚好是节点的完成时间 $f[v]$ 降序排列的！结合前面的括号定理以及依赖关系不难理解，如果我们按照节点的 $f[v]$ 降序排列，我们就得到了我们想要的拓扑排序了！这就是拓扑排序的另一个解法！[在算法导论中该解法是主要介绍的解法，而我们前面提到的那个解法是在算法导论的习题中出现的]

基于上面的想法就能够得到下面的实现代码，函数reurse是一个内部函数，这样它就可以访问到G和res等变量

Python

```
#Topological Sorting
Based on Depth-First Search
```

- 1 #Topological Sorting Based on Depth-First Search
- 2 def dfs_topsort(G):
- 3 S, res = set(), []
- 4 def recurse(u):
- 5 if u in S: return
- 6 S.add(u)
- 7 for v in G[u]:
- 8 recurse(v)
- 9 res.append(u)
- 10 for u in G:
- 11 recurse(u)
- 12 res.reverse()
- 13 return res
- 14
- 15 G = {'a': set('bf'), 'b': set('cdf'), 'c': set('d'), 'd': set('ef'), 'e': set('f'), 'f': set()}
- 16 print(dfs_topsort(G))

[接下来作者介绍了一个Iterative Deepening Depth-First Search，没看懂，貌似和BFS类似]

如果我们在遍历图时“一层一层”式地遍历，先发现的节点先访问，那么我们就得到了广度优先搜索(BFS)。下面是作者给出的一个有意思的区别BFS和DFS的例子，遍历过程就像我们上网一样，DFS是顺着网页上的链接一个个点下去，当访问完了这个网页时就点击Back回退到上一个网页继续访问。而BFS是先在后台打开当前网页上的所有链接，然后按照打开的顺序一个个访问，访问完了一个网页就把它的窗口关闭。

One way of visualizing BFS and DFS is as browsing the Web. DFS is what you get if you keep following links and then use the Back button once you're done with a page. The backtracking is a bit like an “undo.” BFS is more like opening every link in a new window (or tab) behind those you already have and then closing the windows as you finish with each page.

BFS的代码很好实现，主要是使用队列

Python

```
#Breadth-First Search
from collections import deque

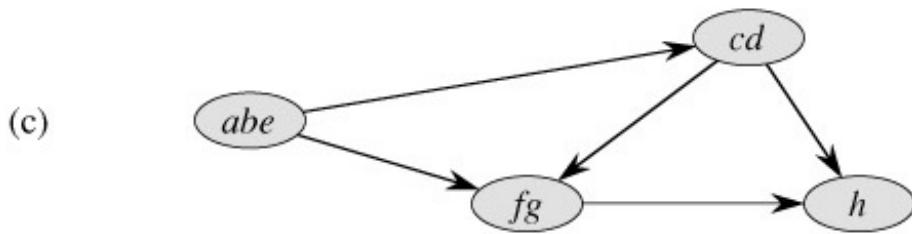
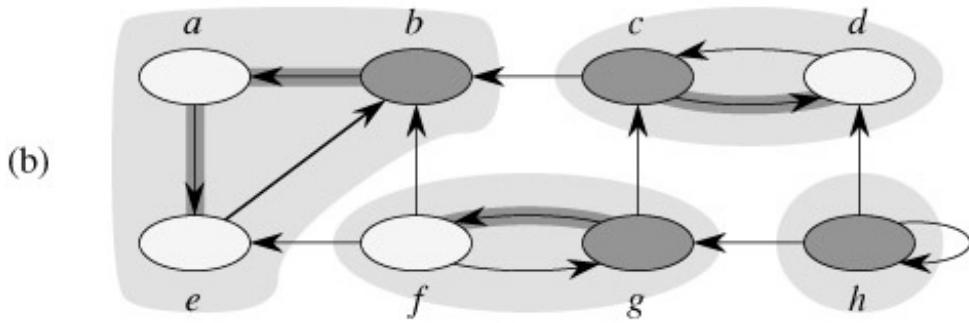
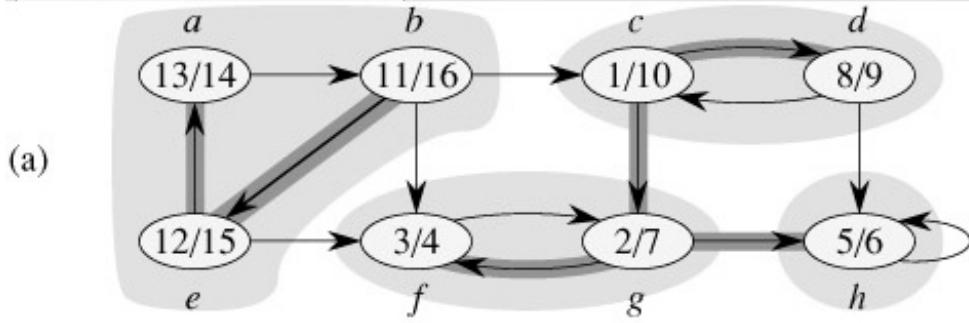
1 #Breadth-First Search
2 from collections import deque
3
4 def bfs(G, s):
5     P, Q = {s: None}, deque([s])           # Parents and FIFO queue
6     while Q:
7         u = Q.popleft()                  # Constant-time for deque
8         for v in G[u]:
9             if v in P: continue          # Already has parent
10            P[v] = u                   # Reached from u: u is parent
11            Q.append(v)
12    return P
13
14 G = some_graph()
15 print bfs(G, 0)
```

Python的list可以很好地充当stack，但是充当queue则性能很差，函数bfs中使用的是collections模块中的deque，即双端队列(double-ended queue)，它一般是使用链表来实现的，这个类有extend、append和pop等方法都是作用于队列右端的，而方法extendleft、appendleft和popleft等方法都是作用于队列左端的，它的内部实现是非常高效的。

Internally, the deque is implemented as a doubly linked list of blocks, each of which is an array of individual elements. Although asymptotically equivalent to using a linked list of individual elements, this reduces overhead and makes it more efficient in practice. For example, the expression $d[k]$ would require traversing the first k elements of the deque d if it were a plain list. If each block contains b elements, you would only have to traverse k/b blocks.

最后我们看下强连通分量，前面的分量是不考虑边的方向的，如果我们考虑边的方向，而且得到的最大子图中，任何两个节点都能够沿着边可达，那么这就是一个强连通分量。

下图是算法导论中的示例图，(a)是对图进行DFS遍历带时间戳的结果；(b)是上图的转置，也就是将上图中所有边的指向反转过来得到的图；(c)是最终得到的强连通分支图，每个节点内部显示了该分支内的节点。



上面的示例图自然不太好明白到底怎么得到的，我们慢慢来分析三幅图 [原书的分析太多了，我被绕晕了+_+，下面是我结合算法导论的分析过程]

先看图(a)，每个灰色区域都是一个强连通分支，我们想想，如果强连通分支 X 内部有一条边指向另一个强连通分支 Y，那么强连通分支 Y 内部肯定不存在一条边指向另一个强连通分支 Y，否则它们能够整合在一起形成一个新的更大气的强连通分支！这也就是说强连通分支图肯定是一个有向无环图！我们从图(c)也可以看出来

再看看图(c)，强连通分支之间的指向，如果我们定义每个分支内的任何顶点的最晚的完成时间为对应分支的完成时间的话，那么分支abe的完成时间是16，分支cd是10，分支fg是7，分支h是6，不难发现，分支之间边的指向都是从完成时间大的指向完成时间小的，换句话说，总是由完成时间晚的强连通分支指向完成时间早的强连通分支！

最后再看看图(b)，该图是原图的转置，但是得到强连通分支是一样的(强连通分支图是会变的，刚好又是原来分支图的转置)，那为什么要将边反转呢？结合前面两个图的分析，既然强连通分支图是有向无环图，而且总是由完成时间晚的强连通分支指向完成时间早的强连通分支，如果我们将边反转，虽然我们得到的强连通分支不变，但是分支之间的指向变了，完成时间晚的就不再指向完成时间早的了！这样的话如果我们对它进行拓扑排序，即按照完成时间的降序再次进行DFS时，我们就能够得到一个个的强连通分支了对不对？因为每次得到的强连通分支都没有办法指向其他分支了，也就是确定了一个强连通分支之后就停止了。[试试画个图得到图(b)的强连通分支图的拓扑排序结果就明白了]

经过上面略微复杂的分析之后我们知道强连通分支算法的流程有下面四步：

- 1.对原图G运行DFS，得到每个节点的完成时间f[v];
- 2.得到原图的转置图GT；
- 3.对GT运行DFS，主循环按照节点的f[v]降序进行访问；
- 4.输出深度优先森林中的每棵树，也就是一个强连通分支。

根据上面的思路可以得到下面的强连通分支算法实现，其中的函数parse_graph是作者用来方便构造图的函数

Python

```
def tr(G):
    # Transpose (rev.
    # Transpose (rev. edges of) G
    GT = {}
    for u in G: GT[u] = set()
    for u in G:
        for v in G[u]:
            GT[v].add(u)
    return GT

def scc(G):
    GT = tr(G)
    sccs, seen = [], set()
    for u in dfs_topsort(G):
        if u in seen: continue
        C = walk(GT, u, seen)
        seen.update(C)
        sccs.append(C)
    return sccs

from string import ascii_lowercase
def parse_graph(s):
    # print zip(ascii_lowercase, s.split("/"))
    # [('a', 'bc'), ('b', 'die'), ('c', 'd'), ('d', 'ah'), ('e', 'f'), ('f', 'g'), ('g', 'eh'), ('h', 'i'), ('i', 'h')]
    G = {}
    for u, line in zip(ascii_lowercase, s.split("/")):
        G[u] = set(line)
    return G

G = parse_graph('bc/die/d/ah/f/g/eh/i/h')
print list(map(list, scc(G)))
#[['a', 'c', 'b', 'd'], ['e', 'g', 'f'], ['i', 'h']]
```

[最后作者提到了一点如何进行更加高效的搜索，也就是通过分支限界来实现对搜索树的剪枝，具体使用可以看下这个问题顶点覆盖问题Vertex Cover Problem]

问题5.17 强连通分支

In Kosaraju's algorithm, we find starting nodes for the final traversal by descending finish times from

an initial DFS, and we perform the traversal in the transposed graph (that is, with all edges reversed). Why couldn't we just use ascending finish times in the original graph?

问题就是说，我们干嘛要对转置图按照完成时间降序遍历一次呢？干嘛不直接在原图上按照完成时间升序遍历一次呢？

Try finding a simple example where this would give the wrong answer. (You can do it with a really small graph.)

1 赞 1 收藏 [评论](#)

Python算法：Counting 101

原文地址：[hujiawei \(@五道口宅男\)](#)

原书主要介绍了一些基础数学，例如排列组合以及递归循环等，但是本节只重点介绍计算算法的运行时间的三种方法

因为本节内容都很简单，所以我只是浏览了一下，重要的只有计算算法的运行时间的三种方法：1.代换法； 2.递归树法； 3.主定理法。

1.代换法

代换法一般是先猜测解的形式，然后用数学归纳法来证明它

下面是算法导论中的一个求解例子

代换法可用来确定一个递归式的上界或下界。作为例子，让我们确定递归式

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \quad (4.4)$$

的一个上界，这个式子与递归式(4.2)和式(4.3)类似。我们猜其解为 $T(n)=O(n \lg n)$ 。我们的方法是证明 $T(n) \leq cn \lg n$ ，其中 $c > 0$ 是某个常数。先假设这个界对 $\lfloor n/2 \rfloor$ 成立，即 $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$ ，对递归式作替换，得：

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n = cn \lg n - cn + n \leq cn \lg n \end{aligned}$$

最后一步只要 $c \geq 1$ 就成立。

接下来应用数学归纳法就要求解对边界条件成立。一般来说，可以通过证明边界条件符合归纳证明的基本情况来说明它的正确性。对于递归式(4.4)，必须证明能够选择足够大的常数 c ，使界 $T(n) \leq cn \lg n$ 也对边界条件成立。这些要求有时会导致问题。假设 $T(1)=1$ 是递归式唯一的边界条件。那么对于 $n=1$ 时，界 $T(n) \leq cn \lg n$ 也就是 $T(1) \leq c1 \lg 1=0$ ，与 $T(1)=1$ 不符。因此，归纳证明的基本情况不能满足。

对特殊边界条件证明归纳假设中的这种困难很容易解决。例如，对递归式(4.4)，利用渐近记号，只要求对 $n \geq n_0$ ，证明 $T(n) \leq cn \lg n$ ，其中 n_0 是常数。这样做的思想是在归纳证明中，对困难的边界条件 $T(1)=1$ 不加考虑。我们可以把 $T(2)$ 和 $T(3)$ 作为归纳证明中的边界条件代替 $T(1)$ ，让 $n_0=2$ ，这是因为对 $n>3$ ，递归不直接依赖于 $T(1)$ 。我们将递归式的基本情况($n=1$)和归纳证明的基本情况($n=2$ 和 $n=3$)区别开了。通过递归式，可以求解出 $T(2)=4$ 和 $T(3)=5$ 。归纳证明 $T(n) \leq cn \lg n$ 正确性时，其中常量 $c \geq 1$ ，只要 c 取足够大的常数使 $T(2) \leq c2 \lg 2$ 和 $T(3) \leq c3 \lg 3$ 即可完成证明。实际上，选取任何 $c \geq 2$ ， $n=2$ 和 $n=3$ 都可满足这个要求。对后面将要讨论的大部分递归式，可以直接扩展边界条件，使递归假设对很小的 n 也成立。

有意思的是，还有一类问题可以通过变量替换变成容易求解的形式

改变变量

有时，对一个陌生的递归式作一些简单的代数变换，就会使之变成读者较熟悉的形式。作为一个例子，考虑

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$$

这个式子看上去较难，但可以对它进行简化，方法是改动变量。为了方便起见，不考虑数的截取整数问题，如将 \sqrt{n} 化为整数，设 $m = \lg n$ ，得

$$T(2^m) = 2T(2^{m/2}) + m$$

再设 $S(m) = T(2^m)$ ，得到新的递归式

$$S(m) = 2S(m/2) + m$$

这个式子看起来与式(4.4)就非常像了，这个新的递归式的界是： $S(m) = O(m \lg m)$ 。将 $S(m)$ 代回 $T(n)$ ，有 $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$ 。

下面是常用的一些递归式以及它们对应的结果还有实际算法实例

#	Recurrence	Solution	Example Applications
1	$T(n) = T(n-1) + 1$	$\Theta(n)$	Processing a sequence, for example, with reduce
2	$T(n) = T(n-1) + n$	$\Theta(n^2)$	Handshake problem
3	$T(n) = 2T(n-1) + 1$	$\Theta(2^n)$	Towers of Hanoi
4	$T(n) = 2T(n-1) + n$	$\Theta(2^n)$	
5	$T(n) = T(n/2) + 1$	$\Theta(\lg n)$	Binary search (see the black box sidebar on <code>bisect</code> in Chapter 6)
6	$T(n) = T(n/2) + n$	$\Theta(n)$	Randomized Select, average case (see Chapter 6)
7	$T(n) = 2T(n/2) + 1$	$\Theta(n)$	Tree traversal (see Chapter 5)
8	$T(n) = 2T(n/2) + n$	$\Theta(n \lg n)$	Sorting by <i>divide and conquer</i> (see Chapter 6)

2. 递归树法

这种方法就是通过画递归树，然后对每层进行求和，最后将每层的结果相加得到对总的算法运行时间的估计

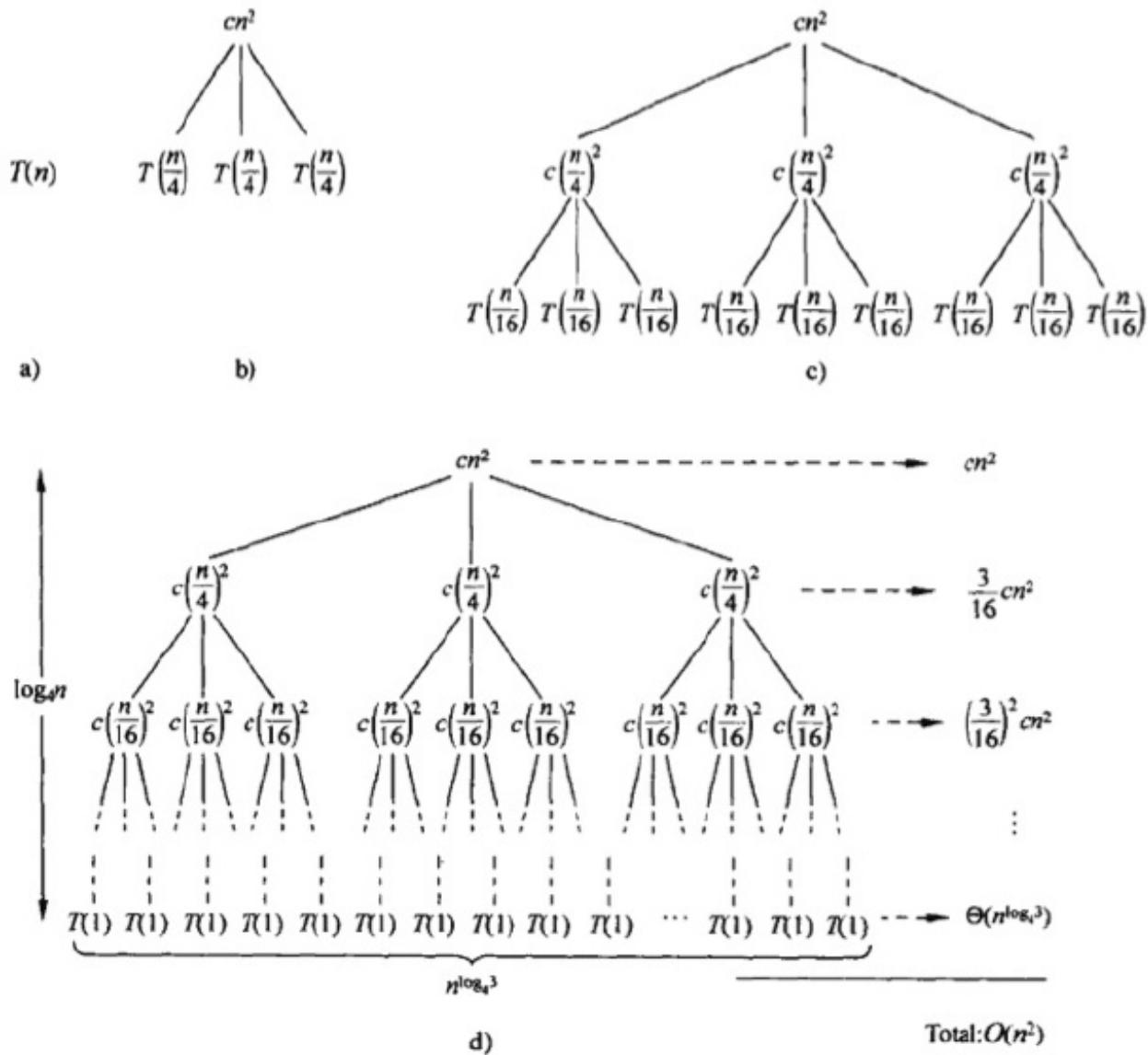


图 4-1 与递归式 $T(n) = 3T(n/4) + cn^2$ 对应的递归树的构造。a)示出了 $T(n)$ ，它在 b)~d)中被不断地扩展而形成了递归树。d)完全扩展了的递归树深度为 $\log_4 n$ (它有 $\log_4 n + 1$ 层)

3. 主定理法

这种方法大家最喜欢，给出了一种就像是公式一样的结论，虽然它没有覆盖所有的情况，而且证明非常复杂，但是很多情况下都是可以直接使用的，还有，需要注意主定理的不同情况下的条件，尤其是多项式大于和多项式小于！

Table 3-2. The Three Cases of the Master Theorem

Case	Condition	Solution	Example
1	$f(n) \in O(n^{\log_b a - \varepsilon})$	$T(n) \in \Theta(n^{\log_b a})$	$T(n) = 2T(n/2) + \lg n$
2	$f(n) \in \Theta(n^{\log_b a})$	$T(n) \in \Theta(n^{\log_b a} \lg n)$	$T(n) = 2T(n/4) + \sqrt{n}$
3	$f(n) \in \Omega(n^{\log_b a + \varepsilon})$	$T(n) \in \Theta(f(n))$	$T(n) = 2T(n/3) + n$

不喜欢本节的可以跳过，不留练习了这次，嘿嘿，想练习的话刷算法导论的题目吧

1 赞 1 收藏 [评论](#)

Python算法：推导、递归和规约

原文出处：[hujiawei \(@五道口宅男\)](#)

注：本节中我给定下面三个重要词汇的中文翻译分别是：Induction(推导)、Recursion(递归)和Reduction(规约)

本节主要介绍算法设计的三个核心知识：Induction(推导)、Recursion(递归)和Reduction(规约)，这是原书的重点和难点部分

正如标题所示，本节主要介绍下面三部分内容：

- Reduction means transforming one problem to another. We normally reduce an unknown problem to one we know how to solve. The reduction may involve transforming both the input (so it works with the new problem) and the output (so it's valid for the original problem).

Reduction(规约)意味着对问题进行转换，例如将一个未知的问题转换成我们能够解决的问题，转换的过程可能涉及到对问题的输入输出的转换。[问题规约在证明一个问题是否是NP完全问题时经常用到，如果我们能够将一个问题规约成一个我们已知的NP完全问题的话，那么这个问题也是NP完全问题]

下面给幅图你就能够明白了，实际上很多时候我们遇到一个问题时都是找一个我们已知的类似能够解决的问题，然后将这个我们新问题A规约到那个已知的问题B，中间经过一些输入输出的转换，我们就能够解决新问题A了。

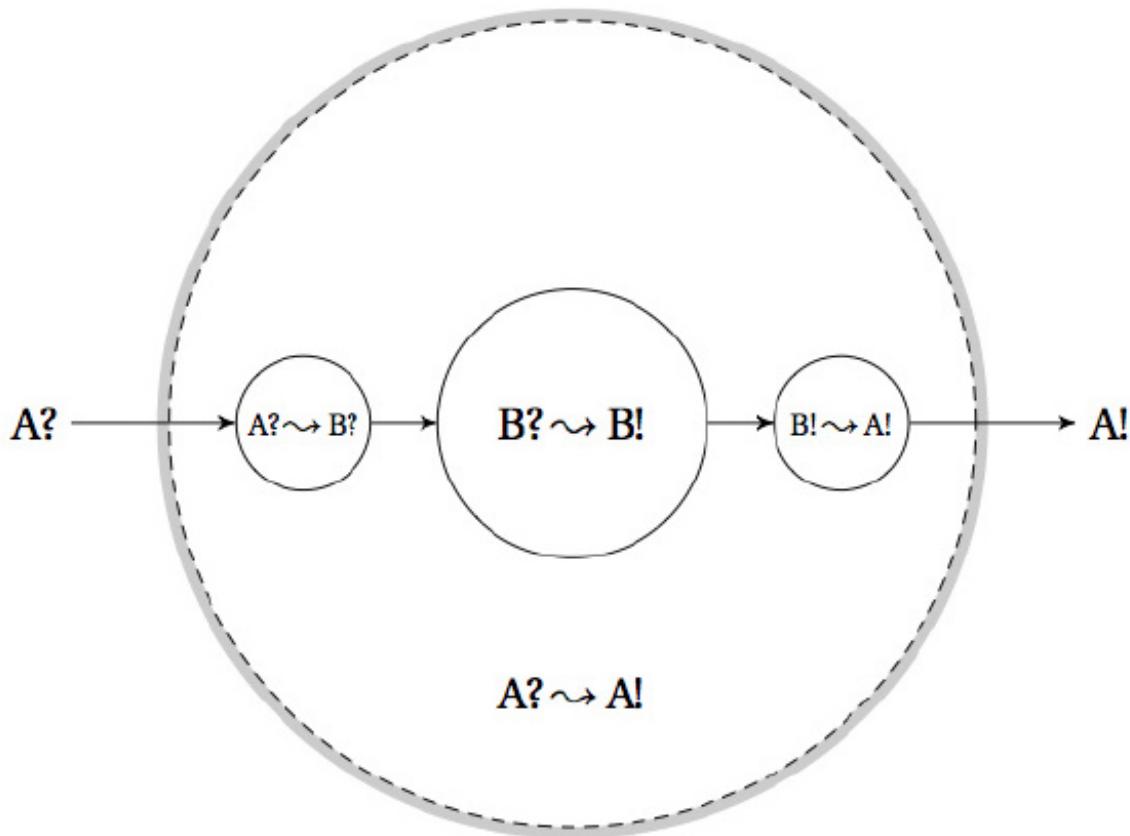


Figure 4-1. Using a reduction from A to B to solve A with an algorithm for B. The algorithm for B (the central, inner circle) can transform the input B? to the output B!, while the reduction consists of the two transformation (the smaller circles) going from A? to B? and from B! to A!, together forming the main algorithm, which transforms the input A? to the output A!

- Induction (or, mathematical induction) is used to show that a statement is true for a large class of objects (often the natural numbers). We do this by first showing it to be true for a base case (such as the number 1) and then showing that it “carries over” from one object to the next (if it’s true for $n - 1$, then it’s true for n).

Induction(推导)是一个数学意义上的推导，类似数学归纳法，主要是用来证明某个命题是正确的。首先我们证明对于基础情况(例如在 $k=1$ 时)是正确的，然后证明该命题递推下去都是正确的(一般假设当 $k=n-1$ 时是正确的，然后证明当 $k=n$ 时也是正确的即可)

- Recursion is what happens when a function calls itself. Here we need to make sure the function works correctly for a (nonrecursive) base case and that it combines results from the recursive calls into a valid solution.

Recursion(递归)经常发生于一个函数调用自身的情况。递归函数说起来简单，但是实现不太容易，我们要确保对于基础情况(不递归的情况)能够正常工作，此外，对于递归情况能够将递归调用的结果组合起来得到一个有效的结果。

以上三个核心有很多相似点，比如它们都专注于求出目标解的某一步，我们只需要仔细思考这一步，剩下的就能够自动完成了。如果我们更加仔细地去理解它们，我们会发现，**Induction(推导)**和**Recursion(递归)**其实彼此相互对应，也就是说一个**Induction**能够写出一个相应的**Recursion**，而一

个Recursion正好对应着一个Induction式子，也可以换个方式理解，Induction是从n-1到n的推导，而Recursion是从n到n-1的递归(下面有附图可以帮助理解)。此外，Induction和Recursion其实都是某种Reduction，即Induction和Recursion的本质就是对问题进行规约！为了能够对问题使用Induction或者说Recursion，Reduction一般是将一个问题变成另一个只是规模减小了的相同问题。

你也许会觉得奇怪，不对啊，刚才不是说Reduction是将一个问题规约成另一个问题吗？现在怎么又说成是将一个问题变成另一个只是规模减小了的相同问题了？其实，Reduction是有两种的，上面的两种都是Reduction！还记得前面介绍过的递归树吗？那其实就是将规模较大的问题转换成几个规模较小的问题，而且问题的形式并没有改变，这就是一种Reduction。你可以理解这种情况下Reduction是降维的含义，也就类似机器学习中的Dimension Reduction，对高维数据进行降维了，问题保持不变。

These are two major variations of reductions: reducing to a different problem and reducing to a shrunken version of the same.

再看下下面这幅图理解Induction和Recursion之间的关系

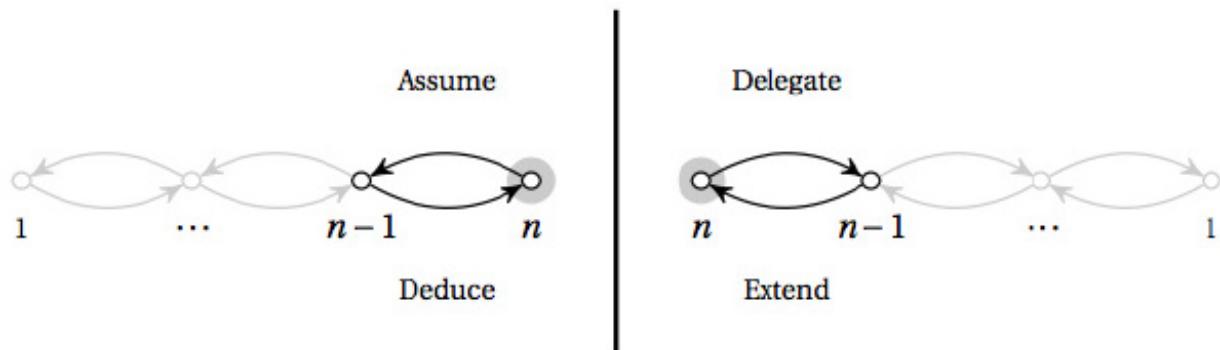


Figure 4-3. Induction (on the left) and recursion (on the right), as mirror images of each other

[关于它们三个的关系的原文阐述：Induction and recursion are, in a sense, mirror images of one another, and both can be seen as examples of reduction. To use induction (or recursion), the reduction must (generally) be between instances of the same problem of different sizes.]

[看了原书你会觉得，作者介绍算法的方式很特别，作者有提到他的灵感来自哪里：In fact, much of the material was inspired by Udi Manber's wonderful paper “Using induction to design algorithms” from 1988 and his book from the following year, Introduction to Algorithms: A Creative Approach.]

也许你还感觉很晕，慢慢地看了后面的例子你就明白了。在介绍例子之前呢，先看下递归和迭代的异同，这个很重要，在后面介绍动态规划算法时我们还会反复提到它们的异同。

[Induction is what you use to show that recursion is correct, and recursion is a very direct way of implementing most inductive algorithm ideas. However, rewriting the algorithm to be iterative can avoid the overhead and limitations of recursive functions in most (nonfunctional) programming languages.]

有了Induction和Recursion，我们很容易就可以将一个inductive idea采用递归(recursion)的方式实现，根据我们的编程经验(事实也是如此)，任何一个递归方式的实现都可以改成非递归方式(即迭代方式)实现(反之亦然)，而且非递归方式要好些，为什么呢？因为非递归版本相对来讲运行速度更快，因为没有用栈去实现，也避免了栈溢出的情况，python中对栈深度是有限制的。

举个例子，下面是一段遍历序列的代码，如果大小设置为100没有问题，如果设置为1000就会报RuntimeError的错误，提示超出了最大的递归深度。[当然，大家都不会像下面那样写代码对吧，这只是一个例子]

Python

```
def trav(seq, i=0):
    if i == len(seq): return
1 def trav(seq, i=0):
2     if i == len(seq): return
3     #print seq[i]
4     trav(seq, i + 1)
5
6 trav(range(1000)) # RuntimeError: maximum recursion depth exceeded
```

所以呢，很多时候虽然递归的思路更好想，代码也更好写，但是迭代的代码更加高效一些，在动态规划中还可以看到迭代版本还有其他的优点，当然，它还有些缺点，比如要考虑迭代的顺序，如果迫不及待想知道请移步阅读[Python算法设计篇之动态规划](#)，不过还是建议且听我慢慢道来

下面我们通过排序来梳理下我们前面介绍的三个核心内容

我们如何对排序问题进行reduce呢？很显然，有很多种方式，假如我们将原问题reduce成两个规模为原来一半的子问题，我们就得到了合并排序(这个我们以后还会详细介绍)；假如我们每次只是reduce一个元素，比如假设前n-1个元素都排好序了，那么我们只需要将第n个元素插入到前面的序列即可，这样我们就得到了插入排序；再比如，假设我们找到其中最大的元素然后将它放在位置n上，一直这么下去我们就得到了选择排序；继续思考下去，假设我们找到某个元素(比如第k大的元素)，然后将它放在位置k上，一直这么下去我们就得到了快速排序(这个我们以后还会详细介绍)。怎么样？我们前面学过的排序经过这么一些reduce基本上都很清晰了对吧？

下面通过代码来体会下插入排序和选择排序的两个不同版本

递归版本的插入排序

Python

```
def ins_sort_rec(seq, i):
    if i == 0: return #
1 def ins_sort_rec(seq, i):
2     if i == 0: return # Base case -- do nothing
3     ins_sort_rec(seq, i - 1) # Sort 0..i-1
4     j = i # Start "walking" down
5     while j > 0 and seq[j - 1] > seq[j]: # Look for OK spot
6         seq[j - 1], seq[j] = seq[j], seq[j - 1] # Keep moving seq[j] down
7         j -= 1 # Decrement j
8
9 from random import randrange
10 seq = [randrange(1000) for i in range(100)]
```

```
11 ins_sort_rec(seq, len(seq)-1)
```

改成迭代版本的插入排序如下

Python

```
def ins_sort(seq):  
    for i in range(1,  
1 def ins_sort(seq):  
2     for i in range(1, len(seq)): # 0..i-1 sorted so far  
3         j = i # Start "walking" down  
4         while j > 0 and seq[j - 1] > seq[j]: # Look for OK spot  
5             seq[j - 1], seq[j] = seq[j], seq[j - 1] # Keep moving seq[j] down  
6             j -= 1 # Decrement j  
7  
8 seq2 = [randrange(1000) for i in range(100)]  
9 ins_sort(seq2)
```

你会发现，两个版本差不多，但是递归版本中list的size不能太大，否则就会栈溢出，而迭代版本不会有这些问题，还有一个区别就是方法参数，一般来说递归版本的参数都会多些

递归版本和迭代版本的选择排序

Python

```
def sel_sort_rec(seq, i):  
    if i == 0: return #  
  
1 def sel_sort_rec(seq, i):  
2     if i == 0: return # Base case -- do nothing  
3     max_j = i # Idx. of largest value so far  
4     for j in range(i): # Look for a larger value  
5         if seq[j] > seq[max_j]: max_j = j # Found one? Update max_j  
6         seq[i], seq[max_j] = seq[max_j], seq[i] # Switch largest into place  
7     sel_sort_rec(seq, i - 1) # Sort 0..i-1  
8  
9 seq = [randrange(1000) for i in range(100)]  
10 sel_sort_rec(seq, len(seq)-1)  
11  
12 def sel_sort(seq):  
13     for i in range(len(seq) - 1, 0, -1): # n..i+1 sorted so far  
14         max_j = i # Idx. of largest value so far  
15         for j in range(i): # Look for a larger value  
16             if seq[j] > seq[max_j]: max_j = j # Found one? Update max_j  
17             seq[i], seq[max_j] = seq[max_j], seq[i] # Switch largest into place  
18  
19 seq2 = [randrange(1000) for i in range(100)]  
20 sel_sort(seq2)
```

下面我们来看个例子，这是一个经典的“名人问题”，我们要从人群中找到那个名人，所有人都认识名人，而名人则任何人都不认识。

[这个问题的一个变种就是从一系列有依赖关系的集合中找到那个依赖关系最开始的元素，比如多线程环境下的线程依赖问题，后面将要介绍的拓扑排序是解决这类问题更实际的解法。A more down-to-earth version of the same problem would be examining a set of dependencies and trying to find a place to start. For example, you might have threads in a multithreaded application waiting for each other, with even some cyclical dependencies (so-called deadlocks), and you're looking for one

thread that isn't waiting for any of the others but that all of the others are dependent on.]

在进一步分析之前我们可以发现，很显然，我们可以暴力求解下， $G[u][v]$ 为True表示 u 认识 v。

Python

```
def naive_celeb(G):
    n = len(G)

1 def naive_celeb(G):
2     n = len(G)
3     for u in range(n): # For every candidate...
4         for v in range(n): # For everyone else...
5             if u == v: continue # Same person? Skip.
6             if G[u][v]: break # Candidate knows other
7             if not G[v][u]: break # Other doesn't know candidate
8         else:
9             return u # No breaks? Celebrity!
10    return None # Couldn't find anyone
```

用下面代码进行测试，得到正确结果57

Python

```
from random import *
n = 100

1 from random import *
2 n = 100
3 G = [[randrange(2) for i in range(n)] for i in range(n)]
4 c = 57 # For testing
5 for i in range(n):
6     G[i][c] = True
7     G[c][i] = False
8
9 print naive_celeb(G) #57
```

上面的暴力求解其实可以看做是一个reduce，每次reduce一个人，确定他是否是名人，显然这样做并不高效。那么，对于名人问题我们还可以怎么reduce呢？假设我们还是将规模为n的问题reduce成规模为n-1的问题，那么我们要找到一个非名人(u)，也就是找到一个人(u)，他要么认识其他某个人(v)，要么某个人(v)不认识他，也就是说，对于任何 $G[u][v]$ ，如果 $G[u][v]$ 为True，那么消去u；如果 $G[u][v]$ 为False，那么消去v，这样就可以明显加快查找的速度！

基于上面的想法就有了下面的python实现，第二个for循环是用来验证我们得到的结果是否正确(因为如果我们保证有一个名人的话那么结果肯定正确，但是如果不能保证的话，那么结果就要进行验证)

Python

```
def celeb(G):
    n = len(G)

1 def celeb(G):
2     n = len(G)
3     u, v = 0, 1 # The first two
4     for c in range(2, n + 1): # Others to check
5         if G[u][v]:
6             u = c # u knows v? Replace u
7         else:
```

```

8     v = c # Otherwise, replace v
9     if u == n:
10        c = v # u was replaced last; use v
11    else:
12        c = u # Otherwise, u is a candidate
13    for v in range(n): # For everyone else...
14        if c == v: continue # Same person? Skip.
15        if G[c][v]: break # Candidate knows other
16        if not G[v][c]: break # Other doesn't know candidate
17    else:
18        return c # No breaks? Celebrity!
19    return None # Couldn't find anyone

```

看起来还不错吧，我们将一个 $O(n^2)$ 的暴力解法变成了一个 $O(n)$ 的快速解法。

[看书看到这里时，我想起了另一个看起来很相似的问题，从n个元素中找出最大值和最小值。如果我们单独地来查找最大值和最小值，共需要 $(2n-2)$ 次比较(也许你觉得还可以少几次，但都还是和 $2n$ 差不多对吧)，但是，如果我们成对来处理，首先比较第一个元素和第二个元素，较大的那个作为当前最大值，较小的那个作为当前最小值(如果n是奇数的话，为了方便可以直接令第一个元素既是最大值又是最小值)，然后向后移动，每次取两个元素出来先比较，较小的那个去和当前最小值比较，较大的那个去和当前最大值比较，这样的策略至多需要 $3\lfloor n/2 \rfloor$ 次比较。两个问题虽然完全没关系，但是解决方式总有那么点千丝万缕有木有？]

接下来我们看另一个更加重要的例子，拓扑排序，这是图中很重要的一个算法，在后面介绍到图算法的时候我们还会提到拓扑排序的另一个解法，它的应用范围也非常广，除了前面的依赖关系例子外，还有一个最突出的例子就是类Linux系统中软件的安装，每当我们终端安装一个软件或者库时，它会自动检测它所依赖的那些部件(components)是否安装了，如果没有那么就先安装那些依赖项。此外，[后面介绍到动态规划时有一个单源最短路径问题](#)就利用了拓扑排序。

下图是一个有向无环图(DAG)和它对应的拓扑排序结果

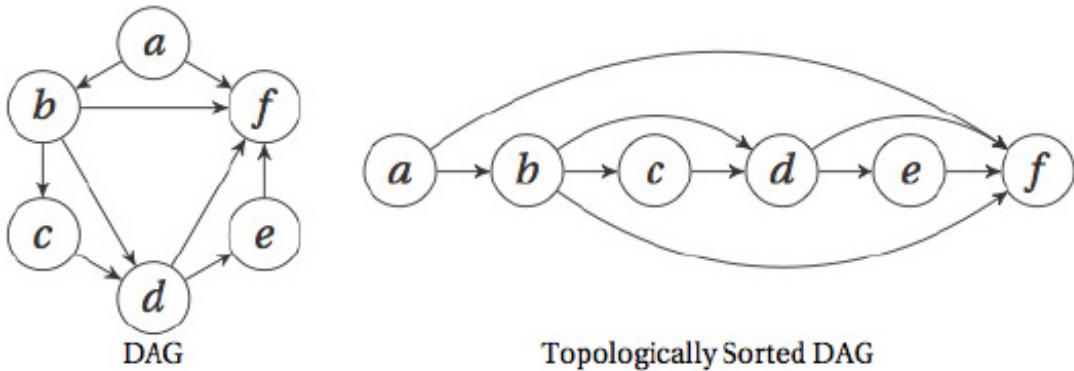


Figure 4-5. A directed acyclic graph (DAG) and its nodes in topologically sorted order

拓扑排序这个问题怎么进行reduce呢？和前面一样，我们最直接的想法可能还是reduce one element，即去掉一个节点，先解决剩下的 $(n-1)$ 个节点的拓扑排序问题，然后将这个去掉的节点插入到合适的位置，这个想法的实现非常类似前面的插入排序，插入的这个节点(也就是前面去掉的节点)的位置是在前面所有对它有依赖的节点之后。

```

def naive_topsort(G,
S=None):
1 def naive_topsort(G, S=None):
2     if S is None: S = set(G) # Default: All nodes
3     if len(S) == 1: return list(S) # Base case, single node
4     v = S.pop() # Reduction: Remove a node
5     seq = naive_topsort(G, S) # Recursion (assumption), n-1
6     min_i = 0
7     for i, u in enumerate(seq):
8         if v in G[u]: min_i = i + 1 # After all dependencies
9     seq.insert(min_i, v)
10    return seq
11
12 G = {'a': set('bf'), 'b': set('cdf'), 'c': set('d'), 'd': set('ef'), 'e': set('f'), 'f': set()}
13 print naive_topsort(G) # ['a', 'b', 'c', 'd', 'e', 'f']

```

上面这个算法是平方时间的，还有没有其他的reduction策略呢？前面的解法类似插入排序，既然又是reduce一个元素，很显然我们可以试试类似选择排序的策略，也就是说，我们找到一个节点，然后把它放在第一个位置上(后面有道练习题思考如果是放在最后一个位置上怎么办)，假设我们直接就是将这个节点去掉会怎样呢？如果剩下的图还是一个DAG的话我们就将原来的问题规约成了一个相似但是规模更小的问题对不对？但是问题是当我们选择哪个节点会使得剩下的图还是一个DAG呢？很显然，如果一个节点的入度为0，也就是说没有任何其他的节点依赖于它，那么它肯定可以直接安全地删除掉对不对？！

基于上面的思路就有了下面的解法，每次从图中删除一个入度为0的节点

Python

```

def topsort(G):
    count = dict((u, 0) for
1 def topsort(G):
2     count = dict((u, 0) for u in G) # The in-degree for each node
3     for u in G:
4         for v in G[u]:
5             count[v] += 1 # Count every in-edge
6     Q = [u for u in G if count[u] == 0] # Valid initial nodes
7     S = [] # The result
8     while Q: # While we have start nodes...
9         u = Q.pop() # Pick one
10        S.append(u) # Use it as first of the rest
11        for v in G[u]:
12            count[v] -= 1 # "Uncount" its out-edges
13            if count[v] == 0: # New valid start nodes?
14                Q.append(v) # Deal with them next
15    return S

```

[扩展知识：有意思的是，拓扑排序还和Python Method Resolution Order 有关，也就是用来确定某个方法是应该调用该实例的还是该实例的父类的还是继续往上调用祖先类的对应方法。对于单继承的语言这个很容易，顺着继承链一直往上找就行了，但是对于Python这类多重继承的语言则不简单，它需要更加复杂的策略，Python中使用了C3 Method Resolution Order，我不懂，[想要了解的可以查看on python docs](#)]

本章后面作者提到了一些其他的内容

1.Strong Assumptions

主要对于Induction，为了更加准确方便地从n-1递推到n，常常需要对问题做很强的假设。

2.Invariants and Correctness

循环不变式，这在算法导论上有详细介绍，循环不变式是用来证明某个算法是正确的一种方式，主要有下面三个步骤[这里和算法导论上介绍的不太一样，道理类似]：

- (1). Use induction to show that it is, in fact, true after each iteration.
- (2). Show that we'll get the correct answer if the algorithm terminates.
- (3). Show that the algorithm terminates.

3.Relaxation and Gradual Improvement

松弛技术是指某个算法使得当前得到的解有进一步的提升，越来越接近最优解(准确解)，这个技术非常实用，每次松弛可以看作是向最终解前进了“一步”，我们的目标自然是希望松弛的次数越少越好，关键就是要确定松弛的顺序(好的松弛顺序可以让我们直接朝着最优解前进，缩短算法运行时间)，后面要介绍的图中的Bellman-Ford算法、Dijkstra算法以及DAG图上的最短路径问题都是如此。

4.Reduction + Contraposition = Hardness Proof

规约是用于证明一个问题是否是一个很难的问题的好方式，假设我们能够将问题A规约至问题B，如果问题B很简单，那么问题A肯定也很简单。逆反一下我们就得到，如果问题A很难，那么问题B就也很难。比如，我们知道哈密顿回路问题是NP完全问题，要证明哈密顿路径问题也是NP完全问题，就可以将哈密顿回路问题规约为哈密顿路径问题。

[这里作者并没有过多的提到问题A规约至问题B的复杂度，算法导论中有提到，作者可能隐藏了规约的复杂度不大的含义，比如说多项式时间内能够完成，也就是下面的fast readuction]

“fast + fast = fast.” 的含义是：fast readuction + fast solution to B = fast solution to A

两条重要的规约经验：

- If you can (easily) reduce A to B, then B is at least as hard as A.
- If you want to show that X is hard and you know that Y is hard, reduce Y to X.

5.Problem Solving Advice

作者提供的解决一个问题的建议：

- (1) Make sure you really understand the problem.

搞明白你要解决的问题

What is the input? The output? What's the precise relationship between the two? Try to represent the problem instances as familiar structures, such as sequences or graphs. A direct, brute-force solution can sometimes help clarify exactly what the problem is.

- (2) Look for a reduction.

寻找一个规约方式

Can you transform the input so it works as input for another problem that you can solve? Can you transform the resulting output so that you can use it? Can you reduce an instance of size n to an instance of size $k < n$ and extend the recursive solution (inductive hypothesis) back to n ?

(3) Are there extra assumptions you can exploit?

还有其他的重要的假设条件吗，有时候我们如果只考虑该问题的特殊情况的话没准能够有所收获

Integers in a fixed value range can be sorted more efficiently than arbitrary values. Finding the shortest path in a DAG is easier than in an arbitrary graph, and using only non-negative edge weights is often easier than arbitrary edge weights.

问题4-18. 随机生成DAG图

Write a function for generating random DAGs. Write an automatic test that checks that topsort gives a valid orderings, using your DAG generator.

You could generate DAGs by, for example, randomly ordering the nodes, and add a random number of forward-pointing edges to each of them.

问题4-19. 修改拓扑排序

Redesign topsort so it selects the last node in each iteration, rather than the first.

This is quite similar to the original. You now have to maintain the out-degrees of the remaining nodes, and insert each node before the ones you have already found. (Remember not to insert anything in the beginning of a list, though; rather, append, and then reverse it at the end, to avoid a quadratic running time.)

[注意是使用append然后reverse，而不要使用insert]

1 赞 1 收藏 [评论](#)

Python算法：基础知识

原文出处：[hujiawei \(@五道口宅男\)](#)

本节主要介绍了三个内容：算法渐近运行时间的表示方法、六条算法性能评估的经验以及Python中树和图的实现方式。

1.计算模型

图灵机模型(Turing machine)： **A Turing machine is a simple (abstract) device that can read from, write to, and move along an infinitely long strip of paper.** The actual behavior of the machines varies. Each is a so-called finite state machine: it has a finite set of states (some of which indicate that it has finished), and every symbol it reads potentially triggers reading and/or writing and switching to a different state. You can think of this machinery as a set of rules. (“If I am in state 4 and see an X, I move one step to the left, write a Y, and switch to state 9.”)

RAM模型(random-access machine)： 标准的单核计算机，它大致有下面三个性质

- We don't have access to any form of concurrent execution; the machine simply executes one instruction after the other.

计算机不能并发执行而只是按照指令顺序依次执行指令。

- Standard, basic operations (such as arithmetic, comparisons, and memory access) all take constant (although possibly different) amounts of time. There are no more complicated basic operations (such as sorting).

基本的操作都是常数时间完成的，没有其他的复杂操作。

- One computer word (the size of a value that we can work with in constant time) is not unlimited but is big enough to address all the memory locations used to represent our problem, plus an extra percentage for our variables.

计算机的字长足够大以使得它能够访问所有的内存地址。

算法的本质： **An algorithm is a procedure, consisting of a finite set of steps (possibly including loops and conditionals) that solves a given problem in finite time.**

the notion of running time complexity (as described in the next section) is based on knowing how big a problem instance is, and that size is simply the amount of memory needed to encode it.

[算法的运行时间是基于问题的大小，这个大小是指问题的输入占用的内存空间大小]

2.算法渐近运行时间

主要介绍了大O符号、大Ω符号以及大Θ符号，这部分内容网上很多资料，大家也都知道了，此处略过，可以参考[wikipedia 大O符号](#)

算法导论介绍到，对于三个符号可以做如下理解： $O = \leq$, $\Omega = \geq$, $\Theta = =$

运行时间的三种特殊的情况：最优情况，最差情况，平均情况

几种常见的运行时间以及算法实例 [点击这里可以参考下wiki中的时间复杂度](#)

Complexity	Name	Examples, Comments
$\Theta(1)$	Constant	Hash table lookup and modification (see black box sidebar on dict).
$\Theta(\lg n)$	Logarithmic	Binary search (see Chapter 6). Logarithm base unimportant.
$\Theta(n)$	Linear	Iterating over a list.
$\Theta(n \lg n)$	Loglinear	Optimal sorting of arbitrary values (see Chapter 6). Same as $\Theta(\lg n!)$.
$\Theta(n^2)$	Quadratic	Comparing n objects to each other (see Chapter 3).
$\Theta(n^3)$	Cubic	Floyd and Warshall's algorithms (see Chapters 8 and 9).
$O(n^k)$	Polynomial	k nested for loops over n (if k is pos. integer). For any constant $k > 0$.
$\Omega(k^n)$	Exponential	Producing every subset of n items ($k = 2$; see Chapter 3). Any $k > 1$.
$\Theta(n!)$	Factorial	Producing every ordering of n values.

3. 算法性能评估的经验

(1) Tip 1: If possible, don't worry about it.

如果暴力求解也还行就算了吧，别去担心了

(2) Tip 2: For timing things, use timeit.

使用timeit模块对运行时间进行分析，在前面的[数据结构篇中第三部分数据结构](#)的list中已经介绍过了timeit模块，在使用的时候需要注意前面的运行不会影响后面的重复的运行(例如，分析排序算法运行时间，如果将前面已经排好序的序列传递给后面的重复运行是不行的)

Python

```
#timeit模块简单使用实  
例
```

```
1 #timeit模块简单使用实例  
2 timeit.timeit("x = sum(range(10))")
```

(3) Tip 3: To find bottlenecks, use a profiler.

使用cProfile模块来获取更多的关于运行情况的内容，从而可以发现问题的瓶颈，如果系统没有cProfile模块，可以使用profile模块代替，关于这两者的更多内容可以查看[Python standard library-Python Profilers](#)

Python

#cProfile模块简单使用 实例

```
1 #cProfile模块简单使用实例
2 import cProfile
3 import re
4 cProfile.run('re.compile("foo|bar")')
5
6 #运行结果：
7
8     194 function calls (189 primitive calls) in 0.000 seconds
9
10 Ordered by: standard name
11
12 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
13      1    0.000   0.000    0.000   0.000 <string>:1(<module>)
14      1    0.000   0.000    0.000   0.000 re.py:188(compile)
15      1    0.000   0.000    0.000   0.000 re.py:226(_compile)
16      1    0.000   0.000    0.000   0.000 sre_compile.py:178(_compile_charset)
17      1    0.000   0.000    0.000   0.000 sre_compile.py:207(_optimize_charset)
18 ...
```

(4) Tip 4: Plot your results.

画出算法性能结果图，如下图所示，可以使用的模块有matplotlib

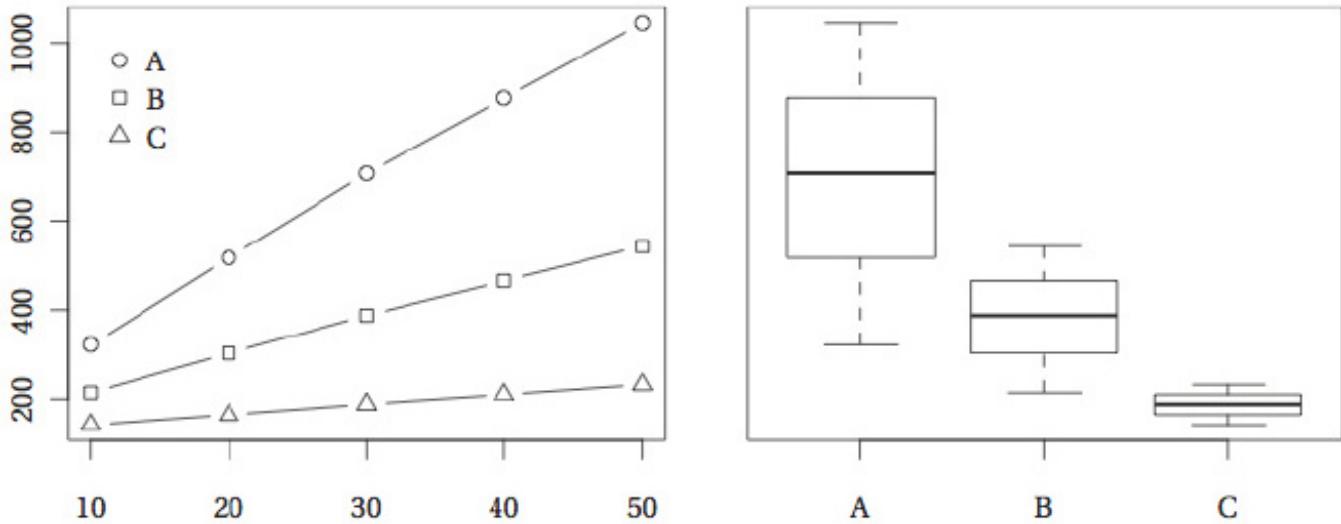


Figure 2-2. Visualizing running times for programs A, B, and C and problem sizes 10–50

(5) Tip 5: Be careful when drawing conclusions based on timing comparisons.

在对基于运行时间的比较而要下结论时需要小心

First, any differences you observe may be because of random variations.

首先，你观察到的差异可能是由于输入中的随机变化而引起的

Second, there are issues when comparing averages.

其次，比较算法的平均情况下的运行时间是存在问题的[这个我未理解，以下是作者的解释]

At the very least, you should stick to comparing averages of actual timings. A common practice to get more meaningful numbers when performing timing experiments is to normalize the running time of each program, dividing it by the running time of some standard, simple algorithm. This can indeed be useful but can in some cases make your results less than meaningful. See the paper “How not to lie with statistics: The correct way to summarize benchmark results” by Fleming and Wallace for a few pointers. For some other perspectives, you could read Bast and Weber’s “Don’t compare averages,” or the more recent paper by Citron et al., “The harmonic or geometric mean: does it really matter?”

Third, your conclusions may not generalize.

最后，你下的结论不要太过于宽泛

(6)Tip 6: Be careful when drawing conclusions about asymptotics from experiments.

在对从实验中得到关于渐近时间的信息下结论时需要小心，实验只是对于理论的一个支撑，可以通过实验来推翻一个渐近时间结果的假设，但是反过来一般不行 [以下是作者的解释]

If you want to say something conclusively about the asymptotic behavior of an algorithm, you need to analyze it, as described earlier in this chapter. Experiments can give you hints, but they are by their nature finite, and asymptotics deal with what happens for arbitrarily large data sizes. On the other hand, unless you’re working in theoretical computer science, the purpose of asymptotic analysis is to say something about the behavior of the algorithm when implemented and run on actual problem instances, meaning that experiments should be relevant.

4. 在Python中实现树和图

[Python中的dict和set]

Python中很多地方都使用了hash策略，在前面的[Python数据结构篇中的搜索部分](#)已经介绍了hash的内容。Python提供了hash函数，例如`hash("Hello, world!")`得到-943387004357456228 (结果不一定相同)。Python中的dict和set都使用了hash机制，所以平均情况下它们获取元素都是常数时间的。

(1)图的表示：最常用的两种表示方式是邻接表和邻接矩阵 [假设要表示的图如下]

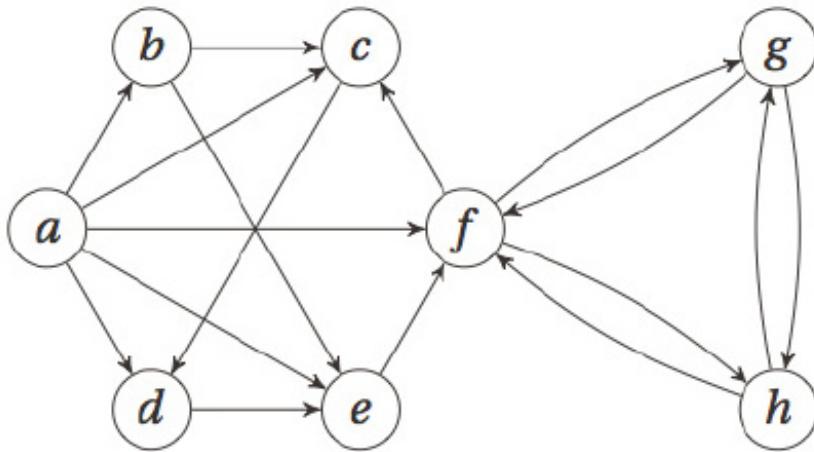


Figure 2-3. A sample graph used to illustrate various graph representations

邻接表 Adjacency Lists：因为历史原因，邻接表往往都是指链表list，但实际上也可以是其他的，例如在python中也可以是set或者dict，不同的表示方式有各自的优缺点，它们判断节点的连接关系和节点的度的方式甚至两个操作的性能都不太一样。

① adjacency lists 表示形式

Python

```
# A Straightforward
Adjacency List
```

```

1 # A Straightforward Adjacency List Representation
2 a, b, c, d, e, f, g, h = range(8)
3 N = [
4     [b, c, d, e, f],  # a
5     [c, e],          # b
6     [d],             # c
7     [e],             # d
8     [f],             # e
9     [c, g, h],       # f
10    [f, h],          # g
11    [f, g]           # h
12 ]
13
14 b in N[a] # Neighborhood membership -> True
15 len(N[f]) # Degree -> 3
```

② adjacency sets 表示形式

Python

```
# A Straightforward
Adjacency Set
```

```

1 # A Straightforward Adjacency Set Representation
2 a, b, c, d, e, f, g, h = range(8)
3 N = [
4     {b, c, d, e, f},  # a
5     {c, e},          # b
6     {d},             # c
7
8
9
10
11
12
13
14
15
```

```

7   {e},          # d
8   {f},          # e
9   {c, g, h},    # f
10  {f, h},      # g
11  {f, g}       # h
12 ]
13
14 b in N[a] # Neighborhood membership -> True
15 len(N[f]) # Degree -> 3

```

基本上和adjacency lists表示形式一样对吧？但是，对于list，判断一个元素是否存在是线性时间 $O(N(v))$ ，而在set中是常数时间 $O(1)$ ，所以对于稠密图使用adjacency sets要更加高效。

③ adjacency dicts 表示形式

Python

```

# A Straightforward
Adjacency Dict

```

```

1 # A Straightforward Adjacency Dict Representation
2 a, b, c, d, e, f, g, h = range(8)
3 N = [
4   {b:2, c:1, d:3, e:9, f:4},  # a
5   {c:4, e:3},                 # b
6   {d:8},                      # c
7   {e:7},                      # d
8   {f:5},                      # e
9   {c:2, g:2, h:2},            # f
10  {f:1, h:6},                # g
11  {f:9, g:8}                 # h
12 ]
13
14 b in N[a] # Neighborhood membership -> True
15 len(N[f]) # Degree -> 3
16 N[a][b] # Edge weight for (a, b) -> 2

```

这种情况下如果边是带权值的都没有问题！

除了上面三种方式外，还可以改变外层数据结构，上面三个都是list，其实也可以使用dict，例如下面的代码，此时节点是用字母表示的。在实际应用中，要根据问题选择最合适的表现形式。

Python

```

N = {
  'a': set('bcdef'),

```

```

1 N = {
2   'a': set('bcdef'),
3   'b': set('ce'),
4   'c': set('d'),
5   'd': set('e'),
6   'e': set('f'),
7   'f': set('cgh'),
8   'g': set('fh'),
9   'h': set('fg')
10 }

```

邻接矩阵 Adjacency Matrix

使用嵌套的list，用1和0表示点和点之间的连接关系，此时对于它们的连接性判断时间是常数，但是对于度的计算时间是线性的

Python

```
# An Adjacency Matrix, Implemented with Nested Lists
1 # An Adjacency Matrix, Implemented with Nested Lists
2 a, b, c, d, e, f, g, h = range(8)
3 N = [[0,1,1,1,1,1,0,0], # a
4      [0,0,1,0,1,0,0,0], # b
5      [0,0,0,1,0,0,0,0], # c
6      [0,0,0,0,1,0,0,0], # d
7      [0,0,0,0,0,1,0,0], # e
8      [0,0,1,0,0,0,1,1], # f
9      [0,0,0,0,0,1,0,1], # g
10     [0,0,0,0,0,1,1,0]] # h
11
12 N[a][b] # Neighborhood membership -> 1
13 sum(N[f]) # Degree -> 3
```

如果边带有权值，也可以使用权值代替1，用inf代替0

Python

```
a, b, c, d, e, f, g, h =
range(8)
1 a, b, c, d, e, f, g, h = range(8)
2 _ = float('inf')
3
4 W = [[0,2,1,3,9,4,_,_], # a
5      [_,0,4,_,_3,_,_], # b
6      [_,_0,8,_,_,_], # c
7      [_,_,_0,7,_,_], # d
8      [_,_,_,_0,5,_], # e
9      [_,_2,_,_0,2,2], # f
10     [_,_,_,_1,0,6], # g
11     [_,_,_,_9,8,0]] # h
12
13 W[a][b] < inf # Neighborhood membership
14 sum(1 for w in W[a] if w < inf) - 1 # Degree
```

NumPy: 这里作者提到了一个最常用的数值计算模块NumPy，它包含了很多与多维数组计算有关的函数。我可能会在以后的机器学习中详细学习它的使用，到时候可能会写篇文章介绍它的使用

(2)树的表示 [假设要表示下面的树]

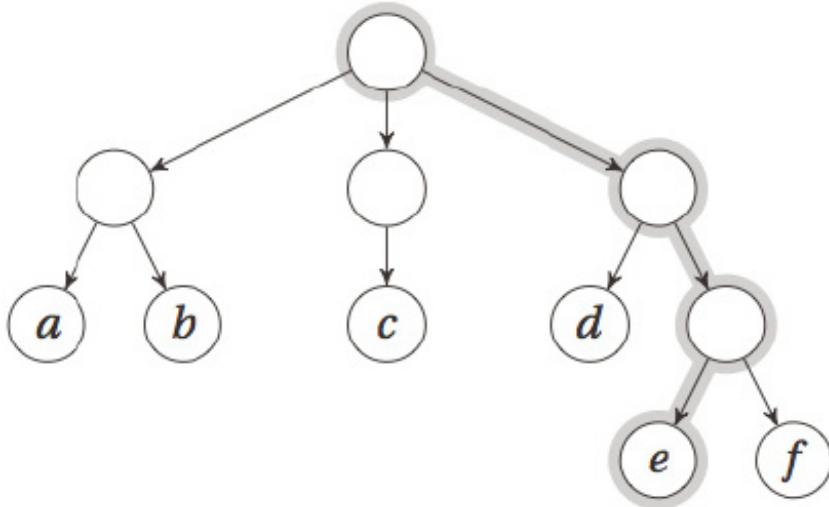


Figure 2-4. A sample tree with a highlighted path from the root to a leaf

树是一种特殊的图，所以可以使用图的表示方法，但是因为树的特殊性，其实有其他更好的表示方法，最简单的就是直接用一个list即可，缺点也很明显，可读性太差了，相当不直观

Python

```

T = [["a", "b"], ["c"], ["d",      █
      ["e", "f"]]]
```

- 1 T = [["a", "b"], ["c"], ["d", ["e", "f"]]]
- 2 T[2][1][0] # 'e'

很多时候我们都能够肯定树中节点的孩子节点个数最多有多少个(比如二叉树，三叉树等等)，所以比较方便的实现方式就是使用类class

Python

```

# A Binary Tree Class █
二叉树实例
```

- 1 # A Binary Tree Class 二叉树实例
- 2 class Tree:
- 3 def __init__(self, left, right):
- 4 self.left = left
- 5 self.right = right
- 6
- 7 t = Tree(Tree("a", "b"), Tree("c", "d"))
- 8 t.right.left # 'c'

上面的实现方式的子节点都是孩子节点，但是还有一种很常用的树的表示方式，那就是“左孩子，右兄弟”表示形式，它就适用于孩子节点数目不确定的情况

Python

```

# 左孩子，右兄弟 表示 █
方式
```

- 1 # 左孩子，右兄弟 表示方式
- 2 class Tree:

```

3 def __init__(self, kids, next=None):
4     self.kids = self.val = kids
5     self.next = next
6 return Tree
7
8 t = Tree(Tree("a", Tree("b", Tree("c", Tree("d")))))
9 t.kids.next.next.val # 'c'

```

[**Bunch Pattern**]: 有意思的是，上面的实现方式使用了Python中一种常用的设计模式，叫做Bunch Pattern，貌似来自经典书籍Python Cookbook，原书介绍如下：

[因为这个不太好理解和翻译，还是原文比较有味，后期等我深刻理解了我可能会详细介绍它]

When prototyping (or even finalizing) data structures such as trees, it can be useful to have a flexible class that will allow you to specify arbitrary attributes in the constructor. In these cases, the “Bunch” pattern (named by Alex Martelli in the Python Cookbook) can come in handy. There are many ways of implementing it, but the gist of it is the following:

Python

```

class Bunch(dict):
    def __init__(self,
1 class Bunch(dict):
2     def __init__(self, *args, **kwds):
3         super(Bunch, self).__init__(*args, **kwds)
4         self.__dict__ = self
5 return Bunch

```

There are several useful aspects to this pattern. First, it lets you create and set arbitrary attributes by supplying them as command-line arguments:

Python

```

>>> x =
Bunch(name="Jayne")
1 >>> x = Bunch(name="Jayne Cobb", position="Public Relations")
2 >>> x.name
3 'Jayne Cobb'

```

Second, by subclassing dict, you get lots of functionality for free, such as iterating over the keys/attributes or easily checking whether an attribute is present. Here's an example:

Python

```

>>> T = Bunch
>>> t = T(left=T(left="a",
1   >>> T = Bunch
2   >>> t = T(left=T(left="a", right="b"), right=T(left="c")))
3   >>> t.left
4   {'right': 'b', 'left': 'a'}
5   >>> t.left.right
6   'b'
7   >>> t['left']['right']
8   'b'

```

```
9 >>> "left" in t.right
10 True
11 >>> "right" in t.right
12 False
```

This pattern isn't useful only when building trees, of course. You could use it for any situation where you'd want a flexible object whose attributes you could set in the constructor.

[与图有关的python模块]:

- NetworkX: <http://networkx.lanl.gov>
 - python-graph: <http://code.google.com/p/python-graph>
 - Graphine: <http://gitorious.org/projects/graphine/pages/Home>
 - Pygr: a graph database <http://bioinfo.mbi.ucla.edu/pygr>
 - Gato: a graph animation toolbox <http://gato.sourceforge.net>
 - PADS: a collection of graph algorithms <http://www.ics.uci.edu/~eppstein/PADS>
-

5. Python编程中的一些细节

In general, the more important your program, the more you should mistrust such black boxes and seek to find out what's going on under the cover.

作者在这里提到，如果你的程序越是重要的话，你就越是需要明白你所使用的数据结构的内部实现，甚至有些时候你要自己重新实现它。

(1) Hidden Squares 隐藏的平方运行时间

有些情况下我们可能没有注意到我们的操作是非常不高效的，例如下面的代码，如果是判断某个元素是否在list中运行时间是线性的，如果是使用set，判断某个元素是否存在只需要常数时间，所以如果我们需要判断很多元素是否存在的话，使用set的性能会更加高效。

Python

```
from random import randrange
1 from random import randrange
2 L = [randrange(10000) for i in range(1000)]
3 42 in L # False
4 S = set(L)
5 42 in S #False
```

(2) The Trouble with Floats 精度带来的烦恼

现有的计算机系统都是不能精确表达小数的！[该部分内容可以阅读与计算机组成原理相关的书籍了解计算机的浮点数系统]在python中，浮点数可能带来很多的烦恼，例如，运行下面的实例，本应该

是相等，但是却返回False。

Python

```
sum(0.1 for i in range(10))  
== 1.0 # False  
  
1 sum(0.1 for i in range(10)) == 1.0 # False
```

永远不要使用小数比较结果来作为两者相等的判断依据！你最多只能判断两个浮点数在有限位数上是相等的，也就是近似相等了。

Python

```
def almost_equal(x, y,  
places=7):  
  
1 def almost_equal(x, y, places=7):  
2     return round(abs(x-y), places) == 0  
3  
4 almost_equal(sum(0.1 for i in range(10)), 1.0) # True
```

除此之外，可以使用一些有用第三方模块，例如decimal，在需要处理金融数据的时候很有帮助

Python

```
from decimal import *  
sum(Decimal("0.1") for i  
  
1 from decimal import *  
2 sum(Decimal("0.1") for i in range(10)) == Decimal("1.0") # Ture
```

还有一个有用的Sage模块，如下所示，它可以进行数学的符号运算得到准确值，如果需要也可以得到近似的浮点数解。[Sage的官方网址](#)

Python

```
sage: 3/5 * 11/7 +  
sqrt(5239)  
  
1 sage: 3/5 * 11/7 + sqrt(5239)  
2 13*sqrt(31) + 33/35
```

更多和Python中的浮点数有关的内容可以查看[Floating Point Arithmetic: Issues and Limitations](#)

问题2-12. (图的表示)

Consider the following graph representation: you use a dictionary and let each key be a pair (tuple) of two nodes, with the corresponding value set to the edge weight. For example $W[u, v] = 42$. What would be the advantages and disadvantages of this representation? Could you supplement it to mitigate the downsides?

The advantages and disadvantages depend on what you're using it for. It works well for looking up edge weights efficiently but less well for iterating over the graph's nodes or a node's neighbors, for example. You could improve that part by using some extra structures (for example, a global list of

nodes, if that's what you need or a simple adjacency list structure, if that's required).

1 赞 2 收藏 [评论](#)

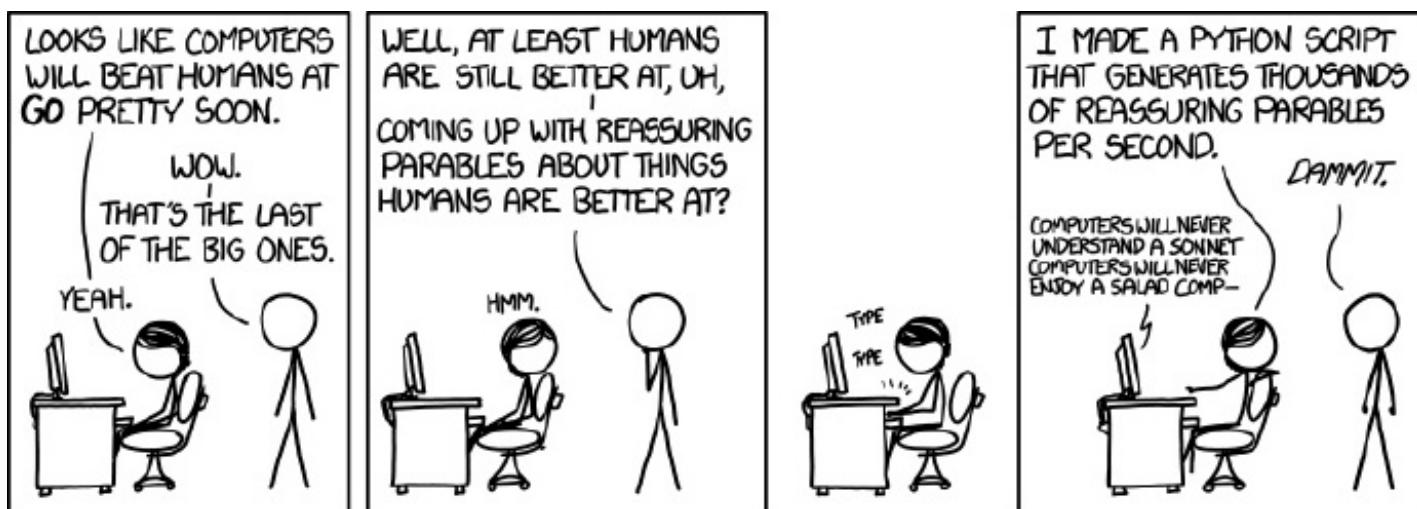
ML/NLP入门教程Python版（第一部分：文本处理）

本文由[伯乐在线 - 霉霉](#)翻译, [Daetalus](#)校稿。未经许可, 禁止转载!

英文出处：www.thoughtly.co。欢迎加入翻译组。

第一部分：文本处理

欢迎来到机器学习和自然语言处理原型编码教程系列的第一部分。Thoughtly正在制作一个着重于理解机器学习基础的系列教程,着重关注于在自然语言处理中的应用。



这一系列教程的目标是提供有据可查的可用代码，附加留言部分的深入探讨。代码将被放到GitHub上，在一个开放的许可证下，允许你任意修改或使用——不必署名（注明来源）。这里的代码为了明白起见以牺牲性能为代价写的比较冗长。如果你有大量的数据要处理，这些工具的可扩展性很可能无法达到完成你目的的要求。幸运的是，我们正在计划通过研究此处讨论的算法在当下最新的实现，来更好地对这个系列进行深入探索。这些内容都是黑盒子，是我们在初始系列中有意避免（到实用的程度）提到的内容。我们相信，在能使用这些黑盒子之前，在机器学习方面打下一个坚实的基础是至关重要的。

第一部分的重点是如何从文本语料库提取出信息来。我们有意用介绍性的水平来开始教程，但是它涉及到很多不同的技巧和测量标准，这些方法都会在之后应用到更深入的机器学习任务上。

文本提取

下文介绍的以及此处代码中用到的工具，都假设我们将所选的语料当作一袋单词。这是你在处理文本文档的时候常常会看到的一个基本概念。将语料当作一袋单词是将文档向量化中的一个典型步骤，以供机器学习算法进一步处理。把文档转换成可处理向量通常还需要采取一些额外步骤，我们将在后面的课程中对此进行讨论。本课程中介绍的概念和工具将作为后面工具的构建模块。也许更重要的是，这些工具可以帮助你通过快速检查一个文本语料库，从而对它所包含的内容有一个基本的了解。

本课程中我们所研究的代码及示例都是使用Python实现的。这些代码能够从NLTK(Python的自然语言工具包)所提供的不同的文本语料库中提取数据。这是个包括了ABC新闻的文字、圣经的创世纪、从

古滕堡计划中选取的部分文本、总统就职演说、国情咨文和从网络上截取的部分文本所组成的语料库。另外，用户还能从他们自己提供的语料库来提取文本。从NLTK导入的代码并不是特别有趣，但我想指出的是，要从NLTK文本语料库中提取数据是非常简单方便的。

Python

```
def load_text_corpus(args):
    1 def load_text_corpus(args):
    2
    3     if args["abc"]:
    4         logging.debug("Loading the ABC corpus.")
    5         name = "ABC"
    6         words = nltk.corpus.abc.words()
    7
    8     elif args["genesis"]:
    9         logging.debug("Loading the ABC corpus.")
   10        name = "Genesis"
   11        words = nltk.corpus.genesis.words()
   12
   13    elif args["gutenberg"]:
   14        logging.debug("Loading the Gutenberg corpus.")
   15        name = "Gutenberg"
   16        words = nltk.corpus.gutenberg.words()
   17
   18    elif args["inaugural"]:
   19        logging.debug("Loading the Inaugural Address corpus.")
   20        name = "Inaugural"
   21        words = nltk.corpus.inaugural.words()
   22
   23    elif args["stateUnion"]:
   24        logging.debug("Loading the State of the Union corpus.")
   25        name = "Union"
   26        words = nltk.corpus.state_union.words()
   27
   28    elif args["webtext"]:
   29        logging.debug("Loading the webtext corpus.")
   30        name = "Web"
   31        words = nltk.corpus.webtext.words()
   32
   33    elif args["custom"] != None:
   34        logging.debug("Loading a custom corpus from " + args["custom"])
   35        name = "Custom"
   36        words = load_custom_corpus(args["custom"])
   37    else:
   38        words = ""
   39        name = "None"
   40
   41    logging.debug("Read " + str(len(words)) + " words: " + str(words[0:20]))
   42
   43    return words, name
```

上面的大部分代码只是日志。有意思的部分在357行、362行、367行等。基于用户选择，每部分加载不同的语料库。NLTK对从现有语料库中提取文本提供了一些非常便利的方法。这包括一些简单的、纯文本的语料库，也包括一些已经用各种方式被标记过的语料库——语料库中的每个文档可能被标记过类别或是语料库中有的语音已被加过标签，如此等等。在本课程中，我们对NLTK的使用仅限于语料库的导入、词汇的切分，以及我们下面将讨论两个操作，词根和词形还原。虽然不会总是如此，但现在为止足够我们需要的所有功能。值得注意的是，您还可以在脚本中使用`-custom`参数导入自定

义语料库。这应该是含有.txt文件的文件夹。该文件夹是递归读入的，所以含有.txt文件的子文件夹也能被处理。

词汇切分

词汇切分是切分语料库，使之变成各个独立部分——通常指单词，的行为。我们这样做是因为大多数ML算法无法处理任意长的文本字符串。相反，他们会假设你已经分割你的语料库为单独的，算法可处理的词块（token）。虽然我们将在后面的课程详细讨论这个话题，算法不一定限于一次只处理一个词块（token）。事实上，许多算法只在处理短序列（n-grams）时有用。本课程中我们将情况限定于一序列（1-grams），或者叫，单序列（unigram）。

对文本语料库做词汇切分的最简单的方法就是仅基于空白字符。这种方法确实非常简单，但它也有缺点。例如，它会导致位于句尾的文本包含有句尾标点符号，而一般不需要这样。在另一方面，类似can't和e.g.这样带有词内标点的单词就没法被正确提取出来了。我们可以添加一步操作来删除所有非字母数字的字符。这将解决句尾标点符号的问题，同时也能将can't和e.g.这样的单词提取出来，尽管是以丢掉了他们的标点符号的方式被提取出来的。然而，这也引入了一个新的问题。对于某些应用，我们还是希望保留标点符号。在创建语言模型的时候，句尾标点能区分一个单词是否是结尾单词，从这方面来说，额外的标点信息是有价值的。

对于这个任务，我们要将一些标点符号（句号）作为一个词，使用NLTK word_tokenizer（它是基于TreebankWordTokenizer来实现的）来做词汇切分。这个分词器有很多针对各式各样的词汇做切分的规则。举例来说，“can't”这样的缩写实际上被分成了两个词(token) – ca和n't。有趣的是，这意味着我们最后会得到ca这样的词，它理想地匹配了can（在某些任务中）。这样的错误匹配是这种符号化算法带来的不幸后果。NLTK支持多种分词器。这是一个及其冗长的文件，<http://www.nltk.org/api/nltk.tokenize.html>，但在里面可以找到它所支持的分词器的细节。

词干提取和词形还原

一旦取到了文字我们就可以开始处理它。脚本提供了许多简单的工具，它们会帮助我们查看我们所选择的内容。之后我们会深入谈到这些工具。首先，让我们思考一下该用什么方法来操作我们取到的文本。通常我们需要为ML算法提供从语料库提取的原始文本词汇（单词）。在其他情况下，将这些单词转成原始内容的各种变形也是有道理的。

具体来说，我们经常要将原始单词截断到它的词根。那么，什么是一个词根呢？英语单词有从原始单词延伸出的通用后缀。就拿单词“run”为例。有很多的扩展它的词 – “runner”，“runs”，“running”等，即对基本定义的进一步阐述。词干提取是从“runner”，“runs”以及“running”中去除所有和“run”不一致的部分的过程。请注意，在上述列表中不包含“ran”——后面我们再对此进行阐述。下面是一个被提取词干的句子的具体实例。

Python

```
stem(Jim is running to work.) => Jim is run
```

1 stem(Jim is running to work.) => Jim is run to work.

我们已经丢失了“吉姆在跑步”这个信息，尽管此处的上下文隐含的所有其他信息都说不通。我们不可能完全扭转这一点——我们可以猜测那里曾经是什么词，但我们很可能会弄错。

此处提供的代码可以让你对你的语料库进行词干提取。实际的词干提取是微不足道的，因为我们会使用NLTK来进行这部分工作。我们只需通过输入数组迭代，并返回使用NLTK Porter Stemmer所得到的各种提取后的词干变体。有许多不同的词干分析器可供选择，还包括非英语语言的选项。Porter Stemmer常用于英语。

Python

```
def  
stem_words_array(word)  
  
1 def stem_words_array(words_array):  
2     stemmer = nltk.PorterStemmer()  
3     stemmed_words_array = [];  
4     for word in words_array:  
5         stem = stemmer.stem(word);  
6         stemmed_words_array.append(stem);  
7     return stemmed_words_array;
```

词形还原类似于词干提取，但又有着重要的区别。与使用一系列简单的规则将一个单词截断成它的词根不同，词形还原尝试对输入的单词确定一个恰当的词根。本质上，词形还原试图找到一个单词的字典项，也称为单词的基本形(base term)。为了使这种查找能正确的工作，词形还原器必须知道您寻找的这个词在句子中的词性。生成语料库的词条与词干提取的代码基本上是相同的（尽管这段代码有上文略为提及的缺点，我们将在下面进一步对此进行讨论）。这里我们用了WordNetLemmatizer，它使用WordNet的数据库作为其查询指定词条的字典。

Python

```
def  
lemmatize_words_array(word)  
  
1 def lemmatize_words_array(words_array):  
2     lemmatizer = nltk.stem.WordNetLemmatizer()  
3     lemmatized_words_array = [];  
4     for word in words_array:  
5         lemma = lemmatizer.lemmatize(word)  
6         lemmatized_words_array.append(lemma)  
7     return lemmatized_words_array;
```

正如上文所述，词形还原知道单词的词性。NLTK WordNetLemmatizer天真地假设，所有传入的单词都是名词。这种假设意味着你必须告诉词形还原器要传递的词不是一个名词，否则它会错误地将其视为一个名词。这个行为，加上对未知的单词（特别是当它混在一段文本中的时候）不做任何处理直接输出的行为，使得词形还原器处理效果很差。举例来说，如果让词形还原器处理“ran”这个词，在不指出“ran”属于一段文本的情况下，它将直接输出“ran”。它不知道的作为名词的“ran”，因为很明显“ran”不是一个名词。但是，如果你正确地指出“ran”是动词，那么词形还原器就能输出“run”。与相对，此处词干分析器就会输出“ran”。因此，如果我们要有效地利用词形还原器，我们也必须付出在源代码中对词性进行标注的代价，我们将在后面的课程中对词性标注的部分进行讨论。标记单词词性的额外成本也是词形还原器不像词干分析器那样应用广泛的原因之一——所添加的功能抵不上所花的成本。

词汇量

现在，使用词干提取或词形还原的方法，我们已经拉取了一个语料库并且(视情况)对它做了变形，终于可以开始查看它的内容了。下面不是一个详尽的清单，但作为审查文本的技术参考。有些是立刻会

用到的，其他则会在以后讨论到。

第一项测量是最简单的——词汇计数。这个指标是语料库内所有唯一字的计数。正如你所期望的，代码很容易实现。唯一一个你之后还会再遇到的技巧，是我们决定使用Python里dictionary的唯一性。即任一字典的条目在字典中不能出现超过一次。

Python

```
def collect_unique_terms(c)
1 def collect_unique_terms(corpus):
2     unique_vocabulary = {}
3     for term in corpus:
4         unique_vocabulary[term] = 1;
5     return unique_vocabulary;
```

这种方法可以让我们对我们的数据有所认知。思考我们使用词干提取及词形还原来考察ABC语料库后的如下输出。

首先是原始语料文本：

Python

```
&gt; python words.py -vv -abc -s -vs
&gt; python words.py -vv -abc -s -vs
Loading the ABC corpus.
1 Read 766811 words: [u'PM', u'denies', u'knowledge', u'of', u'AWB', u'kickbacks', u'The', u'Prime', u'Minister', u'has', u'denied', u'he', u'knew', u'AWB', u'was', u'paying', u'kickbacks', u'to', u'Iraq', u'despite']
The corpus contains 766811 elements after processing
The corpus has a total vocabulary of 31885 unique tokens.
```

其次是词形还原后的语料库：

Python

```
&gt; python words.py -vv -abc -l -vs
&gt; python words.py -vv -abc -l -vs
Loading the ABC corpus.
1 Read 766811 words: [u'PM', u'denies', u'knowledge', u'of', u'AWB', u'kickbacks', u'The', u'Prime', u'Minister', u'has', u'denied', u'he', u'knew', u'AWB', u'was', u'paying', u'kickbacks', u'to', u'Iraq', u'despite']
The corpus contains 766811 elements after processing
The corpus has a total vocabulary of 28699 unique tokens.
```

最后，是词干提取后的语料库：

Python

```
&gt; python words.py -v  
vv -abc -vs
```

```
&gt; python words.py -vv -abc -vs
```

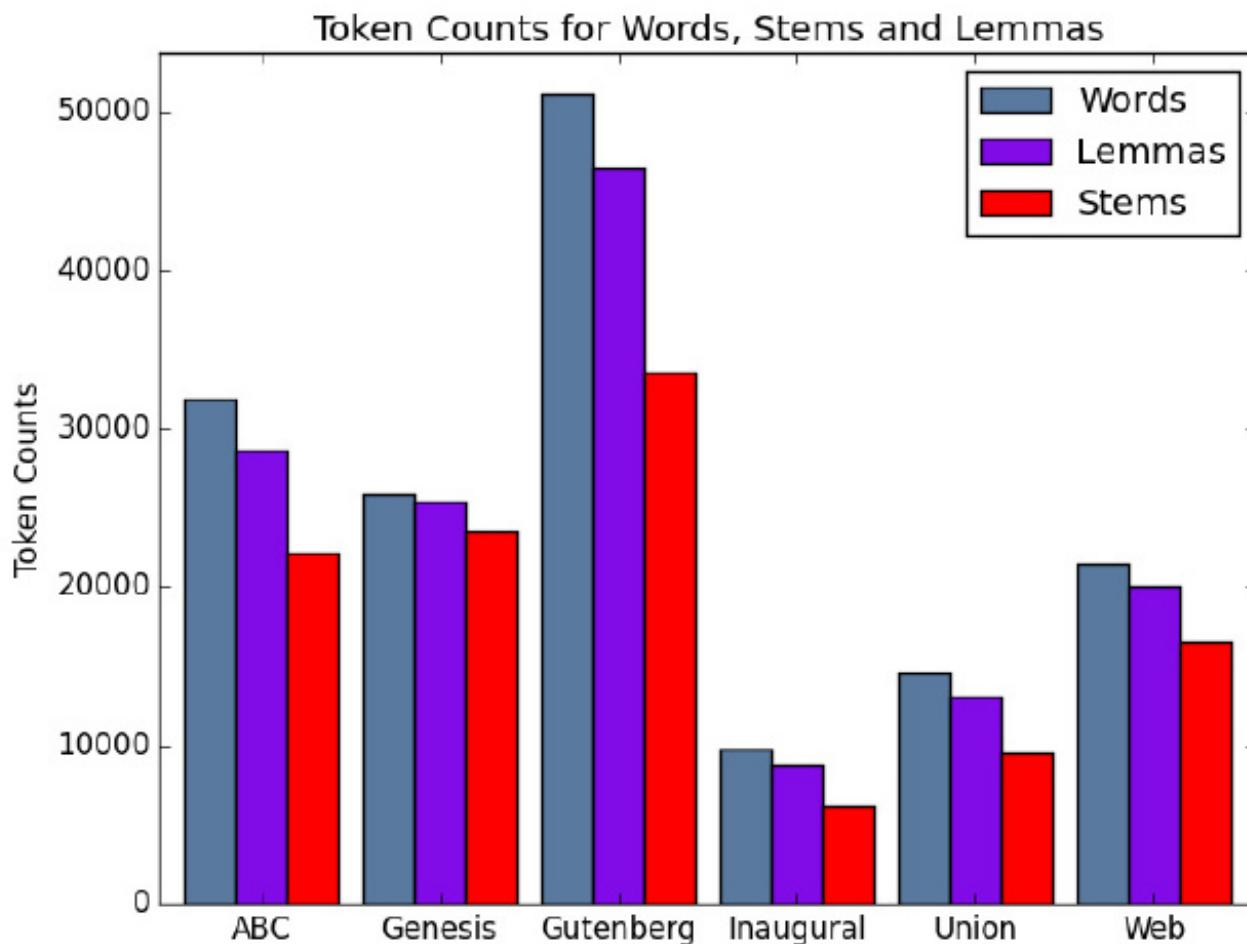
Loading the ABC corpus.

```
1 Read 766811 words: [u'PM', u'denies', u'knowledge', u'of',  
2 u'AWB', u'kickbacks', u'The', u'Prime', u'Minister',  
3 u'has', u'denied', u'he', u'knew', u'AWB',  
4 u'was', u'paying', u'kickbacks', u'to', u'Iraq',  
5 u'despite']
```

The corpus contains 766811 elements after processing

The corpus has a total vocabulary of 22162 unique tokens.

可以看到，从原始数据到词形还原到词干提取后，语料库中唯一字计数值总体在减少，从31K至28K到22K。这个模式重复于每个语料库。在每个实例中，原始语料库的字数统计大于词干提取后的，而词干提取后的字数统计则大于词形还原后的。



上面的图表是使用我们共享工程的Python代码生成。它对非定制语料库列表进行遍历，并分别计算原始、词干提取后、词形还原后的唯一字数量。你可以用命令行重现这个图表。你还可以得到一份同样内容的文本转储。

Python

```
&gt; python words.py -v  
--stemVsLemma
```

```
1 &gt; python words.py -v --stemVsLemma
2 2015-02-02 19:49:22,255 (INFO): Corpora: [&#039;ABC&#039;, &#039;Genesis&#039;, &#039;Gutenberg&#039;,
3 &#039;Inaugural&#039;, &#039;Union&#039;, &#039;Web&#039;]
4 2015-02-02 19:49:22,255 (INFO): Word Counts: [31885, 25841, 51156, 9754, 14591, 21538]
5 2015-02-02 19:49:22,255 (INFO): Lemmatized Word Counts: [28699, 25444, 46456, 8763, 13111, 20056]
6 2015-02-02 19:49:22,255 (INFO): Stemmed Word Counts: [22162, 23542, 33521, 6135, 9533, 16599]
7 2015-02-02 19:49:22,466 (INFO): The corpus contains 0 elements after processing
```

词项存在

加入一点复杂性，我们接下来看看如何输出上文所指出的词项。我们生成一个CSV文件，它包含了出现在语料中的每个唯一字。它使用了上文中collect_unique_terms这个方法，只是不同于仅仅简单地输出唯一字计数，它通过遍历返回的字典会打印出每个键值。

Python

```
def output_corpus_terms(co
1 def output_corpus_terms(corpus, unique_vocabulary=None):
2     if unique_vocabulary is None:
3         unique_vocabulary = collect_unique_terms(corpus)
4     output_csv_file = open_csv_file("corpus_terms.csv", ["Term"])
5     for term in unique_vocabulary:
6         logging.debug(term)
7         output_csv_file.writerow([term])
```

虽然在仅仅输出一个单词的情况下，它可能看起来意义非常有限，但也有比统计每个词的总计数更好的算法（我们接下来将要讨论）。正如对微博(tweets)的情感分析一样——在处理较短的文字序列时，我们更倾向于选择它。

您可以在命令行中使用所提供的代码生成CSV。

Python

```
python words.py -v --
termPresence --
1 python words.py -v --termPresence --gutenberg
2 2015-02-02 20:07:49,623 (INFO): The corpus contains 2621613 elements after processing
```

词频

词频是词项存在的延伸。不是简单地指出一个词的存在，而是在确定词频的时候，我们更关心语料库中的每个词所出现的实例个数。计算这个的代码与确定词项存在的代码是非常相似的。

Python

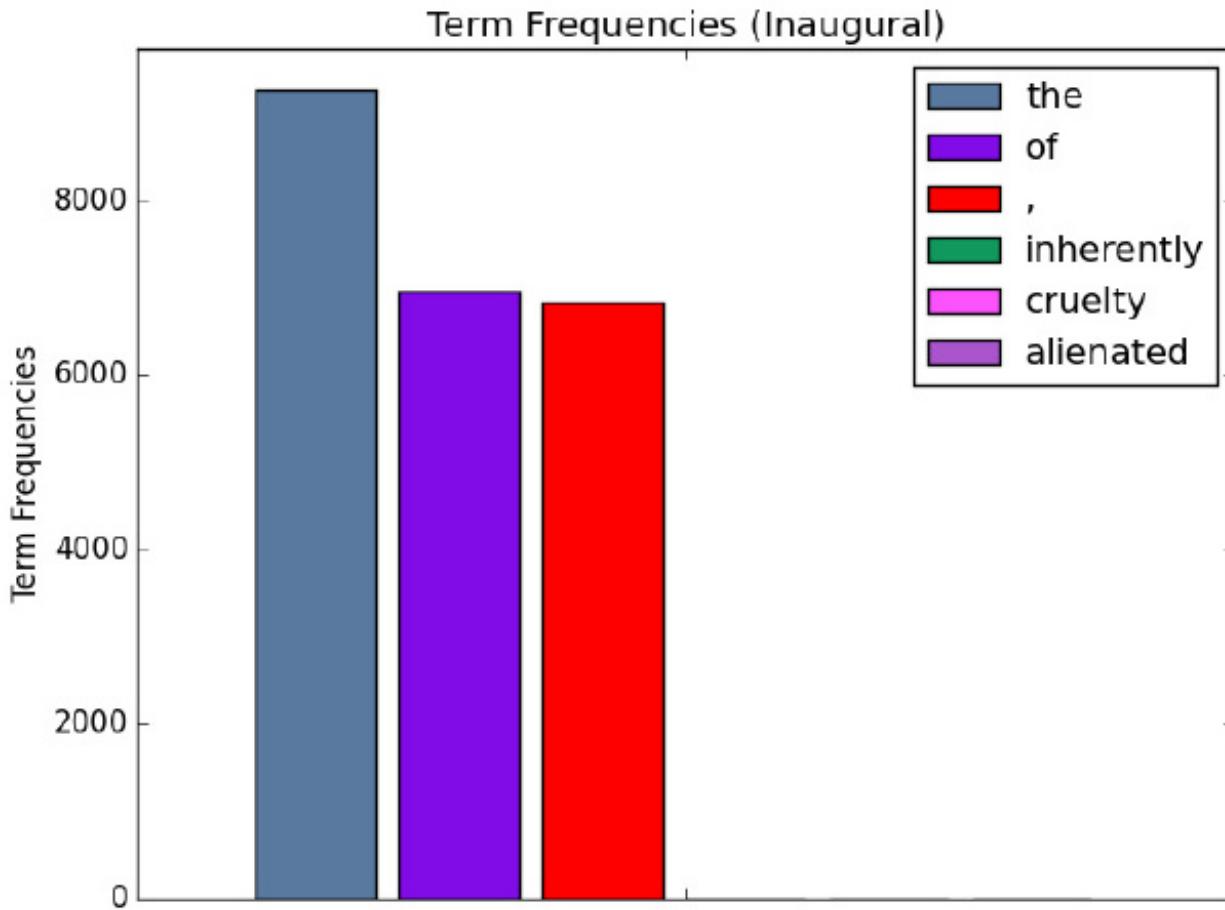
```
def collect_and_output_cor
1 def collect_and_output_corpus_term_frequencies(corpus, corpus_name):
2     term_frequencies = collect_term_counts(corpus)
3
4     output_csv_file = open_csv_file("term_frequencies.csv", ["Term", "Frequency"])
5
```

```

6     unsorted_array = [[key,value] for key, value in term_frequencies.iteritems()]
7     sorted_array = sorted(unsorted_array, key=lambda term_frequency: term_frequency[1], reverse=True)
8
9     for term, frequency in sorted_array:
10        output_csv_file.writerow([term] + [frequency])
11
12 # output a bar chart illustrating the above
13 chart_term_frequencies("term_frequencies.png",
14                         "Term Frequencies (" + corpus_name + ")",
15                         "Term Frequencies",
16                         sorted_array, [0, 1, 2, -3, -2, -1])
17
18 return term_frequencies

```

词频，或者我们将在下一节中看到的基于它的变形，是机器学习的矢量化过程中常见的主要组成部分。一般来说，ML算法需要一组能代表需要判定的单个样本的特征集。但是，文本并不能自动地适应这种模式。要迫使它去适应，我们不能考虑文本本身，而是要看文字实例的数量。词频是将文本域映射到ML友好的实数域一个简单的方法。



您可以在命令行中使用提供的源代码生成包含所有词条和其频率的有序列表的CSV文件以及上面的图表。

Python

```
&gt; python words.py -v  
--termFrequency --
```

```
1 &gt; python words.py -v --termFrequency --genesis  
2 2015-02-02 20:27:45,298 (INFO): The corpus contains 315268 elements after processing
```

记录标准化词频

该图显示了语料库中三个最常见词条和最不常见词条的原始频率。这里提供的代码可以为用户选择的语料库生成上述图表。此外。运行该脚本还会生成一个名为term_frequencies.csv的文件，它能让用户看到一个包含文档中的所有唯一字及其相应词频的电子表格。使用美国总统就职演说来生成就是：

Python

```
&gt; python words.py -  
vv --termFrequency --
```

```
&gt; python words.py -vv --termFrequency --inaugural  
2015-02-02 20:04:09,345 (DEBUG): Loading the Inaugural Address corpus.  
1 2015-02-02 20:04:09,464 (DEBUG): Read 145735 words: [u'Fellow', u'&#039;-',  
2 u'Citizens', u'of', u'the', u'Senate', u'and',  
3 u'of', u'the', u'House', u'of', u'Representatives',  
4 u'&#039;;', u'Among', u'the', u'vicissitudes', u'incident',  
u'to', u'life', u'no']  
2015-02-02 20:04:09,465 (INFO): The corpus contains 145735 elements after processing
```

机器学习算法在特征值没有规范到相似的尺度内的时候常常就不工作了。在计算词频的情况下，相对于罕见的单词来说，常用字可能会出现得非常频繁。这将造成这些组完全不同的频率字之间的显著歪斜。即使算法能处理歪斜的特征量，如果你认为出现率10倍以上的词条就重要10倍以上的话，那么特定的任务可能无法正常工作。收缩特征值大小，同时还允许收缩后的特征值随着原始数据的增长而增长，一种常见的方法是取该特征值的对数。在此实例中，我们使用下面的方程来对词频数据进行归一：

Python

```
LogNormalizedTF = 1 +  
log10(TermFrequency)
```

```
1 LogNormalizedTF = 1 + log10(TermFrequency)
```

使用对数以10为底意味着对于每10倍增加的词频，我们将看到对数归一后的一个数量点的增长。我们将对数归一后的词频初始化为1，这样一来，对于词频是0的词来说值刚好也是1。用来计算归一化后的词频的代码非常简单，并依赖于前一节中提到的频率采集器。请注意，此代码同时转储输出到一个CSV文件。上文其他的示例代码没有这一步，因为它们其实是那些最终去转储CSV方法的辅助方法。这段代码恰好是在转储到CSV前做了少量工作（计算对数归一化）。

Python

```
if term_frequencies is  
None:
```

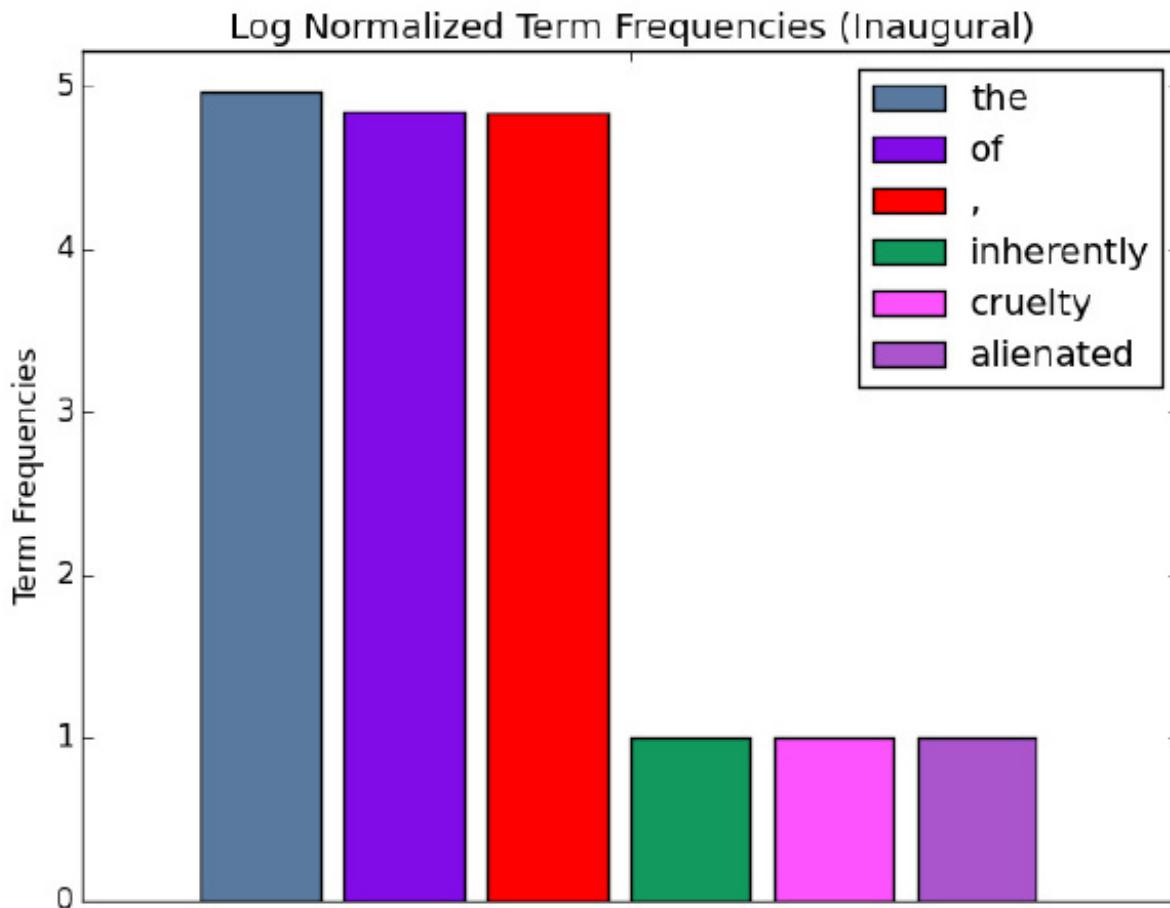
```
1 if term_frequencies is None:  
2     term_frequencies = collect_term_counts(corpus)  
3  
4 output_csv_file = open_csv_file("normalized_term_frequencies.csv", ["Term", "Log Normalized TF"])  
5
```

```

6 unsorted_array = []
7
8 for term, frequency in term_frequencies.iteritems():
9     normalized_term_frequency = (1 + math.log(frequency, 10))
10    unsorted_array.append([term, normalized_term_frequency])
11    output_csv_file.writerow([term] + [normalized_term_frequency])
12
13 sorted_array = sorted(unsorted_array, key=lambda term_frequency: term_frequency[1], reverse=True)
14
15 # output a bar chart illustrating the above
16 chart_term_frequencies("normalized_term_frequencies.png",
17                         "Log Normalized Term Frequencies (" + corpus_name + ")",
18                         "Term Frequencies",
19                         sorted_array, [0, 1, 2, -3, -2, -1])
20
21 return term_frequencies

```

下面的图表和上节“词频”中使用的是相同的数据。它显示了就职演说语料库中的三个最常用的词和三个不最常用的词。令人感兴趣的是词频值的压缩。在语料库中出现频率约2000倍以上的词，它的对数归一化版本比文档中只出现一次得分为1（固有地、不公地、疏远地）的词得分只略微高了一点。出现频率超过8000倍的时候，它的分数也只增长了5倍而已。这种压缩用于将文本特征尺寸保持在一个相对小的数值范围内。



类似前面的例子，此图也可以直接使用所提供的代码生成。

Python

```
&gt; python words.py -v  
--logNormalize --
```

```
1 &gt; python words.py -v --logNormalize --genesis  
2 2015-02-02 20:42:02,538 (INFO): The corpus contains 315268 elements after processing
```

词频频率

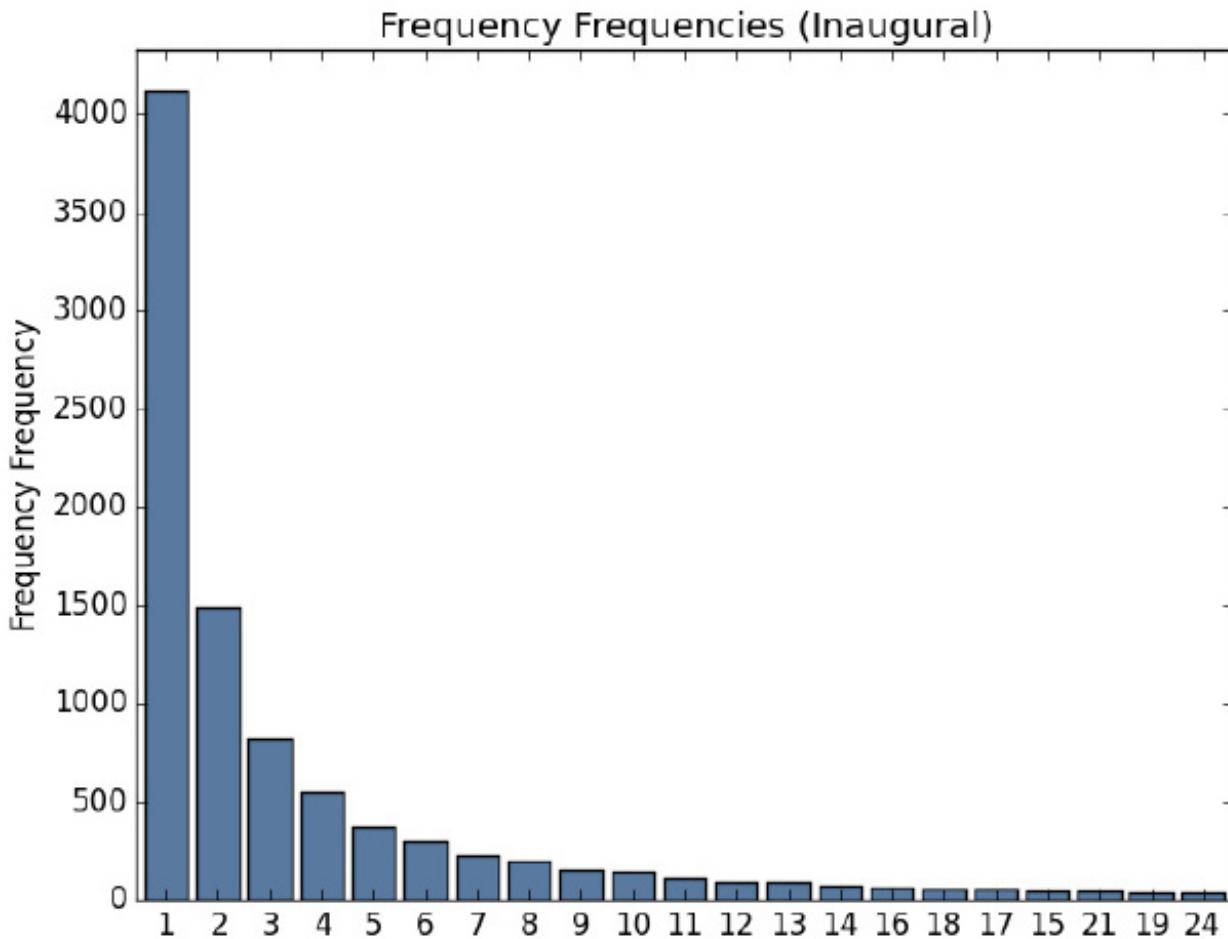
不，这不是一个打字错误。词频频率与我们到目前为止讨论的指标稍有不同。这些信息不太可能被直接当作一个ML算法的特征值来用。然而，它可以为检查语料库结构的人提供很多信息。本质上，词频频率对给定的频率的词项进行计数。说再多也比不上直接举一个例子。如果你运行下面的命令行：

Python

```
&gt; python words.py --  
inaugural -ff
```

```
1 &gt; python words.py --inaugural -ff
```

它会生成一个名为frequency_frequencies.csv的CSV文件，以及下面的图表。



可以看到，它计算了一个单词被使用特定（大约4200次）次数的计数（1次）。这是一种奇怪的指

标，但它多少可以给你这个语料库是否都是由很少使用的单词所组成的一个大致印象。在这个例子中，整个语料库包含145735个字。例如，让我们认为出现四次或更少次数的单词就是罕见词，别的则是常用词。我们知道，这个语料库中有 $4122+1488+817+547=6,974$ 个或约占总字数4.7%的罕见词。与此相比，国情咨文语料库中有9581，总字数为399822，或约占总字数2%的罕见词。这似乎暗示了就职演说与国情咨文相比有着更丰富的词汇量。这是有用的信息吗？可能有。取决于你想了解该文本的什么方面。

计算词频频率的代码是非常简单的。它和上文利用相同的词频数据。该方法通过遍历“词/词频”字典，并建立一个新的frequency_frequencies字典，累计对应频率的不同词个数。总而言之，我们对出现在每个频率的词条数进行计数。

Python

```
def collect_and_output_freq():
    1 def collect_and_output_frequency_frequencies(corpus, corpus_name, term_frequencies):
    2     if term_frequencies is None:
    3         term_frequencies = collect_term_counts(corpus)
    4
    5     frequency_frequencies = {}
    6     for term, frequency in term_frequencies.iteritems():
    7         if frequency_frequencies.has_key(frequency):
    8             frequency_frequencies[frequency] += 1
    9         else:
   10             frequency_frequencies[frequency] = 1
   11
   12     unsorted_array = [[key,value] for key, value in frequency_frequencies.iteritems()]
   13     sorted_array = sorted(unsorted_array, key=lambda frequency_frequency: frequency_frequency[1], reverse=True)
   14
   15     frequency_frequencies_to_chart = []
   16     frequencies_to_chart = []
   17     output_csv_file = open_csv_file("frequency_frequencies.csv", ["Frequency Frequency", "Term Frequency"])
   18
   19     # we collect frequencies_to_chart and frequency_frequencies_to_chart each into their own single dimensional
   20     # array. Then we pass frequency_frequencies_to_chart in an array so that it is 2D as needed by the chart.
   21     # This means there is exactly 1 data set and 6 columns of data in the set. There is no second set to compare
   22     # it to.
   23     for index, (term_frequency, frequency_frequency) in enumerate(sorted_array):
   24         output_csv_file.writerow([frequency_frequency] + [term_frequency])
   25         if index <= 20:
   26             frequencies_to_chart.extend([term_frequency])
   27             frequency_frequencies_to_chart.extend([frequency_frequency])
   28
   29     charting.bar_chart( "frequency_frequencies.png",
   30                         [frequency_frequencies_to_chart],
   31                         "Frequency Frequencies (" + corpus_name + ")",
   32                         frequencies_to_chart,
   33                         "Frequency Frequency",
   34                         None,
   35                         ['#59799e', '#810CE8', '#FF0000', '#12995D', '#FD53FF', '#AA55CC'],
   36                         0.2, 0.0)
   37
   38 return frequency_frequencies
```

总结

在这一课中，我们研究了一些基本指标和文本分析的一些基础模块。我们没有做任何的机器学习相关

的事情。别担心，ML代码的干货即将到来。在我们进行到那里之前，我们要先了解基础的概率论和一些简单的语言建模技术。将本课程、概率论和一些语言建模技术的结合起来，能带领我们接触第一个真正的机器学习任务——朴素贝叶斯分类器。从那里开始，我们将接触一系列不同的ML相关的话题，所以暂时别离开，我们保证讨论ML代码前的主题也能引起你的兴趣。

最后，我们希望这个系列是可延展且有见地的。如果我们发现更多更加清晰的材料或新的有用的例子，我们将把它们添加进来。如果你觉得有什么是值得添加的，也请留言给我们。同样重要的是，如果读者发现任何我们弄错的地方，不管是代码或以其他方面，不要犹豫，马上告知我们。我们对此表示衷心的感谢。

1 赞 1 收藏 [评论](#)

关于作者：[霉霉](#)



据说简介不能为空。 [个人主页](#) · [我的文章](#) · 10

Python爬虫实战（5）：模拟登录淘宝并获取所有订单

本文作者：[伯乐在线 - 崔庆才](#)。未经作者许可，禁止转载！

欢迎加入伯乐在线[专栏作者](#)。

- [Python爬虫入门（1）：综述](#)
- [Python爬虫入门（2）：爬虫基础了解](#)
- [Python爬虫入门（3）：Urllib库的基本使用](#)
- [Python爬虫入门（4）：Urllib库的高级用法](#)
- [Python爬虫入门（5）：URLError异常处理](#)
- [Python爬虫入门（6）：Cookie的使用](#)
- [Python爬虫入门（7）：正则表达式](#)
- [Python爬虫入门（8）：Beautiful Soup的用法](#)
- [Python爬虫实战（1）：爬取糗事百科段子](#)
- [Python爬虫实战（2）：百度贴吧帖子](#)
- [Python爬虫实战（3）：计算大学本学期绩点](#)
- [Python爬虫实战（3）：计算大学本学期绩点](#)
- [Python爬虫实战（5）：模拟登录淘宝并获取所有订单](#)

经过多次尝试，模拟登录淘宝终于成功了，实在是不容易，淘宝的登录加密和验证太复杂了，煞费苦心，在此写出来和大家一起分享，希望大家支持。

本篇内容

1. python模拟登录淘宝网页
2. 获取登录用户的所有订单详情
3. 学会应对出现验证码的情况
4. 体会一下复杂的模拟登录机制

探索部分成果

1. 淘宝的密码用了AES加密算法，最终将密码转化为256位，在POST时，传输的是256位长度的密码。
2. 淘宝在登录时必须要输入验证码，在经过几次尝试失败后最终获取了验证码图片让用户手动输入来验证。
3. 淘宝另外有复杂且每天在变的 ua 加密算法，在程序中我们需要提前获取某一 ua 码才可进行模拟登录。
4. 在获取最后的登录 st 码时，历经了多次请求和正则表达式提取，且 st 码只可使用一次。

整体思路梳理

1. 手动到浏览器获取 ua 码以及 加密后的密码，只获取一次即可，一劳永逸。
2. 向登录界面发送登录请求，POST 一系列参数，包括 ua 码以及密码等等，获得响应，提取验证码图像。
3. 用户输入手动验证码，重新加入验证码数据再次用 POST 方式发出请求，获得响应，提取 J_Htoken。
4. 利用 J_Htoken 向 alipay 发出请求，获得响应，提取 st 码。
5. 利用 st 码和用户名，重新发出登录请求，获得响应，提取重定向网址，存储 cookie。
6. 利用 cookie 向其他个人页面如订单页面发出请求，获得响应，提取订单详情。

是不是没看懂？没事，下面我将一点点说明自己模拟登录的过程，希望大家可以理解。

前期准备

由于淘宝的 ua 算法和 aes 密码加密算法太复杂了，ua 算法在淘宝每天都是在变化的，不过，这个内容你获取之后一直用即可，经过测试之后没有问题，一劳永逸。

那么 ua 和 aes 密码怎样获取呢？

我们就从浏览器里面直接获取吧，打开浏览器，找到淘宝的登录界面，按 F12 或者浏览器右键审查元素。

在这里我用的是火狐浏览器，首先记得在浏览器中设置一下显示持续日志，要不然页面跳转了你就看不到之前抓取的信息了。在这里截图如下：



淘宝网
Taobao.com

默认的 Firefox 开发者工具

- 着色器编辑器
- Canvas
- 性能
- 时间线
- 网络
- 存储
- 网络音频
- 代码草稿纸 *

可用的工具箱按钮

- 从页面中选择一个元素

选择一个 frame 作为当前的日志面板

● 网络(N) ● CSS JS 安全(U) 日志(L) 清空(R)

选择开发工具的主题风格：

暗色主题 亮色主题

常用首选项

启用持续日志

查看器

显示浏览器样式

默认颜色单位(U) Hex

Web 控制台

启用时间戳

样式编辑器

编辑器首选项

检测缩进

自动关闭括号

使用空格缩进

缩进尺寸(I) 2

按键绑定(K) 默认设置

高级设置

禁用缓存(工具箱打开时)

禁用 JavaScript *

启用 chrome 及附加组件调试

启用远程调试

好，那么接下来我们就从浏览器中获取 ua 和 aes 密码

点击网络选项卡，这时都是空的，什么数据也没有截取。这时你就在网页上登录一下试试吧，输入用户名啊，密码啊，有必要时需要输入验证码，点击登录。

淘宝网
Taobao.com

登录

手机号/会员名/邮箱

查看器 控制台 调试器 样式编辑器 Canvas 性能 网络

方法 文件 域名 类型 大小 时间线

• 请进行至少一项请求，或者 **重新载入** 此页面以查阅详细的网络活动信息。

• 点击 **性能** 按钮开始性能分析。

等跳转成功后，你就可以看到好多日志记录了，点击图中的那一行 login.taobao.com，然后查看参数，你就会发现表单数据了，其中包括 ua 还有下面的 password2，把这两复制下来，我们之后要用到的。这就是我们需要的 ua 还有 aes 加密后的密码。

The screenshot shows the NetworkMiner interface. On the left, a list of network requests is displayed, with the fourth POST request to 'login.jhtml' selected. On the right, the details of this selected request are shown in a large pane. The '参数' (Parameters) tab is selected, displaying several parameters: redirectURL, TPL_username, TPL_password, TPL_checkcode, loginsite, newlogin, TPL_redirect_url, from, fc, style, and css_style. The 'ua' parameter is also listed. A red box highlights the 'ua' parameter and its value.

恩，读到这里，你应该获取到了属于自己的 ua 和 password2 两个内容。

输入验证码并获取J_HToken

经过博主本人亲自验证，有时候，在模拟登录时你并不需要输入验证码，它直接返回的结果就是前面所说的下一步用到的 J_HToken，而有时候你则会需要输入验证码，等你手动输入验证码之后，重新请求登录一次。

博主是边写程序边更新文章的，现在写完了是否有必要输入验证码的检验以及在浏览器中呈现验证码。

代码如下

Python

```
author__ = 'CQC'
# -*- coding: utf-8 -*-
import urllib
import urllib2
import cookielib
import re
import webbrowser
#模拟登录淘宝类
class Taobao:
    #初始化方法
    def __init__(self):
        #登录的URL
        self.loginURL = "https://login.taobao.com/member/login.jhtml"
        #代理IP地址，防止自己的IP被封禁
        self.proxyURL = 'http://120.193.146.97:843'
        #登录POST数据时发送的头部信息
        self.loginHeaders = {
            'Host': 'login.taobao.com',
            'User-Agent': 'Mozilla/5.0 (Windows NT 6.1; WOW64; rv:35.0) Gecko/20100101 Firefox/35.0',
            'Referer': 'https://login.taobao.com/member/login.jhtml',
            'Content-Type': 'application/x-www-form-urlencoded',
            'Connection': 'Keep-Alive'
        }
        #用户名
        self.username = 'cqcre'
        #ua字符串，经过淘宝ua算法计算得出，包含了时间戳,浏览器,屏幕分辨率,随机数,鼠标移动,鼠标点击,其实还有键盘输入记录,鼠标移动的记录、点击的记录等等的信息
        self.ua =
'191UW5TcyMNYQwiAiwTR3tCf0J/QnhEcUpkMmQ=|Um5Ockt0TXdPc011TXVKdyE=|U2xMHDJ+H2QJZwBxX39Rb1d5WXcrSixAJ1kjDVsnJVGhXd1lIXGNaYFhkWmJaYI1gV2pldUtyTXRKfkN4Q
#密码，在这里不能输入真实密码，淘宝对此密码进行了加密处理，256位，此处为加密后的密码
self.password2 = '7511aa68sx629e45de220d29174f1066537a73420ef6dbb5b46f202396703a2d56b0312df8769d886e6ca63d587fdbb99ee73927e8c07d9c88cd02182e1a21edc13fb8e140a4
self.post = post = {
    'ua':self.ua,
    'TPL_checkcode':'',
    'CtrlVersion': '1,0,0,7',
    'TPL_password':'',
    'TPL_redirect_url':'http://i.taobao.com/my_taobao.htm?nekot=udm8087E1424147022443',
    'TPL_username':self.username,
    'loginsite':'0',
    'newlogin':'0',
    'from':'tb',
    'fc':'default',
    'style':'default',
    'css_style':'',
    'tid':'XOR_1_00000000000000000000000000000000_625C4720470A0A050976770A',
    'support':'000001',
    'loginType':'4',
    'minititle':'',
    'minipara':'',
    'unto':'NaN',
    'pstrong':'3',
    'lnick':'',
}
```

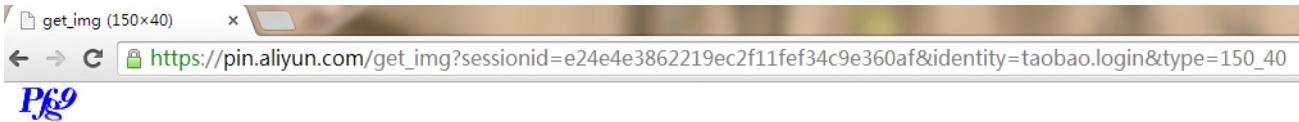
```

55     'sign':",
56     'need_sign':",
57     'isignore':",
58     'full_redirect':",
59     'popid':",
60     'callback':",
61     'guf':",
62     'not_duplike_str':",
63     'need_user_id':",
64     'poy':",
65     'gvfdcname':'10',
66     'gvfdcre':",
67     'from_encoding ':",
68     'sub':",
69     'TPL_password_2':self.password2,
70     'loginASR':1,
71     'loginASRSuc':1',
72     'allp':",
73     'oslanguage':'zh-CN',
74     'sr':1366*768',
75     'osVer':'windows|6.1',
76     'naviVer':'firefox|35'
77 }
78 #将POST的数据进行编码转换
79 self.postData = urllib.urlencode(self.post)
80 #设置代理
81 self.proxy = urllib2.ProxyHandler({'http':self.proxyURL})
82 #设置cookie
83 self.cookie = cookielib.LWPCookieJar()
84 #设置cookie处理器
85 self.cookieHandler = urllib2.HTTPCookieProcessor(self.cookie)
86 #设置登录时用到的opener, 它的open方法相当于urllib2.urlopen
87 self.opener = urllib2.build_opener(self.cookieHandler,self.proxy,urllib2.HTTPHandler)
88
89 #得到是否需要输入验证码, 这次请求的相应有时会不同, 有时需要验证有时不需要
90 def needIdenCode(self):
91     #第一次登录获取验证码尝试, 构建request
92     request = urllib2.Request(self.loginURL,self.postData,self.loginHeaders)
93     #得到第一次登录尝试的相应
94     response = self.opener.open(request)
95     #获取其中的内容
96     content = response.read().decode('gbk')
97     #获取状态吗
98     status = response.getcode()
99     #状态码为200, 获取成功
100    if status == 200:
101        print u"获取请求成功"
102        #\u8bf7\u8f93\u5165\u9a8c\u8bc1\u7801这六个字是请输入验证码的utf-8编码
103        pattern = re.compile(u'\u8bf7\u8f93\u5165\u9a8c\u8bc1\u7801',re.S)
104        result = re.search(pattern,content)
105        #如果找到该字符, 代表需要输入验证码
106        if result:
107            print u"此次安全验证异常, 您需要输入验证码"
108            return content
109        #否则不需要
110        else:
111            print u"此次安全验证通过, 您这次不需要输入验证码"
112            return False
113    else:
114        print u"获取请求失败"
115
116 #得到验证码图片
117 def getIdenCode(self,page):
118     #得到验证码的图片
119     pattern = re.compile('python -V
Python 2.7.9

C:\>python taobao_login.py
Traceback (most recent call last):
  File "taobao_login.py", line 153, in <module>
    taobao.main()
  File "taobao_login.py", line 135, in main
    needResult = self.needIdenCode()
  File "taobao_login.py", line 94, in needIdenCode
    response = self.opener.open<request>
  File "c:\Python27\lib\urllib2.py", line 431, in open
    response = self._open<req, data>
  File "c:\Python27\lib\urllib2.py", line 449, in _open
    '_open', req)
  File "c:\Python27\lib\urllib2.py", line 409, in _call_chain
    result = func(*args)
  File "c:\Python27\lib\urllib2.py", line 1240, in https_open
    context=self._context)
  File "c:\Python27\lib\urllib2.py", line 1197, in do_open
    raise URLError(err)
urllib2.URLError: <urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verification failed <_ssl.c:581>>
```

经过查证，竟然是版本问题，博主本人用的是 2.7.7，而小伙伴用的是 2.7.9。后来换成 2.7.7 就好了…，我也是醉了，希望有相同错误的小伙伴，可以尝试换一下版本…

好啦，运行时会弹出浏览器，如图



那么，我们现在需要手动输入验证码，重新向登录界面发出登录请求，之前的post数据内容加入验证码这一项，重新请求一次，如果请求成功，则会返回下一步我们需要的 J_HToken，如果验证码输入错误，则会返回验证码输入错误的选项。好，下面，我已经写到了获取J_HToken的进度，代码如下，现在运行程序，会蹦出浏览器，然后提示你输入验证码，用户手动输入之后，则会返回一个页面，我们提取出 J_Htoken即可。

注意，到现在为止，你还没有登录成功，只是获取到了J_HToken的值。

目前写到的代码如下

Python

```
author__ = 'CQC'
#-*- coding:utf-8 -*-
import urllib
import urllib2
import cookielib
import re
import webbrowser
#模拟登录淘宝类
class Taobao:
    #初始化方法
    def __init__(self):
        #登录的URL
        self.loginURL = "https://login.taobao.com/member/login.jhtml"
        #代理IP地址，防止自己的IP被封禁
        self.proxyURL = 'http://120.193.146.97:843'
        #登录POST数据时发送的头部信息
        self.loginHeaders = {
            'Host': 'login.taobao.com',
            'User-Agent': 'Mozilla/5.0 (Windows NT 6.1; WOW64; rv:35.0) Gecko/20100101 Firefox/35.0',
            'Referer': 'https://login.taobao.com/member/login.jhtml',
            'Content-Type': 'application/x-www-form-urlencoded',
            'Connection': 'Keep-Alive'
        }
    #用户名
    self.username = 'cqcre'
    #ua字符串，经过淘宝ua算法计算得出，包含了时间戳,浏览器,屏幕分辨率,随机数,鼠标移动,鼠标点击,其实还有键盘输入记录,鼠标移动的记录、点击的记录等等的信息
```

```

29     self.ua =
30     '191UW5TcyMNYQwiAiwTR3tCf0J/QnhEcUpkMmQ=|Um5Ockt0TXdPc011TXVKdyE=|U2xMHDJ+H2QJZwBxX39Rb1d5WXcrSixAJ1kjDVsN|VGhXd1lIXGNaYFhkWmJaYI1gV2pldUtyTXRKfkN4Q
31     #密码，在这里不能输入真实密码，淘宝对此密码进行了加密处理，256位，此处为加密后的密码
32     self.password2 = '7511aa6854629e45de220d29174f1066537a73420ef6dbb5b46f202396703a2d56b0312df8769d886e6ca63d587fdbb99ee73927e8c07d9c88cd02182e1a21edc13fb8e0a4a2
33     self.post = post = {
34         'ua':self.ua,
35         'TPL_checkcode':'',
36         'CtrlVersion': '1.0.0.7',
37         'TPL_password':'',
38         'TPL_redirect_url':'http://i.taobao.com/my_taobao.htm?nekot=udm8087E1424147022443',
39         'TPL_username':self.username,
40         'loginsite':'0',
41         'newlogin':'0',
42         'from':'tb',
43         'fc':'default',
44         'style':'default',
45         'css_style':'',
46         'tid':'XOR_1_00000000000000000000000000000000_625C4720470A0A050976770A',
47         'support':'000001',
48         'loginType':4,
49         'minititle':'',
50         'minipara':'',
51         'umto':'NaN',
52         'pstrong':'3',
53         'lnick':'',
54         'sign':'',
55         'need_sign':'',
56         'isignore':'',
57         'full_redirect':'',
58         'popid':'',
59         'callback':'',
60         'guf':'',
61         'not_duplike_str':'',
62         'need_user_id':'',
63         'pay':'',
64         'gvfdcname':'10',
65         'gvfdcre':'',
66         'from_encoding ':'',
67         'sub':'',
68         'TPL_password_2':self.password2,
69         'loginASR':1,
70         'loginASRSuc':1,
71         'allp':'',
72         'oslanguage':'zh-CN',
73         'sr':1366*768,
74         'osVer':'windows|6.1',
75         'naviVer':'firefox|35'
76     }
77     #将POST的数据进行编码转换
78     self.postData = urllib.urlencode(self.post)
79     #设置代理
80     self.proxy = urllib2.ProxyHandler({'http':self.proxyURL})
81     #设置cookie
82     self.cookie = cookielib.LWPCookieJar()
83     #设置cookie处理器
84     self.cookieHandler = urllib2.HTTPCookieProcessor(self.cookie)
85     #设置登录时用到的opener，它的open方法相当于urllib2.urlopen
86     self.opener = urllib2.build_opener(self.cookieHandler,self.proxy,urllib2.HTTPHandler)
87
88     #得到是否需要输入验证码，这次请求的相应有时会不同，有时需要验证有时不需要
89     def needCheckCode(self):
90         #第一次登录获取验证码尝试，构建request
91         request = urllib2.Request(self.loginURL,self.postData,self.loginHeaders)
92         #得到第一次登录尝试的相应
93         response = self.opener.open(request)
94         #获取其中的内容
95         content = response.read().decode('gbk')
96         #获取状态吗
97         status = response.getcode()
98         #状态码为200，获取成功
99         if status == 200:
100             print u"获取请求成功"
101             ##u8bf7#u8f93#u5165#u9a8c#u8bc1#u7801这六个字是请输入验证码的utf-8编码
102             pattern = re.compile(u'\u8bf7\u8f93\u5165\u9a8c\u8bc1\u7801',re.S)
103             result = re.search(pattern,content)
104             print content
105             #如果找到该字符，代表需要输入验证码
106             if result:
107                 print u"此次安全验证异常，您需要输入验证码"
108                 return content
109             #否则不需要
110             else:
111                 #返回结果直接带有J_HToken字样，表明直接验证通过
112                 tokenPattern = re.compile('id="J_HToken"')
113                 tokenMatch = re.search(tokenPattern,content)
114                 if tokenMatch:
115                     print u"此次安全验证通过，您这次不需要输入验证码"
116                     return False
117             else:
118                 print u"获取请求失败"
119                 return None
120
121     #得到验证码图片
122     def getCheckCode(self,page):
123         #得到验证码的图片
124         pattern = re.compile('(含运费: 0.00) | 交易成功<br>订单详情<br>双方已评 |
|                                       | 保障卡    | 手机订单 |      |                      |                      |

Elements Network Sources Timeline Profiles Resources Audits Console

```

<tbody data-isarchive="true" data-orderid="751436187556149" data-status="TRADE_FINISHED" class="mainOrder751436187556149 success-order xcard">...</tbody>
<tbody data-isarchive="true" data-orderid="711894174136149" data-status="TRADE_FINISHED" class="mainOrder711894174136149 success-order xcard">...</tbody>
<tbody data-isarchive="true" data-orderid="701979936516149" data-status="TRADE_FINISHED" class="mainOrder701979936516149 success-order xcard">
 <tr class="sep-row">...</tr>
 <tr class="order-hd">
 <td class="first">
 <div class="summary">
 ...
 2014-06-20

 "订单号: "
 701979936516149

 </div>
 </td>
 <td class="column" colspan="2">
 谷养生
 </td>
 <td class="column">...</td>
 <td class="last" colspan="3">...</td>

```

我们现在想获取订单时间，订单号，卖家店铺名称，宝贝名称，原价，购买数量，最后付款多少，交易状态这几个量，具体就不再分析啦，正则表达式还不熟悉的同学请参考前面所说的正则表达式的用法，在这里，正则表达式匹配的代码是

Python

```

#u^u8ba2\u5355\u53f7'是订单号的编码
1 #u^u8ba2\u5355\u53f7'是订单号的编码
2 pattern = re.compile(u'dealtime.*?>(.?).*?u8ba2\u5355\u53f7.*?(.?).*?shopname.*?title="(.*?)" .*?baobei-name">.*<a.*?>(.?).*?
3 u'price.*?title="(.*?)" .*?quantity.*?title="(.*?)" .*?amount.*?em.*?>(.?).*?trade-status.*?<a.*?>(.?)',re.S)
4 result = re.findall(pattern,page)
5 for item in result:
6 print '-----'
7 print "购买日期:",item[0].strip(), '订单号:',item[1].strip(), '卖家店铺:',item[2].strip()
8 print '宝贝名称:',item[3].strip()
9 print '原价:',item[4].strip(), '购买数量:',item[5].strip(), '实际支付:',item[6].strip(), '交易状态:',item[7].strip()

```

## 最终代码整理

恩，你懂得，最重要的东西来了，经过博主2天多的奋战，代码基本就构建完成。写了两个类，其中提取页面信息的方法我单独放到了一个类中，叫 tool.py，类名为 Tool。

先看一下运行结果吧~

```
获取请求成功
此次安全验证异常，您需要输入验证码
您需要手动输入验证码
验证码获取成功
请在浏览器中输入您看到的验证码
请输入验证码: 6psk
验证码输入正确
成功获取st码
登录网址成功
找到了共多少页
共 7 页
获取到的商品列表如下

购买日期: 2015-02-20 订单号: 982310262226149 卖家店铺: 小米官方旗舰店
宝贝名称: 小米旗舰店MIUI/小米 小米盒子增强版1G高清网络电视机顶盒播放器
原价: 299.01 购买数量: 1 实际支付: 377.93 交易状态: 快件已揽收

购买日期: 2015-01-29 订单号: 959538596226149 卖家店铺: 五颗星时尚男装店
宝贝名称: 新款冬季男士保暖衬衫加绒加厚大码衬衣男款修身韩版免烫潮男衬衫
原价: 398.00 购买数量: 1 实际支付: 96.00 交易状态: 交易成功

购买日期: 2015-01-25 订单号: 952957475766149 卖家店铺: Hers无线特卖店
宝贝名称: 山东wlan一天卡【潍坊专用】非3天非7天到次日8点 自动发货
原价: 1.48 购买数量: 1 实际支付: 1.48 交易状态: 交易成功

购买日期: 2015-01-23 订单号: 951139553306149 卖家店铺: 颐妍堂旗舰店
```

最终代码如下

Python

```
tool.py
```

1 tool.py

Python

```
author__ = 'CQC'
1 __author__ = 'CQC'
2 # -*- coding:utf-8 -*-
3
4 import re
5
6 #处理获得的宝贝页面
7 class Tool:
8
9 #初始化
10 def __init__(self):
11 pass
12
13 #获得页码数
14 def getPageNum(self,page):
15 pattern = re.compile(u'<div class="total">.*?\u5171(.*)\u9875',re.S)
16 result = re.search(pattern,page)
17 if result:
18 print "找到了共多少页"
19 pageNum = result.group(1).strip()
20 print '共',pageNum,'页'
21 return pageNum
22
23 def getGoodsInfo(self,page):
24 #u'\u8ba2\u5355\u53f7'是订单号的编码
25 pattern = re.compile(u'dealtime.*?>(.*).*?\u8ba2\u5355\u53f7.*?(.*).*?shopname.*?title="(.*?)" .*?baobei-name">.*?<a.*?>(.?).*?
26 u'price.*?title="(.*?)" .*?quantity.*?title="(.*?)" .*?amount.*?em.*?>(.*).*?trade-status.*?<a.*?>(.?)',re.S)
27 result = re.findall(pattern,page)
28 for item in result:
29 print '-----'
30 print "购买日期:",item[0].strip(), '订单号:',item[1].strip(), '卖家店铺:',item[2].strip()
31 print "宝贝名称:",item[3].strip()
32 print '原价:',item[4].strip(), '购买数量:',item[5].strip(), '实际支付:',item[6].strip(), '交易状态:',item[7].strip()
```

Python

```
taobao.py
```

1 taobao.py

Python

```
author__ = 'CQC'
1 __author__ = 'CQC'
```

1



```

104 #获取其中的内容
105 content = response.read().decode('gbk')
106 #获取状态吗
107 status = response.getcode()
108 #状态码为200， 获取成功
109 if status == 200:
110 print u"获取请求成功"
111 #\u8bf7\u8f93\u5165\u9a8c\u8bc1\u7801这六个字是请输入验证码的utf-8编码
112 pattern = re.compile(u'\u8bf7\u8f93\u5165\u9a8c\u8bc1\u7801',re.S)
113 result = re.search(pattern,content)
114 #如果找到该字符，代表需要输入验证码
115 if result:
116 print u"此次安全验证异常，您需要输入验证码"
117 return content
118 #否则不需要
119 else:
120 #返回结果直接带有J_HToken字样，表明直接验证通过
121 tokenPattern = re.compile('id="J_HToken" value="(.*?)"')
122 tokenMatch = re.search(tokenPattern,content)
123 if tokenMatch:
124 self.J_HToken = tokenMatch.group(1)
125 print u"此次安全验证通过，您这次不需要输入验证码"
126 return False
127 else:
128 print u"获取请求失败"
129 return None
130
131 #得到验证码图片
132 def getCheckCode(self,page):
133 #得到验证码的图片
134 pattern = re.compile('(.*?).*?(.*?).*?(.*?)',re.S)
23 items = re.findall(pattern,page)
24 for item in items:
25 print item[0],item[1],item[2],item[3],item[4]
26
27spider = Spider()
```

```
28 spider.getContents(1)
```

运行结果如下



```
D:\python2.7\python.exe D:/MyPy/mm/spider.py
http://mm.taobao.com/json/request_top_list.htm?page=1
http://mm.taobao.com/687471686.htm http://img07.taobaocdn.com/sns_logo/i7/TB18D3rGVXXXXvXVXXSubbFXXX.jpg_60x60.jpg 田媛媛 25 广州市
http://mm.taobao.com/405095521.htm http://img05.taobaocdn.com/sns_logo/i5/TB1FM8wGFXXXXb1XXXXSubbFXXX.jpg_60x60.jpg 朱亦颖 25 杭州市
http://mm.taobao.com/631300490.htm http://img08.taobaocdn.com/sns_logo/i8/TB1qi5WGVXXXXxFXXXSubbFXXX.jpg_60x60.jpg 崔辰辰 26 杭州市
http://mm.taobao.com/414457129.htm http://img01.taobaocdn.com/sns_logo/i1/TB1gxkiGVXXXXZpXXSubbFXXX.jpg_60x60.jpg 大猫儿 29 广州市
http://mm.taobao.com/141234233.htm http://img01.taobaocdn.com/sns_logo/i1/TB1P980FVXXXXQXFXXXSubbFXXX.jpg_60x60.jpg 金甜甜 24 广州市
http://mm.taobao.com/96614110.htm http://img04.taobaocdn.com/sns_logo/i4/TB1nxAbGFXXXXa3aXXXSubbFXXX.jpg_60x60.jpg 紫轩 28 杭州市
http://mm.taobao.com/37448401.htm http://img08.taobaocdn.com/sns_logo/i8/TB1bU.UFFXXXXcOXpXXSubbFXXX.jpg_60x60.jpg 谢婷婷 26 杭州市
http://mm.taobao.com/74386764.htm http://img08.taobaocdn.com/sns_logo/i8/TB1HmB7GXXXXbHXFXXXSubbFXXX.jpg_60x60.jpg 夏晨洁 26 杭州市
http://mm.taobao.com/523216808.htm http://img03.taobaocdn.com/sns_logo/i3/TB1CQbiGVXXXXaYXFXXXSubbFXXX.jpg_60x60.jpg Cherry 27 广州市
http://mm.taobao.com/46599595.htm http://img07.taobaocdn.com/sns_logo/i7/TB1QPszGXXXXbSXVXXSubbFXXX.jpg_60x60.jpg 雪倩nik 24 杭州市

Process finished with exit code 0
```

## 2.文件写入简介

在这里，我们有写入图片和写入文本两种方式

### 1) 写入图片

Python

```
运行结果如下
```

```
1 运行结果如下
2
3 QQ截图20150220234132
4 2.文件写入简介
5
6 在这里，我们有写入图片和写入文本两种方式
7 1) 写入图片
```

### 2) 写入文本

Python

```
def saveBrief(self,content,n)
1 def saveBrief(self,content,name):
2 fileName = name + "/" + name + ".txt"
3 f = open(fileName,"w+")
4 print u"正在偷偷保存她的个人信息为",fileName
5 f.write(content.encode('utf-8'))
```

### 3) 创建新目录

Python

```
#创建新目录
def mkdir(self,path):
1 #创建新目录
2 def mkdir(self,path):
3 path = path.strip()
4 # 判断路径是否存在
5 # 存在 True
6 # 不存在 False
7 exists=os.path.exists(path)
8 # 判断结果
9 if not exists:
10 # 如果不存在则创建目录
11 # 创建目录操作函数
12 os.makedirs(path)
13 return True
14 else:
15 # 如果目录存在则不创建，并提示目录已存在
16 return False
```

## 3.代码完善

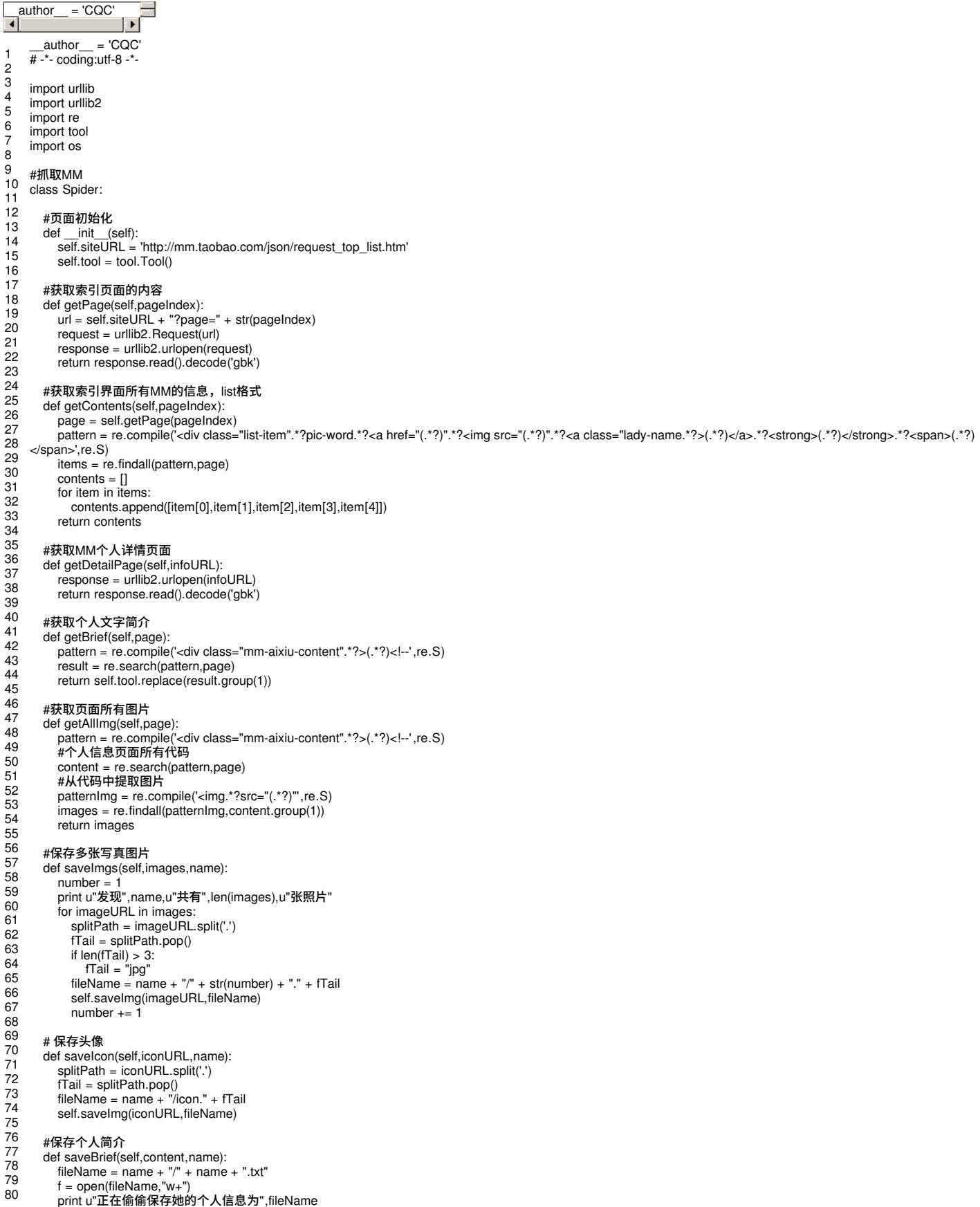
主要的知识点已经在前面都涉及到了，如果大家前面的章节都已经看了，完成这个爬虫不在话下，具体的详情在此不再赘述，直接帖代码啦。

## Python

spider.py

1 spider.py

## Python

```
author__ = 'CQC' 
1 __author__ = 'CQC'
2 # -*- coding:utf-8 -*
3
4 import urllib
5 import urllib2
6 import re
7 import tool
8 import os
9
10 #抓取MM
11 class Spider:
12 #页面初始化
13 def __init__(self):
14 self.siteURL = 'http://mm.taobao.com/json/request_top_list.htm'
15 self.tool = tool.Tool()
16
17 #获取索引页面的内容
18 def getPage(self,pageIndex):
19 url = self.siteURL + "?page=" + str(pageIndex)
20 request = urllib2.Request(url)
21 response = urllib2.urlopen(request)
22 return response.read().decode('gbk')
23
24 #获取索引界面所有MM的信息，list格式
25 def getContents(self,pageIndex):
26 page = self.getPage(pageIndex)
27 pattern = re.compile('<div class="list-item".*?pic-word.*?<a href="(.*?)".*?(.*?).*?(.*?).*?(.*?)',re.S)
28 items = re.findall(pattern,page)
29 contents = []
30 for item in items:
31 contents.append([item[0],item[1],item[2],item[3],item[4]])
32 return contents
33
34
35 #获取MM个人详情页面
36 def getDetailPage(self,infoURL):
37 response = urllib2.urlopen(infoURL)
38 return response.read().decode('gbk')
39
40 #获取个人文字简介
41 def getBrief(self,page):
42 pattern = re.compile('<div class="mm-aixiu-content".*?>(.*?)<!--',re.S)
43 result = re.search(pattern,page)
44 return self.tool.replace(result.group(1))
45
46 #获取页面所有图片
47 def getAllImg(self,page):
48 pattern = re.compile('<div class="mm-aixiu-content".*?>(.*?)<!--',re.S)
49 #个人信息页面所有代码
50 content = re.search(pattern,page)
51 #从代码中提取图片
52 patternImg = re.compile('<img.*?src="(.*?)"',re.S)
53 images = re.findall(patternImg,content.group(1))
54 return images
55
56 #保存多张写真图片
57 def saveImgs(self,images,name):
58 number = 1
59 print u"发现",name,u"共有",len(images),u"张照片"
60 for imageURL in images:
61 splitPath = imageURL.split('.')
62 fTail = splitPath.pop()
63 if len(fTail) > 3:
64 fTail = "jpg"
65 fileName = name + "/" + str(number) + "." + fTail
66 self.saveImg(imageURL,fileName)
67 number += 1
68
69 #保存头像
70 def saveIcon(self,iconURL,name):
71 splitPath = iconURL.split('.')
72 fTail = splitPath.pop()
73 fName = name + "/icon." + fTail
74 self.saveImg(iconURL,fName)
75
76 #保存个人简介
77 def saveBrief(self,content,name):
78 fName = name + "/" + name + ".txt"
79 f = open(fName,"w+")
80 print u"正在偷偷保存她的个人信息为",fName
```

```

81 f.write(content.encode('utf-8'))
82
83 #传入图片地址，文件名，保存单张图片
84 def saveImg(self,imageURL,fileNamed):
85 u = urllib.urlopen(imageURL)
86 data = u.read()
87 f = open(fileNamed, 'wb')
88 f.write(data)
89 print u"正在悄悄保存她的一张图片为",fileNamed
90 f.close()
91
92 #创建新目录
93 def mkdir(self,path):
94 path = path.strip()
95 # 判断路径是否存在
96 # 存在 True
97 # 不存在 False
98 isExists=os.path.exists(path)
99 # 判断结果
100 if not isExists:
101 # 如果不存在则创建目录
102 print u"偷偷新建了名字叫做",path,u'的文件夹'
103 # 创建目录操作函数
104 os.makedirs(path)
105 return True
106 else:
107 # 如果目录存在则不创建，并提示目录已存在
108 print u"名为",path,'的文件夹已经创建成功'
109 return False
110
111 #将一页淘宝MM的信息保存起来
112 def savePageInfo(self,pageIndex):
113 #获取第一页淘宝MM列表
114 contents = self.getContents(pageIndex)
115 for item in contents:
116 #item[0]个人详情URL,item[1]头像URL,item[2]姓名,item[3]年龄,item[4]居住地
117 print u"发现一位模特,名字叫",item[2],u"芳龄",item[3],u",她在",item[4]
118 print u"正在偷偷地保存",item[2],"的信息"
119 print u"又意外地发现她的个人地址是",item[0]
120 #个人详情页面的URL
121 detailURL = item[0]
122 #得到个人详情页面代码
123 detailPage = self.getDetailPage(detailURL)
124 #获取个人简介
125 brief = self.getBrief(detailPage)
126 #获取所有图片列表
127 images = self.getAllImg(detailPage)
128 self.mkdir(item[2])
129 #保存个人简介
130 self.saveBrief(brief,item[2])
131 #保存头像
132 self.saveIcon(item[1],item[2])
133 #保存图片
134 self.saveImgs(images,item[2])
135
136 #传入起止页码，获取MM图片
137 def savePagesInfo(self,start,end):
138 for i in range(start,end+1):
139 print u"正在偷偷寻找第",i,u"个地方，看看MM们在不在"
140 self.savePageInfo(i)
141
142 #传入起止页码即可，在此传入了2,10,表示抓取第2到10页的MM
143 spider = Spider()
144 spider.savePagesInfo(2,10)

```

Python

tool.py

1 tool.py

Python

```

author__ = 'CQC'
#-*- coding:utf-8 -*-
1 __author__ = 'CQC'
2 #-*- coding:utf-8 -*-
3 import re
4
5 #处理页面标签类
6 class Tool:
7 #去除img标签,1-7位空格,
8 removeImg = re.compile('<img.*?>| {1,7}| ')
9 #删除超链接标签
10 removeAddr = re.compile('<a.*?>|')
11 #把换行的标签换为\n
12 replaceLine = re.compile('
|<div>|</div>|</p>')
13 #将表格制表<td>替换为\t
14 replaceTD = re.compile('<td>')
15 #将换行符或双换行符替换为\n
16 replaceBR = re.compile('

|
')

```

```

17 #将其余标签剔除
18 removeExtraTag = re.compile('<.*?>')
19 #将多行空行删除
20 removeNoneLine = re.compile('\n+')
21 def replace(self,x):
22 x = re.sub(self.removeImg,"",x)
23 x = re.sub(self.removeAddr,"",x)
24 x = re.sub(self.replaceLine,"\n",x)
25 x = re.sub(self.replaceTD,"|t",x)
26 x = re.sub(self.replaceBR,"|n",x)
27 x = re.sub(self.removeExtraTag,"",x)
28 x = re.sub(self.removeNoneLine,"|n",x)
29 #strip()将前后多余内容删除
30 return x.strip()

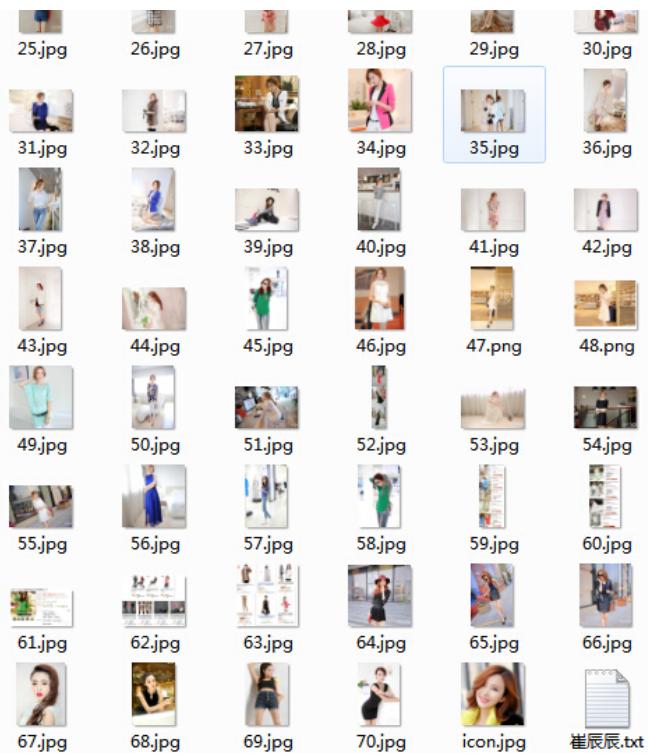
```

以上两个文件就是所有的代码内容，运行一下试试看，那叫一个酸爽啊

正在悄悄保存她的一张图片为 戴誉利/99.jpg  
 正在悄悄保存她的一张图片为 戴誉利/100.jpg  
 正在悄悄保存她的一张图片为 戴誉利/101.jpg  
 正在悄悄保存她的一张图片为 戴誉利/102.jpg  
 正在悄悄保存她的一张图片为 戴誉利/103.jpg  
 正在偷偷寻找第 3 个地方，看看MM们在不在  
 发现一位模特，名字叫 滕雨佳 芳龄 23，她在 杭州市  
 正在偷偷地保存 滕雨佳 的信息  
 又意外地发现她的个人地址是 <http://mm.taobao.com/539549300.htm>  
 名为 滕雨佳 的文件夹已经创建成功  
 正在偷偷保存她的个人信息为 滕雨佳/滕雨佳.txt  
 正在悄悄保存她的一张图片为 滕雨佳/icon.jpg  
 发现 滕雨佳 共有 83 张照片  
 正在悄悄保存她的一张图片为 滕雨佳/1.jpg  
 正在悄悄保存她的一张图片为 滕雨佳/2.jpg  
 正在悄悄保存她的一张图片为 滕雨佳/3.jpg  
 正在悄悄保存她的一张图片为 滕雨佳/4.jpg  
 正在悄悄保存她的一张图片为 滕雨佳/5.jpg  
 正在悄悄保存她的一张图片为 滕雨佳/6.jpg  
 正在悄悄保存她的一张图片为 滕雨佳/7.jpg  
 正在悄悄保存她的一张图片为 滕雨佳/8.jpg  
 正在悄悄保存她的一张图片为 滕雨佳/9.jpg

看看[文件夹](#)里面有什么变化

|        |                |     |
|--------|----------------|-----|
| CC若梵   | 2015/2/21 1:56 | 文件夹 |
| Cherry | 2015/2/21 1:36 | 文件夹 |
| CILY   | 2015/2/21 1:56 | 文件夹 |
| nikki  | 2015/2/21 1:58 | 文件夹 |
| 阿朵拉    | 2015/2/21 1:56 | 文件夹 |
| 崔辰辰    | 2015/2/21 1:22 | 文件夹 |
| 大猫儿    | 2015/2/21 1:34 | 文件夹 |
| 戴誉利    | 2015/2/21 1:57 | 文件夹 |
| 姬嘉琳    | 2015/2/21 1:55 | 文件夹 |
| 金甜甜    | 2015/2/21 1:34 | 文件夹 |
| 李喳喳    | 2015/2/21 1:36 | 文件夹 |
| 喵小咪    | 2015/2/21 1:58 | 文件夹 |
| 沁源     | 2015/2/21 1:54 | 文件夹 |
| 滕雨佳    | 2015/2/21 1:54 | 文件夹 |
| 田媛媛    | 2015/2/21 1:18 | 文件夹 |
| 夏晨洁    | 2015/2/21 1:35 | 文件夹 |
| 夏欢欢    | 2015/2/21 1:49 | 文件夹 |
| 晓莉莉    | 2015/2/21 1:57 | 文件夹 |
| 谢婷婷    | 2015/2/21 1:35 | 文件夹 |
| 熊仔欣    | 2015/2/21 1:56 | 文件夹 |
| 雪倩nika | 2015/2/21 1:36 | 文件夹 |
| 杨羽凡    | 2015/2/21 1:59 | 文件夹 |
| 悦小舞    | 2015/2/21 1:49 | 文件夹 |



不知不觉，海量的MM图片已经进入了你的电脑，还不快快去试试看！！

代码均为本人所敲，写的不好，大神勿喷，写来方便自己，同时分享给大家参考！希望大家支持！

**打赏支持我写出更多好文章，谢谢！**

**打赏作者**

**打赏支持我写出更多好文章，谢谢！**

任选一种支付方式



2 赞 18 收藏 [21 评论](#)

**关于作者：崔庆才**



# Python爬虫实战（3）：计算大学本学期绩点

本文作者：[伯乐在线 - 崔庆才](#)。未经作者许可，禁止转载！

欢迎加入伯乐在线[专栏作者](#)。

- [Python爬虫入门（1）：综述](#)
- [Python爬虫入门（2）：爬虫基础了解](#)
- [Python爬虫入门（3）：Urllib库的基本使用](#)
- [Python爬虫入门（4）：Urllib库的高级用法](#)
- [Python爬虫入门（5）：URLError异常处理](#)
- [Python爬虫入门（6）：Cookie的使用](#)
- [Python爬虫入门（7）：正则表达式](#)
- [Python爬虫入门（8）：Beautiful Soup的用法](#)
- [Python爬虫实战（1）：爬取糗事百科段子](#)
- [Python爬虫实战（2）：百度贴吧帖子](#)
- [Python爬虫实战（3）：计算大学本学期绩点](#)
- [Python爬虫实战（3）：计算大学本学期绩点](#)
- [Python爬虫实战（5）：模拟登录淘宝并获取所有订单](#)

大家好，本次为大家带来的项目是计算大学本学期绩点。首先说明的是，博主来自山东大学，有属于个人的成绩管理系统，需要学号密码才可以登录，不过可能广大读者没有这个学号密码，不能实际进行操作，所以最主要的是了解cookie的相关操作。

## 本篇目标

1. 模拟登录学生成绩管理系统

2. 抓取本学期成绩界面

3. 计算打印本学期成绩

## 1. URL的获取

恩，博主来自山东大学~

先贴一个URL，让大家知道我们学校学生信息系统的网站构架，主页是 [http://jwxt.sdu.edu.cn:7890/zhxt\\_bks/zhxt\\_bks.html](http://jwxt.sdu.edu.cn:7890/zhxt_bks/zhxt_bks.html)，山东大学学生个人[信息系统](#)，进去之后，Oh不，他竟然用了frame，一个多么古老的而又任性的写法，真是惊出一身冷汗~

算了，就算他是frame又能拿我怎么样？我们点到登录界面，审查一下元素，先看看登录界面的URL是怎样的？

The screenshot shows a browser window displaying a login interface. On the left, there is a sidebar with a tree view of menu items: '用户菜单' (User Menu) with '重新登录' (Logout), '修改密码' (Change Password), and '退出系统' (Logout); '个人信息' (Personal Information) with '学籍信息' (Student Record); '选择教材' (Select教材); '选课' (Select Course) with '选定课程' (Selected Courses), '重修报名' (Re-enrollment), '上课表显示' (Class Schedule Display), '本学期课程查询' (Query Courses This Semester), '选课帮助' (Help), and '选课公告' (Announcements); '成绩查询' (Grade Inquiry) with '教学计划' (Teaching Plan) and '已修课程' (Completed Courses). The main content area contains a form with a title '请输入你的学号和口令：'. It has two input fields: '学号' (Student ID) containing '201200131012' and '口令' (Password) containing '\*\*\*\*\*'. Below the fields are two buttons: '确认' (Confirm) and '重置' (Reset). At the bottom of the browser window, the developer tools are open, specifically the 'Elements' tab. The DOM tree shows the following structure:

```
<html>
 <head>...</head>
 <frameset cols="20%,80%">
 <frame name="w_left" src="w_left.html" scrolling="auto" resize>
 <frameset framespacing="0" rows="90%,10%">
 <frame name="w_right" src="xk_login.html" scrolling="auto" resize>
 #document
 <html>
 <head>...</head>
 <body bgcolor="#EAE2F3">...</body>
 </html>
 </frame>
 <frame name="w_footer" src="w_footer.html" scrolling="auto" resize>
 </frameset>
 </frameset>
```

恩，看到了右侧的frame名称，[src="xk\\_login.html"](#)，可以分析出完整的登录界面的网址为[http://jwxt.sdu.edu.cn:7890/zhxt\\_bks/xk\\_login.html](http://jwxt.sdu.edu.cn:7890/zhxt_bks/xk_login.html)，点进去看看，真是棒棒哒，他喵的竟然是清华大学[选课](#)系统，醉了，你说你抄袭就抄袭吧，改改名字也不错啊~

算了，就不和他计较了。现在，我们登录一下，用浏览器监听网络。

我用的是猎豹浏览器，审查元素时会有一个网络的选项，如果大家用的Chrome，也有相对应的功能，Firefox需要装插件HttpFox，同样可以实现。

这个网络监听功能可以监听表单的传送以及请求头，响应头等等的信息。截个图看一下，恩，我偷偷把密码隐藏了，你看不到~

大家看到的是登录之后出现的信息以及NetWork监听，显示了headers的详细信息。

计算机科学与技术学院 计算机科学与技术专业 崔庆才 (201200131012)

你的使用时间为2015-02-20 09:41, 使用地点为140.246.128.46

#### 请您注意！

1. 请节约时间
2. 离开时请注销
3. 每次登录时间限为30分钟

The screenshot shows the Firefox Network Monitor. On the left, there's a list of requests: 'bks\_login2.login?jym2005=6230.324099350951' (status 200 OK), 'bks\_login2.loginmessage' (status 302 Found), and 'style.css' (status 200 OK). On the right, the 'Headers' tab is selected for the first request. It displays the following details:

- Remote Address:** 211.86.56.238:7890
- Request URL:** [http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bks\\_login2.login?jym2005=6230.324099350951](http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bks_login2.login?jym2005=6230.324099350951)
- Request Method:** POST
- Status Code:** 200 OK
- Request Headers:** view source
  - Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8
  - Accept-Encoding: gzip, deflate
  - Accept-Language: zh-CN,zh;q=0.8
  - Cache-Control: max-age=0
  - Connection: keep-alive
  - Content-Length: 32
  - Content-Type: application/x-www-form-urlencoded
  - Host: jwxt.sdu.edu.cn:7890
  - Origin: <http://jwxt.sdu.edu.cn:7890>
  - Referer: [http://jwxt.sdu.edu.cn:7890/zhxt\\_bks/xk\\_login.html](http://jwxt.sdu.edu.cn:7890/zhxt_bks/xk_login.html)
  - User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36
- Query String Parameters:** view source view URL encoded
  - jym2005: 6230.324099350951
- Form Data:** view source view URL encoded
  - stuid: 201200131012
  - pwd: [REDACTED]

3 requests | 1.1 KB transferred | 175 ms (...)

最主要的内容，我们可以发现有一个表单提交的过程，提交方式为POST，两个参数分别为stuid和pwd。

请求的URL为[http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bks\\_login2.login](http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bks_login2.login)，没错，找到表单数据和目标地址就是这么简单。

在这里注意，刚才的[http://jwxt.sdu.edu.cn:7890/zhxt\\_bks/xk\\_login.html](http://jwxt.sdu.edu.cn:7890/zhxt_bks/xk_login.html)只是登录界面的地址，刚刚得到的这个地址才是登录索要提交到的真正的URL。希望大家这里不要混淆。

不知道山大这个系统有没有做headers的检查，我们先不管这么多，先尝试一下模拟登录并保存Cookie。

## 2. 模拟登录

好，通过以上信息，我们已经找到了登录的目标地址为[http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bks\\_login2.login](http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bks_login2.login)

有一个表单提交到这个URL，表单的两个内容分别为stuid和pwd，学号和密码，没有其他的隐藏信息，提交方式为POST。

好，现在我们首先构造以下代码来完成登录。看看会不会获取到登录之后的提示页面。

Python

```
author__ = 'CQC'
-*- coding:utf-8 -*-
1 __author__ = 'CQC'
2 # -*- coding:utf-8 -*-
3
4 import urllib
5 import urllib2
6 import cookielib
7 import re
8
9 #山东大学绩点运算
10 class SDU:
11
12 def __init__(self):
13 self.loginUrl = 'http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bks_login2.login'
```

```

14 self.cookies = cookielib.CookieJar()
15 self.postdata = urllib.urlencode({
16 'stuid':'201200131012',
17 'pwd':'xxxxxx'
18 })
19 self.opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(self.cookies))
20
21 def getPage(self):
22 request = urllib2.Request(
23 url = self.loginUrl,
24 data = self.postdata)
25 result = self.opener.open(request)
26 #打印登录内容
27 print result.read().decode('gbk')
28
29 sdu = SDU()
30 sdu.getPage()

```

测试一下，竟然成功了，[山大](#)这网竟然没有做headers检查，很顺利就登录进去了。

说明一下，在这里我们利用了前面所说的cookie，用到了CookieJar这个对象来保存cookies，另外通过构建[opener](#)，利用open方法实现了登录。如果大家觉得这里有疑惑，请看[Python爬虫入门六之Cookie的使用](#)，这篇文章说得比较详细。

好，我们看一下运行结果

```

Run demo4
D:\python2.7\python.exe D:/MyPy/demo/demo4.py
<HTML><HEAD><TITLE>登录结果</TITLE></HEAD>
<link href="/zhxt_bks/style.css" rel="stylesheet" type="text/css">
<BODY bgColor="#F1ECF9">

计算机科学与技术学院 计算机科学与技术专业 崔庆才 (201200131012)

你的使用时间为 2015-02-20 10:01, 使用地点为 140.246.128.46

<H3>请您注意!</H3>
1. 请节约时间

2. 离开时请注销

3. 每次登录时间限为 30 分钟

</BODY>
</HTML>
<HR>

登录成功!


```

酸爽啊，接下来我们只要再获取到本学期成绩界面然后把成绩抓取出来就好了。

### 3. 抓取本学期成绩

让我们先在浏览器中找到本学期成绩界面，点击左边的本学期成绩。

用户菜单  
+重新登录  
+修改密码  
+退出系统

个人信息  
+学籍信息

选择教材

选课  
+选定课程  
+重修报名  
+课表显示  
+本学期课程查询  
+选课帮助  
+选课公告

成绩查询  
+教学计划  
+已修课程  
+本学期成绩  
+不及格课程  
+成绩查询帮助

**本科生综合查询**

1. 下面列出所有课程的成绩，选中某门课程，按最下端的显示排名按钮，可计算成绩排名。  
2. 计算排名的时间会稍长一些，选择排名的课程不要超过20门。

本学期课程成绩表					
排名否	课程号	课程名	序号	学分	考试时间
<input type="checkbox"/>	0133200310	编译原理与技术	1	2.5	20150112
	0133200461	操作系统课程设计(双语)	1	2	20150112
<input type="checkbox"/>	0133200810	汇编语言	1	2.5	20150112
<input type="checkbox"/>	0133201160	计算机网络课程设计	1	2	20150112
<input type="checkbox"/>	0133201310	面向对象技术	1	2.5	20150112
<input type="checkbox"/>	0133201910	微机原理与接口	1	2.5	20150112
	0133202360	组成原理课程设计	1	2	20150112
<input type="checkbox"/>	0281000410	中国化的马克思主义	335	3	20150112
	0901000510	形势政策与社会实践(5)	336		20150112

**[显示排名] [复位]**

Elements Network Sources Timeline Profiles Resources Audits Console

```

▼<html>
 ▶<head>...</head>
 ▶<frameset cols="20%,80%">
 ▶<frame name="w_left" src="w_left.html" scrolling="auto" resize>
 ▶<frameset framespacing="0" rows="90%,10%">
 ▶<frame name="w_right" src="xk_login.html" scrolling="auto" resize>
 ▶<frame name="w_footer" src="w_footer.html" scrolling="auto" resize>
 </frameset>
 </frameset>

```

重新审查元素，你会发现这个frame的src还是没有变，仍然是xk\_login.html，引起这个页面变化的原因是在左边的本学期成绩这个超链接设置了一个目标frame，所以，那个页面就显示在右侧了。

所以，让我们再审查一下本学期成绩这个超链接的内容是什么~

成绩查询  
+教学计划  
+已修课程  
+本学期成绩  
+不及格课程  
+成绩查询帮助

<input type="checkbox"/>	0133202360	组成原理课程设计
<input type="checkbox"/>	0281000410	中国化的马克思主义
	0901000510	形势政策与社会实践(5)

**[显示排名]**

Elements Network Sources Timeline Profiles Resources Audits Console

```

▼<tr bgcolor="#EAE2F3">
 ▶<td bgcolor="#E2D8EF">
 ▶
 " "

 本学期成绩

 </td>
 </td>
</tr>
▶<tr bgcolor="#EAE2F3">...</tr>

```

恩，找到它了，[本学期成绩](/pls/wwwbks/bkscjcx.curscocre)

那么，完整的URL就是 <http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bkscjcx.curscocre>，好，URL已经找到了，我们继续完善一下代码，获取这个页面。

Python

```

author__ = 'CQC'
-*- coding:utf-8 -*-
1 __author__ = 'CQC'
2 # -*- coding:utf-8 -*-
3
4 import urllib
5 import urllib2
6 import cookielib
7 import re
8
9 #山东大学绩点运算
10 class SDU:
11
12 def __init__(self):
13 #登录URL

```

```

14 self.loginUrl = 'http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bks_login2.login'
15 #本学期成绩URL
16 self.gradeUrl = 'http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bkscjcx.curscopre'
17 self.cookies = cookielib.CookieJar()
18 self.postdata = urllib.urlencode({
19 'stuid':'201200131012',
20 'pwd':'xxxxxx'
21 })
22 #构建opener
23 self.opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(self.cookies))
24
25 #获取本学期成绩页面
26 def getPage(self):
27 request = urllib2.Request(
28 url = self.loginUrl,
29 data = self.postdata)
30 result = self.opener.open(request)
31 result = self.opener.open(self.gradeUrl)
32 #打印登录内容
33 print result.read().decode('gbk')
34
35 sdu = SDU()
36 sdu.getPage()

```

上面的代码，我们最主要的是增加了

Python

```

result =
self.opener.open(self.gr
1 result = self.opener.open(self.gradeUrl)

```

这句代码，用原来的[opener](#)访问一个本学期成绩的URL即可。运行结果如下

```

<td bgcolor="#EAE2F3"><p align="center"><input type="checkbox" NAME="p_pm" VALUE="013320191012015011288" 微机原理与接口"></p></td>
<td bgcolor="#EAE2F3"><p align="center">0133201910</p></td>
<td bgcolor="#EAE2F3"><p align="center">微机原理与接口</p></td>
<td bgcolor="#EAE2F3"><p align="center">1</p></td>
<td bgcolor="#EAE2F3"><p align="center">2.5</p></td>
<td bgcolor="#EAE2F3"><p align="center">20150112</p></td>
<td bgcolor="#EAE2F3"><p align="center">88</p></td>
<td bgcolor="#EAE2F3"><p align="center">必修</p></td>
</TR>
<TR>
<td bgcolor="#EAE2F3"><p align="center"></p></td>
<td bgcolor="#EAE2F3"><p align="center">0133202360</p></td>
<td bgcolor="#EAE2F3"><p align="center">组成原理课程设计</p></td>
<td bgcolor="#EAE2F3"><p align="center">1</p></td>
<td bgcolor="#EAE2F3"><p align="center">2</p></td>
<td bgcolor="#EAE2F3"><p align="center">20150112</p></td>
<td bgcolor="#EAE2F3"><p align="center">优秀</p></td>
<td bgcolor="#EAE2F3"><p align="center">必修</p></td>
</TR>
<TR>
<td bgcolor="#EAE2F3"><p align="center"><input type="checkbox" NAME="p_pm" VALUE="028100044352015011288" 中国化的马克思主义"></p></td>
<td bgcolor="#EAE2F3"><p align="center">0281000410</p></td>
<td bgcolor="#EAE2F3"><p align="center">中国化的马克思主义</p></td>
<td bgcolor="#EAE2F3"><p align="center">335</p></td>
<td bgcolor="#EAE2F3"><p align="center">3</p></td>

```

恩，本学期成绩的页面已经被我们抓取下来了，接下来用正则表达式提取一下，然后计算学分即可

## 4. 抓取有效信息

接下来我们就把页面内容提取一下，最主要的是学分以及分数了。

平均绩点 =  $\sum (\text{每科学分} * \text{每科分数}) / \text{总学分}$

所以我们把每科的学分以及分数抓取下来就好了，对于有些课打了良好或者优秀等级的，我们不进行抓取。

我们可以发现每一科都是TR标签，然后是一系列的td标签

Python

```

<TR>
<td
1 <td bgcolor="#EAE2F3"><p align="center"><input type="checkbox" NAME="p_pm" VALUE="013320131012015011294 面向对象技术"></p></td>
2 <td bgcolor="#EAE2F3"><p align="center">0133201310</p></td>
3 <td bgcolor="#EAE2F3"><p align="center">面向对象技术</p></td>
4 <td bgcolor="#EAE2F3"><p align="center">1</p></td>
5 <td bgcolor="#EAE2F3"><p align="center">2.5</p></td>
6 <td bgcolor="#EAE2F3"><p align="center">2.5</p></td>

```

```
7 <td bgcolor="#EAE2F3"><p align="center">20150112</p></td>
8 <td bgcolor="#EAE2F3"><p align="center">94</p></td>
9 <td bgcolor="#EAE2F3"><p align="center">必修</p></td>
10 </TR>
```

我们用下面的正则表达式进行提取即可，部分代码如下

Python

```
page = self.getPage()
myItems =
1 page = self.getPage()
2 myItems = re.findall('<TR>.*?<p.*?<p.*?<p.*?<p.*?>(.*?)</p>.*?<p.*?<p.*?>(.*?)</p>.*?</TR>',page,re.S)
3 for item in myItems:
4 self.credit.append(item[0].encode('gbk'))
5 self.grades.append(item[1].encode('gbk'))
```

主要利用了findall方法，这个方法在此就不多介绍了，前面我们已经用过多次了。

得到的学分和分数我们都用列表list进行存储，所以用了 append 方法，每获取到一个信息就把它加进去。

## 5.整理计算最后绩点

恩，像上面那样把学分绩点都保存到列表list中了，所以我们最后用一个公式来计算学分绩点就好了，最后整理后的代码如下：

Python

```
-*- coding: utf-8 -*-
1 # -*- coding: utf-8 -*-
2
3 import urllib
4 import urllib2
5 import cookielib
6 import re
7 import string
8
9 #绩点运算
10 class SDU:
11
12 #类的初始化
13 def __init__(self):
14 #登录URL
15 self.loginUrl = 'http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bks_login2.login'
16 #成绩URL
17 self.gradeUrl = 'http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bkscjcx.curscopre'
18 #CookieJar对象
19 self.cookies = cookielib.CookieJar()
20 #表单数据
21 self.postdata = urllib.urlencode({
22 'stuid':'201200131012',
23 'pwd':'xxxxx'
24 })
25 #构建opener
26 self.opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(self.cookies))
27 #学分list
28 self.credit = []
29 #成绩list
30 self.grades = []
31
32 def getPage(self):
33 req = urllib2.Request(
34 url = self.loginUrl,
35 data = self.postdata)
36 result = self.opener.open(req)
37 result = self.opener.open(self.gradeUrl)
38 #返回本学期成绩页面
39 return result.read().decode('gbk')
40
41 def getGrades(self):
42 #获得本学期成绩页面
43 page = self.getPage()
44 #正则匹配
45 myItems = re.findall('<TR>.*?<p.*?<p.*?<p.*?<p.*?>(.*?)</p>.*?<p.*?<p.*?>(.*?)</p>.*?</TR>',page,re.S)
46 for item in myItems:
47 self.credit.append(item[0].encode('gbk'))
48 self.grades.append(item[1].encode('gbk'))
49 self.getGrade()
50
51 def getGrade(self):
52 #计算总绩点
53 sum = 0.0
54 weight = 0.0
55 for i in range(len(self.credit)):
56 if(self.grades[i].isdigit()):
57 sum += string.atof(self.credit[i])*string.atof(self.grades[i])
58 weight += string.atof(self.credit[i])
59
60 print u"本学期绩点为:",sum/weight
61
62 sdu = SDU()
63 sdu.getGrades()
```

好，最后就会打印输出本学期绩点是多少，小伙伴们最主要的了解上面的编程思路就好。

最主要的内容就是Cookie的使用，模拟登录的功能。

本文思路参考来源：[汪海的爬虫](#)

希望小伙伴们加油，加深一下理解。

**打赏支持我写出更多好文章，谢谢！**

[打赏作者](#)

**打赏支持我写出更多好文章，谢谢！**

任选一种支付方式



1 赞 3 收藏 [2 评论](#)

**关于作者：**[崔庆才](#)



静觅 静静寻觅生活的美好个人站点 [cuiqingcai.com](http://cuiqingcai.com) [个人主页](#) · [我的文章](#) · 13 ·

# Python爬虫实战（2）：百度贴吧帖子

本文作者：[伯乐在线 - 崔庆才](#)。未经作者许可，禁止转载！

欢迎加入伯乐在线[专栏作者](#)。

- [Python爬虫入门（1）：综述](#)
- [Python爬虫入门（2）：爬虫基础了解](#)
- [Python爬虫入门（3）：Urllib库的基本使用](#)
- [Python爬虫入门（4）：Urllib库的高级用法](#)
- [Python爬虫入门（5）：URLError异常处理](#)
- [Python爬虫入门（6）：Cookie的使用](#)
- [Python爬虫入门（7）：正则表达式](#)
- [Python爬虫入门（8）：Beautiful Soup的用法](#)
- [Python爬虫实战（1）：爬取糗事百科段子](#)
- [Python爬虫实战（2）：百度贴吧帖子](#)
- [Python爬虫实战（3）：计算大学本学期绩点](#)
- [Python爬虫实战（3）：计算大学本学期绩点](#)
- [Python爬虫实战（5）：模拟登录淘宝并获取所有订单](#)

大家好，上次我们实验了爬取了糗事百科的段子，那么这次我们来尝试一下爬取百度贴吧的帖子。与上一篇不同的是，这次我们需要用到文件的相关操作。

## 本篇目标

- 1.对百度贴吧的任意帖子进行抓取
- 2.指定是否只抓取楼主发帖内容
- 3.将抓取到的内容分析并保存到文件

### 1.URL格式的确定

首先，我们先观察一下百度贴吧的任意一个帖子。

比如：[http://tieba.baidu.com/p/3138733512?see\\_lz=1&pn=1](http://tieba.baidu.com/p/3138733512?see_lz=1&pn=1)，这是一个关于NBA50大的盘点，分析一下这个地址。

Python

```
http:// 代表资源传输使
用http协议
1 http:// 代表资源传输使用http协议
2 tieba.baidu.com 是百度的二级域名，指向百度贴吧的服务器。
3 /p/3138733512 是服务器某个资源，即这个帖子的地址定位符
4 see_lz和pn是该URL的两个参数，分别代表了只看楼主和帖子页码，等于1表示该条件为真
```

所以我们可以把URL分为两部分，一部分为基础部分，一部分为参数部分。

例如，上面的URL我们划分基础部分是 <http://tieba.baidu.com/p/3138733512>，参数部分是 `?see_lz=1&pn=1`

## 2.页面的抓取

熟悉了URL的格式，那就让我们用urllib2库来试着抓取页面内容吧。上一篇糗事百科我们最后改成了[面向对象](#)的编码方式，这次我们直接尝试一下，定义一个类名叫BDTB(百度贴吧)，一个初始化方法，一个获取页面的方法。

其中，有些帖子我们想指定给程序是否要只看楼主，所以我们把只看楼主的参数初始化放在类的初始化上，即init方法。另外，获取页面的方法我们需要知道一个参数就是帖子页码，所以这个参数的指定我们放在该方法中。

综上，我们初步构建出基础代码如下：

Python

```
author__ = 'CQC'
-*- coding:utf-8 -*-
1 __author__ = 'CQC'
2 # -*- coding:utf-8 -*-
3 import urllib
4 import urllib2
5 import re
6
7 #百度贴吧爬虫类
8 class BDTB:
9
10 #初始化，传入基地址，是否只看楼主的参数
11 def __init__(self,baseUrl,seeLZ):
12 self.baseUrl = baseUrl
13 self.seeLZ = "?see_lz=' + str(seeLZ)
14
15 #传入页码，获取该页帖子的代码
16 def getPage(self,pageNum):
17 try:
18 url = self.baseUrl+ self.seeLZ + '&pn=' + str(pageNum)
```

```

19 request = urllib2.Request(url)
20 response = urllib2.urlopen(request)
21 print response.read()
22 return response
23 except urllib2.URLError, e:
24 if hasattr(e, "reason"):
25 print u"连接百度贴吧失败,错误原因", e.reason
26 return None
27
28 baseURL = 'http://tieba.baidu.com/p/3138733512'
29 bdtb = BDDB(baseURL, 1)
30 bdtb.getPage(1)

```

运行代码，我们可以看到屏幕上打印出了这个帖子第一页楼主发言的所有内容，形式为HTML代码。

The screenshot shows a Python code editor window with the title 'Run bdtb'. The code has been executed, and the resulting HTML output is displayed in the main pane. The output includes navigation links for the forum post, reply counts, and author information. The bottom of the window shows tabs for 'Run', 'TODO', 'Python Console', and 'Terminal'.

```

</p>
</div>
</div><div class="p_thread thread_theme_5" id="thread_theme_5"><div class="l_thread_info"><ul class="l_posts_num">
<li class="l_pager pager_theme_4 pb_list_pager">1

3
4
5
下一页
尾页

<li class="l_reply_num" style="margin-left:8px ">138回复贴, 共5页
<li class="l_reply_num" >, 跳到 <input theme="4" id="jumpPage4" max="page" type="text" class="jump_input_bright" /> 页
<button id="pager_go4" t

<div id="tofrs_up" class="tofrs_up">返回nba吧</div></div><div class="loading-tip" style="display:none;"><spa
<div class="louzhubiaoshi j_louzhubiaoshi" author="明之翼">

</div>
</div> <ul class="p_author">
<li class="icon">
<div class="icon_relative j_user_card" data-field='{"un": "\u660e\u4e4b\u7ffc"}'>

```

### 3.提取相关信息

#### 1) 提取帖子标题

首先，让我们提取帖子的标题。

在浏览器中审查元素，或者按F12，查看页面源代码，我们找到标题所在的代码段，可以发现这个标题的HTML代码是

Python

```

<h1 title="纯原创我心中——的NBA2014-2015赛季">
1 <h1 title="纯原创我心中的NBA2014-2015赛季现役50大" style="width: 396px">纯原创我心中的NBA2014-2015赛季现役50大</h1>

```

所以我们想提取

# 标签中的内容，同时还要指定这个class确定唯一，因为h1标签实在太多啦。

正则表达式如下

Python

```

<h1
class="core_title_txt.*?>
1 <h1 class="core_title_txt.*?>(.*)</h1>

```

所以，我们增加一个获取页面标题的方法

Python

```

#获取帖子标题
def getTitle(self):
1 #获取帖子标题
2 def getTitle(self):
3 page = self.getPage(1)
4 pattern = re.compile('<h1 class="core_title_txt.*?>(.*)</h1>',re.S)
5 result = re.search(pattern,page)
6 if result:
7 #print result.group(1) #测试输出
8 return result.group(1).strip()
9 else:
10 return None

```

## 2) 提取帖子页数

同样地，帖子总页数我们也可以通过分析页面中的共?页来获取。所以我们的获取总页数的方法如下

Python

```
#获取帖子一共有多少页
def getPageNum(self):
 1 #获取帖子一共有多少页
 2 def getPageNum(self):
 3 page = self.getPage(1)
 4 pattern = re.compile('<li class="l_reply_num.*?.*?<span.*?>(.*)',re.S)
 5 result = re.search(pattern,page)
 6 if result:
 7 #print result.group(1) #测试输出
 8 return result.group(1).strip()
 9 else:
 10 return None
```

## 3) 提取正文内容

审查元素，我们可以看到百度贴吧每一层楼的主要内容都在<div id="post\_content\_xxxx"></div>标签里面，所以我们可以写如下的正则表达式

Python

```
<div id="post_content_.*?>
(.*)</div>
1 <div id="post_content_.*?>(.*)</div>
```

相应地，获取页面所有楼层数据的方法可以写成如下方法

Python

```
#获取每一层楼的内容, 传入页面内容
1 #获取每一层楼的内容,传入页面内容
2 def getContent(self,page):
3 pattern = re.compile('<div id="post_content_.*?>(.?)</div>',re.S)
4 items = re.findall(pattern,page)
5 for item in items:
6 print item
```

好，我们运行一下结果看一下

很多媒体都在每赛季之前给球员排个名，我也有这个癖好…………，我会尽量理性的分析球队地位，[a href="http://www.baidu.com/s?wd=个人能力&ie=gbk&tn=SE\\_hldp00990\\_u6vqbx10](http://www.baidu.com/s?wd=个人能力&ie=gbk&tn=SE_hldp00990_u6vqbx10)  
  
开更今天的了！  
| {7}|')
7     #删除超链接标签
8     removeAddr = re.compile('<a.*?>|</a>')
9     #把换行的标签换为\n
10    replaceLine = re.compile('<tr>|<div>|</div>|</p>')
11    #将表格制表<td>替换为\t
12    replaceTD= re.compile('<td>')
13    #把段落开头换为\n加空两格
14    replacePara = re.compile('<p.*?>')
15    #将换行符或双换行符替换为\n
16    replaceBR = re.compile('<br><br>|<br>')
17    #将其余标签剔除
18    removeExtraTag = re.compile('<.*?>')
19    def replace(self,x):
20        x = re.sub(self.removeImg,"",x)
21        x = re.sub(self.removeAddr,"",x)
22        x = re.sub(self.replaceLine,"\n",x)
23        x = re.sub(self.replaceTD,"\\t",x)
24        x = re.sub(self.replacePara,"\\n ",x)
25        x = re.sub(self.replaceBR,"\\n",x)
26        x = re.sub(self.removeExtraTag,"",x)
27        #strip()将前后多余内容删除
28        return x.strip()

```

在使用时，我们只需要初始化一下这个类，然后调用replace方法即可。

现在整体代码是如下这样子的，现在我的代码是写到这样子的

Python

```

import re
1 import re
2
3 #处理页面标签类
4 class Tool:
5     #去除img标签,7位长空格
6     removeImg = re.compile('<img.*?>| {7}|')
7     #删除超链接标签
8     removeAddr = re.compile('<a.*?>|</a>')
9     #把换行的标签换为\n
10    replaceLine = re.compile('<tr>|<div>|</div>|</p>')
11    #将表格制表<td>替换为\t
12    replaceTD= re.compile('<td>')
13    #把段落开头换为\n加空两格
14    replacePara = re.compile('<p.*?>')
15    #将换行符或双换行符替换为\n
16    replaceBR = re.compile('<br><br>|<br>')
17    #将其余标签剔除
18    removeExtraTag = re.compile('<.*?>')
19    def replace(self,x):
20        x = re.sub(self.removeImg,"",x)
21        x = re.sub(self.removeAddr,"",x)
22        x = re.sub(self.replaceLine,"\n",x)
23        x = re.sub(self.replaceTD,"\\t",x)
24        x = re.sub(self.replacePara,"\\n ",x)
25        x = re.sub(self.replaceBR,"\\n",x)
26        x = re.sub(self.removeExtraTag,"",x)
27        #strip()将前后多余内容删除
28        return x.strip()

```

我们尝试一下，重新再看一下效果，这下经过处理之后应该就没问题了，是不是感觉好酸爽！

```

50 惊喜新人王 迈卡威
上赛季数据
篮板 6.2 助攻 6.3 抢断 1.9 盖帽 0.6 失误 3.5 犯规 3 得分 16.7

```

新赛季第50位，我给上赛季的新新人王迈卡威。上赛季迈卡威在彻底重建的76人中迅速掌握了球队，一开始就三双搞定了热火赢得了万千眼球。后来也屡屡有经验的表现，新秀赛季就拿过作为上赛季弱队的老大，迈卡威刷出了不错的数据，但我们静下心来看一看他，还是发现他有很多问题。首先，投篮偏弱刚刚40%的命中率和惨淡的26%的三分命中率肯定是不合格的！加之说完缺点，来说说优点，作为后卫篮板球非常突出，高大的身形能较好的影响对方的出手，也能发现己方的空位球员。突破虽然速度一般，但节奏感不错，大局观也在平均水准之上。提醒就球队地位而言，迈卡威现在是绝对的老大，你想怎么玩就怎么玩，数据你想怎么刷就怎么刷！去年的潜力新人诺尔是蓝领，其他人都可以清退，恩比德还受伤不能打，76人队的战绩怎

Process finished with exit code 0

4) 替换楼层

至于这个问题，我感觉直接提取楼层没什么必要呀，因为只看楼主的话，有些楼层的编号是间隔的，所以我们得到的楼层序号是不连续的，这样我们保存下来也没什么用。

所以可以尝试下面的方法：

- 1.每打印输出一段楼层，写入一行横线来间隔，或者换行符也好。
- 2.试着重新编一个楼层，按照顺序，设置一个变量，每打印出一个结果变量加一，打印出这个变量当做楼层。

这里我们尝试一下吧，看看效果怎样

把getContent方法修改如下

Python

```
#获取每一层楼的内容。
1 #获取每一层楼的内容,传入页面内容
2 def getContent(self,page):
3     pattern = re.compile('<div id="post_content_.*?>(.*)</div>',re.S)
4     items = re.findall(pattern,page)
5     floor = 1
6     for item in items:
7         print floor,u'楼'-----\n
8         print self.tool.replace(item)
9         floor += 1
```

运行一下看看效果

21 楼-----

41 麦迪35秒13分？我有28秒11分的 米尔萨普

上赛季数据

篮板8.5 助攻3.1 抢断1.7 盖帽1.1 失误2.5 犯规2.8 得分17.9

大胖上赛季继续着自己低调而发光发热的表现，老鹰用他取代史密斯是太聪明的决定了，米胖除了快攻的时候跑得没史密斯快，几乎是完爆史密斯，入选全明星是他最大的肯定！米胖的进攻手段很丰富，虽然跑得不快，弹跳一般，臂展也不行，但咱们工人有力量！米胖力量很好，打球速率很快！（这里解释一下，熟悉是变幻动作的速度，米胖的第一步转身的那一防守端，米胖受制于身体条件，对高举高打的打法没有办法。但是米胖作为内线场均1.7的抢断显示着防守端下三路很优秀，防守拆不如前任史密斯，但也不算太坏（起码比布鲁克好多了）缺点，米胖天赋是硬伤，作为一个2轮末尾的球员，打成现在这个样子已经是他的上限了，对比易建联，他靠着自身的努力和锻炼打出了全明星，但是可惜，下赛季霍福德归来依然是老

22 楼-----

再顶一顶，写得好辛苦，别又变成单机了

23 楼-----

话说现在除了一个人都没有人来说高了，低了呢，好奇怪

24 楼-----

篮下终结机 小乔丹

上赛季数据

篮板13.6 助攻0.9 抢断1 盖帽2.5 失误1.5 犯规3.2 得分10.4

在里弗斯到了快船以后，小乔丹获得了新生，从一个无脑弹簧+暴扣男，变成了一个防守覆盖整个两分区域，进攻终结能力犀利的真男人。他现在是名符其实的快船第三巨头上赛季小乔丹两项数据冠绝全联盟，1是场均13.6的篮板球，2是恐怖的67.6%的命中率，你说投得少命中率高就不说了，但这家伙本赛季得分创了生涯新高，第一次上双，命中率还越来越高防守端的小乔丹就更恐怖了，光看数据就知道，篮板王+盖帽第三。还有些是数据无法体现的，比如脚步灵活身高臂长，防守拆效果很好。弹速很快，协防潜力很大，经常飞人的冒，加还是说说缺点，首先那可怜的罚球命中率我就不提了，进攻端还是完全没有自主进攻的能力，作为高大中锋抢下后场篮板后第一时间一传能力也不强（好吧我要求过高了）。防守端太喜欢总的来说，小乔丹用自身的表表现来回报快船给他的大合同，快船三叉戟有望在未来1-2年内打出自己最佳的表现。

25 楼-----

嘿嘿，效果还不错吧，感觉真酸爽！接下来我们完善一下，然后写入文件

4.写入文件

最后便是写入文件的过程，过程很简单，就几句话的代码而已，主要是利用了以下两句

```
file = open("tb.txt", "w")
file.writelines(obj)
```

这里不再赘述，稍后直接贴上完善之后的代码。

5.完善代码

现在我们对代码进行优化，重构，在一些地方添加必要的打印信息，整理如下

Python

```
author__ = 'CQC'
1 __author__ = 'CQC'
2 # -*- coding:utf-8 -*-
3 import urllib
4 import urllib2
5 import re
6
7 #处理页面标签类
8 class Tool:
9     #去除img标签,7位长空格
10    removeImg = re.compile('<img.*?>| {7}|')
11    #删除超链接标签
12    removeAddr = re.compile('<a.*?>|</a>')
```

```

13 #把换行的标签换为\n
14 replaceLine = re.compile('<tr>|<div>|</div>|<p>')
15 #将表格制表<td>替换成|
16 replaceTD= re.compile('<td>')
17 #把段落开头换为\n加空两格
18 replacePara = re.compile('<p.*?>')
19 #将换行符或双换行符替换成\n
20 replaceBR = re.compile('<br>|<br>|<br>')
21 #将其余标签剔除
22 removeExtraTag = re.compile('<.*?>')
23 def replace(self,x):
24     x = re.sub(self.removeImg,"",x)
25     x = re.sub(self.removeAddr,"",x)
26     x = re.sub(self.replaceLine,"\n",x)
27     x = re.sub(self.replaceTD,"|",x)
28     x = re.sub(self.replacePara,"`n` ",x)
29     x = re.sub(self.replaceBR,"`n",x)
30     x = re.sub(self.removeExtraTag,"",x)
31     #strip()将前后多余内容删除
32     return x.strip()
33
34 #百度贴吧爬虫类
35 class BDTB:
36
37     #初始化，传入基地址，是否只看楼主的参数
38     def __init__(self,baseUrl,seeLZ,floorTag):
39         #base链接地址
40         self.baseURL = baseUrl
41         #是否只看楼主
42         self.seeLZ = '?see_lz=' + str(seeLZ)
43         #HTML标签剔除工具类对象
44         self.tool = Tool()
45         #全局file变量，文件写入操作对象
46         self.file = None
47         #楼层标号，初始为1
48         self.floor = 1
49         #默认的标题，如果没有成功获取到标题的话则会用这个标题
50         self.defaultTitle = u"百度贴吧"
51         #是否写入楼分隔符的标记
52         self.floorTag = floorTag
53
54     #传入页码，获取该页帖子的代码
55     def getPage(self,pageNum):
56         try:
57             #构建URL
58             url = self.baseURL+ self.seeLZ + '&pn=' + str(pageNum)
59             request = urllib2.Request(url)
60             response = urllib2.urlopen(request)
61             #返回UTF-8格式编码内容
62             return response.read().decode('utf-8')
63             #无法连接，报错
64         except urllib2.URLError, e:
65             if hasattr(e,"reason"):
66                 print u'连接百度贴吧失败,错误原因',e.reason
67             return None
68
69     #获取帖子标题
70     def getTitle(self,page):
71         #得到标题的正则表达式
72         pattern = re.compile('<h1 class="core_title_txt.*?>(.*)</h1>',re.S)
73         result = re.search(pattern,page)
74         if result:
75             #如果存在，则返回标题
76             return result.group(1).strip()
77         else:
78             return None
79
80     #获取帖子一共有多少页
81     def getPageNum(self,page):
82         #获取帖子页数的正则表达式
83         pattern = re.compile('<li class="l_reply_num.*?</span>.*?<span.*?>(.*)</span>',re.S)
84         result = re.search(pattern,page)
85         if result:
86             return result.group(1).strip()
87         else:
88             return None
89
90     #获取每一层楼的内容,传入页面内容
91     def getContents(self,page):
92         #匹配所有楼层的内容
93         pattern = re.compile('<div id="post_content_.*?>(.*)</div>',re.S)
94         items = re.findall(pattern,page)
95         contents = []
96         for item in items:
97             #将文本进行去除标签处理，同时在前后加入换行符
98             content = "\n"+self.tool.replace(item)+"\n"
99             contents.append(content.encode('utf-8'))
100        return contents
101
102    def setFileTitle(self,title):
103        #如果标题不是为None，即成功获取到标题
104        if title is not None:
105            self.file = open(title + ".txt","w+")
106        else:
107            self.file = open(self.defaultTitle + ".txt","w+")
108
109    def writeData(self,contents):
110        #向文件写入每一楼的信息
111        for item in contents:
112            if self.floorTag == '1':
113                #楼之间的分隔符
114                floorLine = "\n" + str(self.floor) + u"-----\n"

```

```

115     self.file.write(floorLine)
116     self.file.write(item)
117     self.floor += 1
118
119 def start(self):
120     indexPage = self.getPage(1)
121     pageNum = self.getPageNum(indexPage)
122     title = self.getTitle(indexPage)
123     self.setTitle(title)
124     if pageNum == None:
125         print "URL已失效, 请重试"
126         return
127     try:
128         print "该帖子共有" + str(pageNum) + "页"
129         for i in range(1,int(pageNum)+1):
130             print "正在写入第" + str(i) + "页数据"
131             page = self.getPage(i)
132             contents = self.getContent(page)
133             self.writeData(contents)
134     #出现写入异常
135     except IOError,e:
136         print "写入异常, 原因" + e.message
137     finally:
138         print "写入任务完成"
139
140 print u"请输入帖子代号"
141 baseURL = 'http://tieba.baidu.com/p/' + str(raw_input(u'http://tieba.baidu.com/p/'))
142 seeLZ = raw_input("是否只获取楼主发言, 是输入1, 否输入0\n")
143 floorTag = raw_input("是否写入楼层信息, 是输入1, 否输入0\n")
144 bdtb = BDTB(baseURL,seeLZ,floorTag)
145 bdtb.start()

```

现在程序演示如下

请输入帖子代号
<http://tieba.baidu.com/p/3513281888>
 是否只获取楼主发言, 是输入1, 否输入0
₁
 是否写入楼层信息, 是输入1, 否输入0
₁
 该帖子共有7页
 正在写入第1页数据
 正在写入第2页数据
 正在写入第3页数据
 正在写入第4页数据
 正在写入第5页数据
 正在写入第6页数据
 正在写入第7页数据
 写入任务完成

完成之后, 可以查看一下当前目录下多了一个以该帖子命名的txt文件, 内容便是帖子的所有数据。

抓贴吧, 就是这么简单和任性!

打赏支持我写出更多好文章, 谢谢!

打赏作者

打赏支持我写出更多好文章, 谢谢!

任选一种支付方式



2 赞 9 收藏 [20 评论](#)

关于作者：崔庆才



静觅 静静寻觅生活的美好个人站点 cuiqingcai.com [个人主页](#) · [我的文章](#) · 13 ·

Python爬虫实战（1）：爬取糗事百科段子

本文作者：[伯乐在线 - 崔庆才](#)。未经作者许可，禁止转载！

欢迎加入伯乐在线[专栏作者](#)。

- [Python爬虫入门（1）：综述](#)
- [Python爬虫入门（2）：爬虫基础了解](#)
- [Python爬虫入门（3）：Urllib库的基本使用](#)
- [Python爬虫入门（4）：Urllib库的高级用法](#)
- [Python爬虫入门（5）：URLError异常处理](#)
- [Python爬虫入门（6）：Cookie的使用](#)
- [Python爬虫入门（7）：正则表达式](#)
- [Python爬虫入门（8）：Beautiful Soup的用法](#)

大家好，前面入门已经说了那么多基础知识了，下面我们做几个实战项目来挑战一下吧。那么这次为大家带来，Python爬取糗事百科的小段子的例子。

首先，糗事百科大家都听说过吧？糗友们发的搞笑的段子一抓一大把，这次我们尝试一下用爬虫把他们抓取下来。

本篇目标

1. 抓取糗事百科热门段子
2. 过滤带有图片的段子
3. 实现每按一次回车显示一个段子的发布时间，发布人，段子内容，点赞数。

糗事百科是不需要登录的，所以也没必要用到Cookie，另外糗事百科有的段子是附图的，我们把图抓下来图片不便于显示，那么我们就尝试过滤掉有图的段子吧。

好，现在我们尝试抓取一下糗事百科的热门段子吧，每按一次回车我们显示一个段子。

1. 确定URL并抓取页面代码

首先我们确定好页面的URL是 <http://www.qiushibaike.com/hot/page/1>，其中最后一个数字1代表页数，我们可以传入不同的值来获得某一页的段子内容。

我们初步构建如下的代码来打印页面代码内容试试看，先构造最基本的页面抓取方式，看看会不会成功

Python

```
# -*- coding:utf-8 -*-
import urllib
1 # -*- coding:utf-8 -*-
2 import urllib
3 import urllib2
4
5 page = 1
6 url = "http://www.qiushibaike.com/hot/page/" + str(page)
7 try:
8     request = urllib2.Request(url)
9     response = urllib2.urlopen(request)
10    print response.read()
11 except urllib2.URLError, e:
12     if hasattr(e,"code"):
13         print e.code
14     if hasattr(e,"reason"):
15         print e.reason
```

运行程序，哦不，它竟然报错了，真是时运不济，命途多舛啊

Python

```
line 373, in
read_status
1 line 373, in _read_status
2 raise BadStatusLine(line)
3 httplib.BadStatusLine: "
```

好吧，应该是headers验证的问题，我们加上一个headers验证试试看吧，将代码修改如下

Python

```
# -*- coding:utf-8 -*-
import urllib
1 # -*- coding:utf-8 -*-
2 import urllib
3 import urllib2
4
5 page = 1
6 url = "http://www.qiushibaike.com/hot/page/" + str(page)
7 user_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'
8 headers = { 'User-Agent' : user_agent }
9 try:
10    request = urllib2.Request(url,headers = headers)
```

```
11 response = urllib2.urlopen(request)
12 print response.read()
13 except urllib2.URLError, e:
14     if hasattr(e,"code"):
15         print e.code
16     if hasattr(e,"reason"):
17         print e.reason
```

嘿嘿，这次运行终于正常了，打印出了第一页的HTML代码，大家可以运行下代码试试看。在这里运行结果太长就不贴了。

2.提取某一页的所有段子

好，获取了HTML代码之后，我们开始分析怎样获取某一页的所有段子。

首先我们审查元素看一下，按浏览器的F12，截图如下

```
<div class="article block untagged mb15" id="qiushi_tag_100689752">
  <div class="author clearfix">
    <a href="/users/17361709">...</a>
    <a href="/users/17361709">冬天很热，夏天很2</a>
    ::after
  </div>
  <div class="content" title="2015-02-17 07:31:31">
    她开车实在不敢看
  </div>
  <div class="thumb">...</div>
  <div class="stats clearfix">
    <span class="stats-vote">
      <i class="number">2421</i>
      "好笑"
    </span>
    <span class="stats-comments">...</span>
    ::after
  </div>
  <div id="qiushi_counts_100689752" class="stats-buttons bar clearfix">...</div>
</div>
<div class="article block untagged mb15" id="qiushi_tag_100703384">...</div>
<div class="article block untagged mb15" id="qiushi_tag_100706180">...</div>
<div class="article block untagged mb15" id="qiushi_tag_100704848">...</div>
<div class="article block untagged mb15" id="qiushi_tag_100696732">...</div>
<div class="article block untagged mb15" id="qiushi_tag_100696940">...</div>
```

我们可以看到，每一个段子都是`<div class="article block untagged mb15" id="...">...</div>`包裹的内容。

现在我们想获取发布人，发布日期，段子内容，以及点赞的个数。不过另外注意的是，段子有些是带图片的，如果我们想在控制台显示图片是不现实的，所以我们直接把带有图片的段子给它剔除掉，只保存仅含文本的段子。

所以我们加入如下正则表达式来匹配一下，用到的方法是`re.findall`是找寻所有匹配的内容。方法的用法详情可以看前面说的正则表达式的介绍。

好，我们的正则表达式匹配语句书写如下，在原来的基础上追加如下代码

Python

```
content =
1 content = response.read().decode('utf-8')
2 pattern = re.compile('<div.*?class="author.*?>.*?<a.*?</a>.*?<a.*?>(.*)</a>.*?<div.*?class'+ 
3   '=content".*?title="(.?)">(.?)</div>(.?)<div class="stats.*?class="number">(.?)</i>',re.S)
4 items = re.findall(pattern,content)
5 for item in items:
6   print item[0],item[1],item[2],item[3],item[4]
```

现在正则表达式在这里稍作说明

1) `.*?`是一个固定的搭配，`.`和`*`代表可以匹配任意无限多个字符，加上`?`表示使用非贪婪模式进行匹配，也就是我们会尽可能短地做匹配，以后我们还会大量用到`.*?`的搭配。

2) `(.?)`代表一个分组，在这个正则表达式中我们匹配了五个分组，在后面的遍历`item`中，`item[0]`就代表第一个`(.?)`所指代的内容，`item[1]`就代表第二个`(.?)`所指代的内容，以此类推。

3) `re.S`标志代表在匹配时为点任意匹配模式，点`.`也可以代表换行符。

现在我们可以看一下部分运行结果

儒雅男神 2015-02-17 14:34:42

小时候一个一个拆着放的举个爪...

```
<div class="thumb">  
<a href="/article/100705418?list=hot&s=4747301" target="_blank" onclick="_hmt.push(['_trackEvent', 'post', 'click', 'signlePost'])">  
  
</a>  
</div>
```

7093

奇怪的名字啊 2015-02-17 14:49:16

回家的路，你追我赶，回家的心情和窗外的阳光一样灿烂。一路向前，离亲人越来越近了。哪里有爸妈哪里才是家，希望所有糗友的爸爸妈妈都身体健康.....

4803

这是其中的两个段子，分别打印了发布人，发布时间，发布内容，附加图片以及点赞数。

其中，附加图片的内容我把图片代码整体抠了出来，这个对应item[3]，所以我们只需要进一步判断item[3]里面是否含有img这个字样就可以进行过滤了。

好，我们再把上述代码中的for循环改为下面的样子

Python

```
for item in items:  
    haveImg =  
1 for item in items:  
2     haveImg = re.search("img",item[3])  
3     if not haveImg:  
4         print item[0],item[1],item[2],item[4]
```

现在，整体的代码如下

Python

```
# -*- coding:utf-8 -*-  
# -*- coding:utf-8 -*  
1 # -*- coding:utf-8 -*  
2 import urllib  
3 import urllib2  
4 import re  
5  
6 page = 1  
7 url = 'http://www.qiushibaike.com/hot/page/' + str(page)  
8 user_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'  
9 headers = { 'User-Agent' : user_agent }  
10 try:  
11     request = urllib2.Request(url,headers = headers)  
12     response = urllib2.urlopen(request)  
13     content = response.read().decode('utf-8')  
14     pattern = re.compile('<div.*?class="author.*?>.*?<a.*?>.*?<a.*?>(.*)</a>.*?<div.*?class=' +  
15     '=content".*?title="(.*)">(.*)</div>(.*)<div class="stats.*?class="number">(.*)</i>',re.S)  
16     items = re.findall(pattern,content)  
17     for item in items:  
18         haveImg = re.search("img",item[3])  
19         if not haveImg:  
20             print item[0],item[1],item[2],item[4]  
21 except urllib2.URLError, e:  
22     if hasattr(e,"code"):  
23         print e.code  
24     if hasattr(e,"reason"):  
25         print e.reason
```

运行一下看下效果

带着灵魂小漫步 2015-02-17 14:36:34

嘴贱，朋友陪异地女友打电话，我在旁边装女声妩媚，害的他们小两口差点分手。被爆打一顿后还要陪着朋友远赴异地解释清楚，到了门口还要再学一遍当时的女声他女朋友才破涕为笑，

12719 |

奇怪的名字啊 2015-02-17 14:49:16

回家的路，你追我赶，回家的心情和窗外的阳光一样灿烂。一路向前，离亲人越来越近了。哪里有爸妈哪里才是家，希望所有糗友的爸爸妈妈都身体健康.....

4954

请叫我小辛 2015-02-17 14:55:41

前几天在路边看到一司机刚停下车开车门，旁边有条狗就撞车门上了，那司机来了句，哥们大过年的你也碰瓷啊！

5122

蔡蔡不是小菜 2015-02-17 13:35:50

楼主上厕所习惯了马桶，每次完事儿都随手往身后按一下冲水的按钮，今天回到老家蹲坑，完事儿随手往后一按，失去重心，掉坑里了…………到现在他们都不理我…………

14171

CAT&fox 2015-02-17 10:42:11

楼主东北妹纸，早上起床流鼻血，慌忙用纸擦，我爸看到了说，哎呀？流鼻血啦？快别擦了，多浪费纸，赶紧出去站一会，冻住了就不流血了！唉，不说了，还没冻住呢

恩，带有图片的段子已经被剔除啦。是不是很开森？

3.完善交互，设计面向对象模式

好啦，现在最核心的部分我们已经完成啦，剩下的就是修一下边边角角的东西，我们想达到的目的是：

按下回车，读取一个段子，显示出段子的发布人，发布日期，内容以及点赞个数。

另外我们需要设计面向对象模式，引入类和方法，将代码做一下优化和封装，最后，我们的代码如下所示

Python

```
author__ = 'CQC'     
1 __author__ = 'CQC'  
2 # -*- coding:utf-8 -*-  
3 import urllib  
4 import urllib2  
5 import re  
6 import thread  
7 import time  
8  
9 #糗事百科爬虫类  
10 class QSBK:  
11  
12     #初始化方法，定义一些变量  
13     def __init__(self):  
14         self.pageIndex = 1  
15         self.user_agent = 'Mozilla/4.0 (compatible; MSIE 5.5; Windows NT)'  
16         #初始化headers  
17         self.headers = { 'User-Agent': self.user_agent }  
18         #存放段子的变量，每一个元素是每一页的段子们  
19         self.stories = []  
20         #存放程序是否继续运行的变量  
21         self.enable = False  
22     #传入某一页的索引获得页面代码  
23     def getPage(self,pageIndex):  
24         try:  
25             url = 'http://www.qiushibaike.com/hot/page/' + str(pageIndex)  
26             #构建请求的request  
27             request = urllib2.Request(url,headers = self.headers)  
28             #利用urlopen获取页面代码  
29             response = urllib2.urlopen(request)  
30             #将页面转化为UTF-8编码  
31             pageCode = response.read().decode('utf-8')  
32             return pageCode  
33  
34         except urllib2.URLError, e:  
35             if hasattr(e,"reason"):  
36                 print u"连接糗事百科失败,错误原因",e.reason  
37                 return None  
38  
39     #传入某一页代码，返回本页不带图片的段子列表  
40     def getPagettems(self,pageIndex):  
41         pageCode = self.getPage(pageIndex)  
42         if not pageCode:  
43             print "页面加载失败...."  
44             return None  
45         pattern = re.compile('<div.*?class="author.*?>.*?<a.*?>.*?<a.*?>(.*)</a>.*?<div.*?class="content".*?title="(.*?)">(.*?)</div>(.*?)<div class="stats.*?class="number">(.*?)</i>',re.S)  
46         items = re.findall(pattern,pageCode)
```

```

48     #用来存储每页的段子们
49     pageStories = []
50     #遍历正则表达式匹配的信息
51     for item in items:
52         #是否含有图片
53         haveImg = re.search("img",item[3])
54         #如果不含有图片，把它加入list中
55         if not haveImg:
56             #item[0]是一个段子的发布者， item[1]是发布时间,item[2]是内容， item[4]是点赞数
57             pageStories.append([item[0].strip(),item[1].strip(),item[2].strip(),item[4].strip()])
58     return pageStories
59
60 #加载并提取页面的内容，加入到列表中
61 def loadPage(self):
62     #如果当前未看的页数少于2页，则加载新一页
63     if self.enable == True:
64         if len(self.stories) < 2:
65             #获取新一页
66             pageStories = self.getPageItems(self.pageIndex)
67             #将该页的段子存放到全局list中
68             if pageStories:
69                 self.stories.append(pageStories)
70                 #获取完之后页码索引加一，表示下次读取下一页
71             self.pageIndex += 1
72
73 #调用该方法，每次敲回车打印输出一个段子
74 def getOneStory(self,pageStories,page):
75     #遍历一页的段子
76     for story in pageStories:
77         #等待用户输入
78         input = raw_input()
79         #每当输入回车一次，判断一下是否要加载新页面
80         self.loadPage()
81         #如果输入Q则程序结束
82         if input == "Q":
83             self.enable = False
84             return
85         print u"%d页\\t发布人:%s\\t发布时间:%s\\n%s\\n赞:%s\\n" %(page,story[0],story[1],story[2],story[3])
86
87 #开始方法
88 def start(self):
89     print u"正在读取糗事百科,按回车查看新段子，Q退出"
90     #使变量为True，程序可以正常运行
91     self.enable = True
92     #先加载一页内容
93     self.loadPage()
94     #局部变量，控制当前读到了第几页
95     nowPage = 0
96     while self.enable:
97         if len(self.stories)>0:
98             #从全局list中获取一页的段子
99             pageStories = self.stories[0]
100            #当前读到的页数加一
101            nowPage += 1
102            #将全局list中第一个元素删除，因为已经取出
103            del self.stories[0]
104            #输出该页的段子
105            self.getOneStory(pageStories,nowPage)
106
107 spider = QSBK()
108 spider.start()

```

好啦，大家来测试一下吧，点一下回车会输出一个段子，包括发布人，发布时间，段子内容以及点赞数，是不是感觉爽爆了！

我们第一个爬虫实战项目介绍到这里，欢迎大家继续关注，小伙伴们加油！

打赏支持我写出更多好文章，谢谢！

[打赏作者](#)

打赏支持我写出更多好文章，谢谢！

任选一种支付方式



1 赞 24 收藏 [24 评论](#)

关于作者：[崔庆才](#)



静觅 静静寻觅生活的美好个人站点 cuiqingcai.com [个人主页](#) · [我的文章](#) · 13 ·

Python爬虫入门（8）：Beautiful Soup的用法

本文作者：[伯乐在线 - 崔庆才](#)。未经作者许可，禁止转载！
欢迎加入伯乐在线[专栏作者](#)。

- [Python爬虫入门（1）：综述](#)
- [Python爬虫入门（2）：爬虫基础了解](#)
- [Python爬虫入门（3）：Urllib库的基本使用](#)
- [Python爬虫入门（4）：Urllib库的高级用法](#)
- [Python爬虫入门（5）：URLError异常处理](#)
- [Python爬虫入门（6）：Cookie的使用](#)
- [Python爬虫入门（7）：正则表达式](#)
- [Python爬虫入门（8）：Beautiful Soup的用法](#)

上一节我们介绍了正则表达式，它的内容其实还是蛮多的，如果一个正则匹配稍有差池，那可能程序就处在永久的循环之中，而且有的小伙伴们也对写正则表达式的写法用得不熟练，没关系，我们还有一个更强大的工具，叫Beautiful Soup，有了它我们可以很方便地提取出HTML或XML标签中的内容，实在是方便，这一节就让我们一起来感受一下Beautiful Soup的强大吧。

1. Beautiful Soup的简介

简单来说，Beautiful Soup是[python](#)的一个库，最主要的功能是从网页抓取数据。官方解释如下：

Beautiful Soup提供一些简单的、python式的[函数](#)用来处理导航、搜索、修改分析树等功能。它是一个工具箱，通过解析文档为用户提供需要抓取的数据，因为简单，所以不需要多少代码就可以写出一个完整的应用程序。

Beautiful Soup自动将输入文档转换为Unicode编码，输出文档转换为utf-8编码。你不需要考虑编码方式，除非文档没有指定一个编码方式，这时，Beautiful Soup就不能自动识别编码方式了。然后，你仅仅需要说明一下原始编码方式就可以了。

Beautiful Soup已成为和[xml](#)、[html5lib](#)一样出色的[python](#)解释器，为用户灵活地提供不同的解析策略或强劲的速度。

废话不多说，我们来试一下吧~

2. Beautiful Soup 安装

Beautiful Soup 3 目前已经停止开发，推荐在现在的项目中使用Beautiful Soup 4，不过它已经被移植到BS4了，也就是说导入时我们需要 import bs4。所以这里我们用的版本是 Beautiful Soup 4.3.2 (简称BS4)，另外据说 BS4 对 Python3 的支持不够好，不过我用的是 Python2.7.7，如果有小伙伴用的是 Python3 版本，可以考虑下载 BS3 版本。

如果你用的是新版的Debain或Ubuntu,那么可以通过系统的软件包管理来安装，不过它不是最新版本，目前是4.2.1版本

Python

```
sudo apt-get install  
Python-bs4
```

1 sudo apt-get install Python-bs4

如果想安装最新的版本，请直接下载安装包来手动安装，也是十分方便的方法。在这里我安装的是Beautiful Soup 4.3.2

[Beautiful Soup 3.2.1](#)[Beautiful Soup 4.3.2](#)

下载完成之后解压

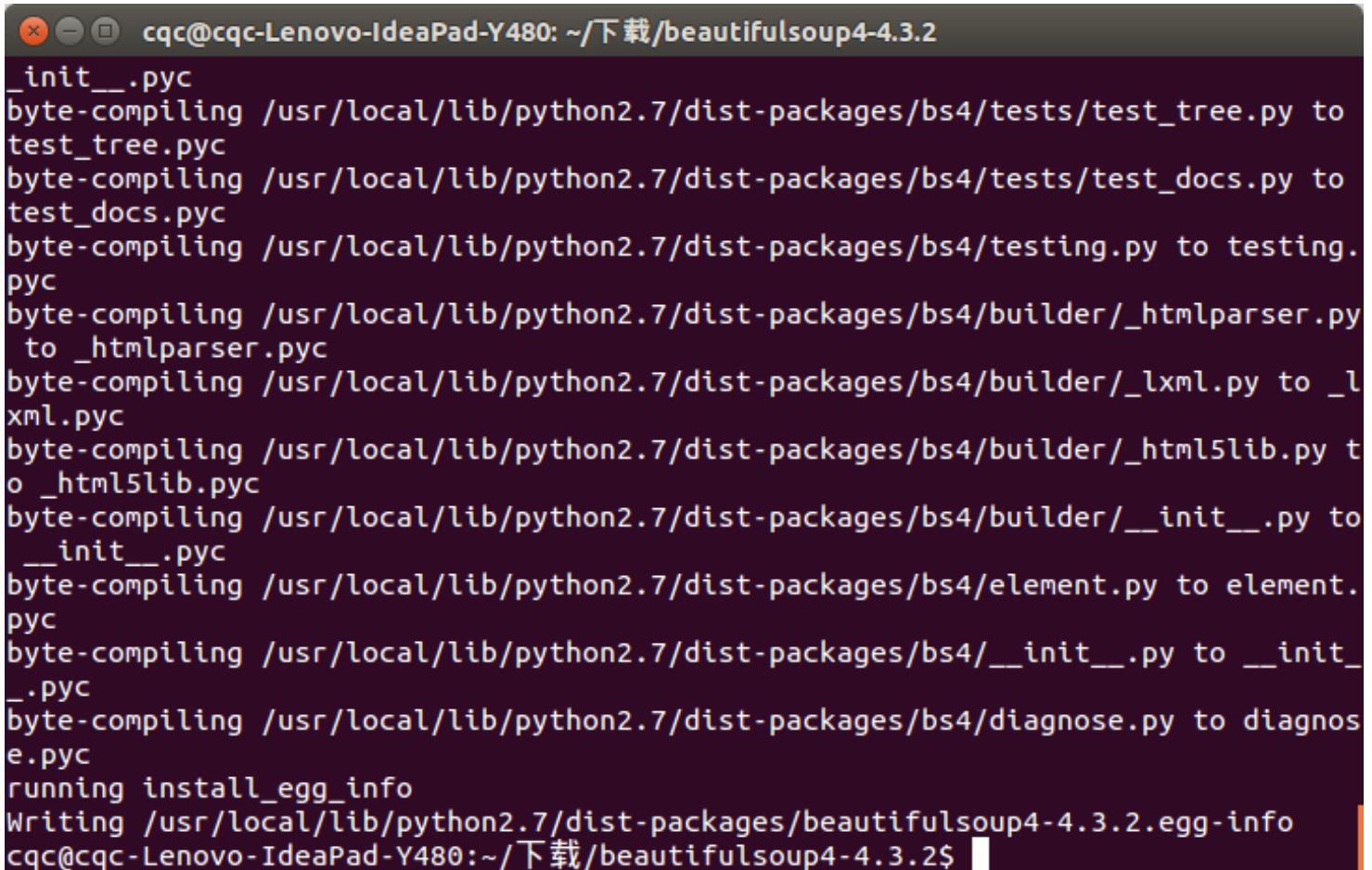
运行下面的命令即可完成安装

Python

```
sudo python setup.py  
install
```

1 sudo python setup.py install

如下图所示，证明安装成功了



cqc@cqc-Lenovo-IdeaPad-Y480: ~/下载/beautifulsoup4-4.3.2

```
_init_.pyc  
byte-compiling /usr/local/lib/python2.7/dist-packages/bs4/tests/test_tree.py to test_tree.pyc  
byte-compiling /usr/local/lib/python2.7/dist-packages/bs4/tests/test_docs.py to test_docs.pyc  
byte-compiling /usr/local/lib/python2.7/dist-packages/bs4/testing.py to testing.pyc  
byte-compiling /usr/local/lib/python2.7/dist-packages/bs4/builder/_htmlparser.py to _htmlparser.pyc  
byte-compiling /usr/local/lib/python2.7/dist-packages/bs4/builder/_lxml.py to _lxml.pyc  
byte-compiling /usr/local/lib/python2.7/dist-packages/bs4/builder/_html5lib.py to _html5lib.pyc  
byte-compiling /usr/local/lib/python2.7/dist-packages/bs4/builder/__init__.py to __init__.pyc  
byte-compiling /usr/local/lib/python2.7/dist-packages/bs4/element.py to element.pyc  
byte-compiling /usr/local/lib/python2.7/dist-packages/bs4/__init__.py to __init__.pyc  
byte-compiling /usr/local/lib/python2.7/dist-packages/bs4/diagnose.py to diagnose.pyc  
running install_egg_info  
Writing /usr/local/lib/python2.7/dist-packages/beautifulsoup4-4.3.2.egg-info  
cqc@cqc-Lenovo-IdeaPad-Y480:~/下载/beautifulsoup4-4.3.2$
```

然后需要安装 lxml

Python

```
sudo apt-get install  
Python-lxml
```

```
1 sudo apt-get install Python-lxml
```

Beautiful Soup支持Python标准库中的HTML解析器,还支持一些第三方的解析器,如果我们不安装它,则 Python 会使用 Python默认的解析器, [lxml](#) 解析器更加强大,速度更快,推荐安装。

3. 开启Beautiful Soup 之旅

在这里先分享官方文档链接,不过内容是有些多,也不够条理,在此本文章做一下整理方便大家参考。

[官方文档](#)

4. 创建 Beautiful Soup 对象

首先必须要导入 bs4 库

Python

```
from bs4 import  
BeautifulSoup
```

```
1 from bs4 import BeautifulSoup
```

我们创建一个字符串,后面的例子我们便会用它来演示

Python

```
html = """  
<html><head>  
1 html = """  
2 <html><head><title>The Dormouse's story</title></head>  
3 <body>  
4 <p class="title" name="dromouse"><b>The Dormouse's story</b></p>  
5 <p class="story">Once upon a time there were three little sisters; and their names were  
6 <a href="http://example.com/elsie" class="sister" id="link1"><!-- Elsie --></a>,  
7 <a href="http://example.com/lacie" class="sister" id="link2">Lacie</a> and  
8 <a href="http://example.com/tillie" class="sister" id="link3">Tillie</a>;  
9 and they lived at the bottom of a well.</p>  
10 <p class="story">...</p>  
11 """"
```

创建 beautifulsoup 对象

Python

```
soup =  
BeautifulSoup(html)
```

```
1 soup = BeautifulSoup(html)
```

另外,我们还可以用本地 HTML 文件来创建对象,例如

Python

```
soup =  
BeautifulSoup(open('ind  
1 soup = BeautifulSoup(open('index.html'))
```

上面这句代码便是将本地 index.html 文件打开，用它来创建 soup 对象

下面我们来打印一下 soup 对象的内容，格式化输出

Python

```
print soup.prettify()  
1 print soup.prettify()
```

Python

```
&lt;html&gt;  
&lt;head&gt;  
1 &lt;html&gt;  
2 &lt;head&gt;  
3 &lt;title&gt;  
4 The Dormouse's story  
5 &lt;/title&gt;  
6 &lt;/head&gt;  
7 &lt;body&gt;  
8 &lt;p class="title" name="dromouse"&gt;  
9 &lt;b&gt;  
10 The Dormouse's story  
11 &lt;/b&gt;  
12 &lt;/p&gt;  
13 &lt;p class="story"&gt;  
14 Once upon a time there were three little sisters; and their names were  
15 &lt;a class="sister" href="http://example.com/elsie" id="link1"&gt;  
16 &lt;!-- Elsie --&gt;  
17 &lt;/a&gt;  
18 ,  
19 &lt;a class="sister" href="http://example.com/lacie" id="link2"&gt;  
20 Lacie  
21 &lt;/a&gt;  
22 and  
23 &lt;a class="sister" href="http://example.com/tillie" id="link3"&gt;  
24 Tillie  
25 &lt;/a&gt;  
26 ;  
27 and they lived at the bottom of a well.  
28 &lt;/p&gt;  
29 &lt;p class="story"&gt;  
30 ...  
31 &lt;/p&gt;  
32 &lt;/body&gt;  
33 &lt;/html&gt;
```

以上便是输出结果，格式化打印出了它的内容，这个函数经常用到，小伙伴们要记好咯。

5. 四大对象种类

Beautiful Soup 将复杂HTML文档转换成一个复杂的树形结构,每个节点都是Python对象,所有对象可以归纳为4种:

- Tag
- NavigableString
- BeautifulSoup
- Comment

下面我们进行一一介绍

(1) Tag

Tag 是什么? 通俗点讲就是 HTML 中的一个个标签, 例如

Python

```
<title>The Dormouse's  
story</title>
```

```
1 <title>The Dormouse's story</title>
```

Python

```
&lt;a class="sister"  
href="http://example.co
```

```
1 &lt;a class="sister" href="http://example.com/elsie" id="link1"&gt;Elsie&lt;/a&gt;
```

上面的 title a 等等 HTML 标签加上里面包括的内容就是 Tag, 下面我们来感受一下怎样用 Beautiful Soup 来方便地获取 Tags

下面每一段代码中注释部分即为运行结果

Python

```
print soup.title  
#<title>The Dormouse's
```

```
1 print soup.title  
2 #<title>The Dormouse's story</title>
```

Python

```
print soup.head  
#<head><title>The
```

```
1 print soup.head  
2 #<head><title>The Dormouse's story</title></head>
```

Python

```
print soup.a  
#<a class="sister"
```

```
1 print soup.a  
2 #<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>
```

Python

```
print soup.p  
#<p class="title"  
  
1 print soup.p  
2 #<p class="title" name="dromouse"><b>The Dormouse's story</b></p>
```

我们可以利用 soup 加标签名轻松地获取这些标签的内容，是不是感觉比正则表达式方便多了？不过有一点是，它查找的是在所有内容中的第一个符合要求的标签，如果要查询所有的标签，我们在后面进行介绍。

我们可以验证一下这些对象的类型

Python

```
print type(soup.a)  
#<class  
  
1 print type(soup.a)  
2 #<class 'bs4.element.Tag'>
```

对于 Tag，它有两个重要的属性，是 name 和 attrs，下面我们分别来感受一下

name

Python

```
print soup.name  
print soup.head.name  
  
1 print soup.name  
2 print soup.head.name  
3 #[document]  
4 #head
```

soup 对象本身比较特殊，它的 name 即为 [document]，对于其他内部标签，输出的值便为标签本身的名称。

attrs

Python

```
print soup.p.attrs  
#{'class': ['title'], 'name':  
  
1 print soup.p.attrs  
2 #{'class': ['title'], 'name': 'dromouse'}
```

在这里，我们把 p 标签的所有属性打印输出了出来，得到的类型是一个字典。

如果我们想要单独获取某个属性，可以这样，例如我们获取它的 class 叫什么

Python

```
print soup.p['class']  
#[title]
```

```
1 print soup.p['class']
2 #['title']
```

还可以这样，利用get方法，传入属性的名称，二者是等价的

Python

```
print soup.p.get('class')
#['title']
```

```
1 print soup.p.get('class')
2 #['title']
```

我们可以对这些属性和内容等等进行修改，例如

Python

```
soup.p['class']="newCla
ss"
```

```
1 soup.p['class']="newClass"
2 print soup.p
3 #<p class="newClass" name="dromouse"><b>The Dormouse's story</b></p>
```

还可以对这个属性进行删除，例如

Python

```
del soup.p['class']
print soup.p
```

```
1 del soup.p['class']
2 print soup.p
3 #<p name="dromouse"><b>The Dormouse's story</b></p>
```

不过，对于修改删除的操作，不是我们的主要用途，在此不做详细介绍了，如果有需要，请查看前面提供的官方文档

(2) NavigableString

既然我们已经得到了标签的内容，那么问题来了，我们要想获取标签内部的文字怎么办呢？很简单，用 .string 即可，例如

Python

```
print soup.p.string
#The Dormouse's story
```

```
1 print soup.p.string
2 #The Dormouse's story
```

这样我们就轻松获取到了标签里面的内容，想想如果用正则表达式要多麻烦。它的类型是一个 NavigableString，翻译过来叫 可以遍历 的字符串，不过我们最好还是称它英文名字吧。

Python

```
print type(soup.p.string) 
```

```
1 print type(soup.p.string)
2 #<class 'bs4.element.NavigableString'>
```

来检查一下它的类型

Python

```
print type(soup.p.string) 
```

```
1 print type(soup.p.string)
2 #<class 'bs4.element.NavigableString'>
```

(3) BeautifulSoup

BeautifulSoup 对象表示的是一个文档的全部内容.大部分时候,可以把它当作 Tag 对象, 是一个特殊的 Tag, 我们可以分别获取它的类型, 名称, 以及属性来感受一下

Python

```
print type(soup.name) 
```

```
1 print type(soup.name)
2 #<type 'unicode'>
3 print soup.name
4 # [document]
5 print soup.attrs
6 #{ } 空字典
```

(4) Comment

Comment 对象是一个特殊类型的 NavigableString 对象, 其实输出的内容仍然不包括注释符号, 但是如果不好好处理它, 可能会对我们的文本处理造成意想不到的麻烦。

我们找一个带注释的标签

Python

```
print soup.a 
print soup.a.string 
```

```
1 print soup.a
2 print soup.a.string
3 print type(soup.a.string)
```

运行结果如下

Python

```
<a class="sister" 
href="http://example.co" 
```

```
1 <a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>
2 Elsie
```

```
3 <class 'bs4.element.Comment'>
```

a [标签](#)里的内容实际上是注释，但是如果我们利用 .string 来输出它的内容，我们发现它已经把注释符号去掉了，所以这可能会给我们带来不必要的麻烦。

另外我们打印输出下它的类型，发现它是一个 Comment 类型，所以，我们在使用前最好做一下判断，判断代码如下

Python

```
if  
type(soup.a.string)==bs  
1 if type(soup.a.string)==bs4.element.Comment:  
2   print soup.a.string
```

上面的代码中，我们首先判断了它的类型，是否为 Comment 类型，然后再进行其他操作，如打印输出。

6. 遍历文档树

(1) 直接子节点

要点：.contents .children 属性

.contents

[tag](#) 的 .content 属性可以将tag的子节点以列表的方式输出

Python

```
print  
soup.head.contents  
1 print soup.head.contents  
2 #[<title>The Dormouse's story</title>]
```

输出方式为列表，我们可以用列表索引来获取它的某一个元素

Python

```
print  
soup.head.contents[0]  
1 print soup.head.contents[0]  
2 #[<title>The Dormouse's story</title>]
```

.children

它返回的不是一个 list，不过我们可以通过[遍历](#)获取所有子节点。

我们打印输出 .children 看一下，可以发现它是一个 list 生成器对象

Python

```
print soup.head.children
```

```
#<listiterator object at
```

```
1 print soup.head.children
2 #<listiterator object at 0x7f71457f5710>
```

我们怎样获得里面的内容呢？很简单，遍历一下就好了，代码及结果如下

Python

```
for child in
soup.body.children:
```

```
1 for child in soup.body.children:
2   print child
```

Python

```
<p class="title"
name="dromouse">
```

```
1 <p class="title" name="dromouse"><b>The Dormouse's story</b></p>
2
3 <p class="story">Once upon a time there were three little sisters; and their names were
4 <a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>,
5 <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> and
6 <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;
7 and they lived at the bottom of a well.</p>
8
9 <p class="story">...</p>
```

(2) 所有子孙节点

知识点：.descendants 属性

.descendants

.contents 和 .children 属性仅包含tag的直接子节点，.descendants 属性可以对所有tag的子孙节点进行递归循环，和 children类似，我们也需要遍历获取其中的内容。

Python

```
for child in
soup.descendants:
```

```
1 for child in soup.descendants:
2   print child
```

运行结果如下，可以发现，所有的节点都被打印出来了，先从最外层的 HTML 标签，其次从 head 标签一个个剥离，以此类推。

Python

```
<html><head>
<title>The Dormouse's
```

```
1 <html><head><title>The Dormouse's story</title></head>
2 <body>
3 <p class="title" name="dromouse"><b>The Dormouse's story</b></p>
```

```

4 <p class="story">Once upon a time there were three little sisters; and their names were
5 <a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>,
6 <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> and
7 <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;
8 and they lived at the bottom of a well.</p>
9 <p class="story">...</p>
10 </body></html>
11 <head><title>The Dormouse's story</title></head>
12 <title>The Dormouse's story</title>
13 The Dormouse's story
14
15 <body>
16 <p class="title" name="dromouse"><b>The Dormouse's story</b></p>
17 <p class="story">Once upon a time there were three little sisters; and their names were
18 <a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>,
19 <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> and
20 <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;
21 and they lived at the bottom of a well.</p>
22 <p class="story">...</p>
23 </body>
24
25 <p class="title" name="dromouse"><b>The Dormouse's story</b></p>
26 <b>The Dormouse's story</b>
27 The Dormouse's story
28
29 <p class="story">Once upon a time there were three little sisters; and their names were
30 <a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>,
31 <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> and
32 <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;
33 and they lived at the bottom of a well.</p>
34 Once upon a time there were three little sisters; and their names were
35
36 <a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>
37 Elsie
38 ,
39
40 <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>
41 Lacie
42 and
43
44 <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>
45 Tillie
46 ;
47 and they lived at the bottom of a well.
48
49 <p class="story">...</p>
50 ...

```

(3) 节点内容

知识点：.string 属性

如果tag只有一个 NavigableString 类型子节点,那么这个tag可以使用 .string 得到子节点。如果一个tag仅有一个子节点,那么这个tag也可以使用 .string 方法,输出结果与当前唯一子节点的 .string 结果相同。

通俗点说就是：如果一个标签里面没有标签了，那么 .string 就会返回标签里面的内容。如果标签里面只有唯一的一个标签了，那么 .string 也会返回最里面的内容。例如

```
print soup.head.string
```

```
#The Dormouse's story
```

```
1 print soup.head.string  
2 #The Dormouse's story  
3 print soup.title.string  
4 #The Dormouse's story
```

如果[tag](#)包含了多个子节点,tag就无法确定, string 方法应该调用哪个子节点的内容, .string 的输出结果是 None

Python

```
print soup.html.string
```

```
# None
```

```
1 print soup.html.string  
2 # None
```

(4) 多个内容

知识点： .strings .stripped_strings 属性

.strings

获取多个内容，不过需要[遍历](#)获取，比如下面的例子

Python

```
for string in
```

```
soup.strings:
```

```
1 for string in soup.strings:  
2     print(repr(string))  
3     # u"The Dormouse's story"  
4     # u'\n\n'  
5     # u"The Dormouse's story"  
6     # u'\n\n'  
7     # u'Once upon a time there were three little sisters; and their names were\n'  
8     # u'Elsie'  
9     # u',\n'  
10    # u'Lacie'  
11    # u' and\n'  
12    # u'Tillie'  
13    # u';\nand they lived at the bottom of a well.'  
14    # u'\n\n'  
15    # u'...'  
16    # u'\n'
```

.stripped_strings

输出的字符串中可能包含了很多空格或空行,使用 .stripped_strings 可以去除多余空白内容

Python

```
for string in
```

```
soup.stripped_strings:
```

```
1 for string in soup.stripped_strings:
```

```
2 print(repr(string))
3 # u"The Dormouse's story"
4 # u"The Dormouse's story"
5 # u'Once upon a time there were three little sisters; and their names were'
6 # u'Elsie'
7 # u','
8 # u'Lacie'
9 # u'and'
10 # u'Tillie'
11 # u';\nand they lived at the bottom of a well.'
12 # u'...'
```

(5) 父节点

知识点： .parent 属性

Python

```
p = soup.p
print p.parent.name
```

```
1 p = soup.p
2 print p.parent.name
3 #body
```

Python

```
content =
soup.head.title.string
```

```
1 content = soup.head.title.string
2 print content.parent.name
3 #title
```

(6) 全部父节点

知识点： .parents 属性

通过元素的 .parents 属性可以递归得到元素的所有父辈节点，例如

Python

```
content =
soup.head.title.string
```

```
1 content = soup.head.title.string
2 for parent in content.parents:
3     print parent.name
```

Python

```
title
head
```

```
1 title
2 head
3 html
4 [document]
```

(7) 兄弟节点

知识点：.next_sibling .previous_sibling 属性

兄弟节点可以理解为和本节点处在统一级的节点，.next_sibling 属性获取了该节点的下一个兄弟节点，.previous_sibling 则与之相反，如果节点不存在，则返回 None

注意：实际文档中的tag的.next_sibling 和.previous_sibling 属性通常是字符串或空白，因为空白或者换行也可以被视作一个节点，所以得到的结果可能是空白或者换行

Python

```
print soup.p.next_sibling  
# 实际该处为空白  
  
1 print soup.p.next_sibling  
2 # 实际该处为空白  
3 print soup.p.prev_sibling  
4 #None 没有前一个兄弟节点，返回 None  
5 print soup.p.next_sibling.next_sibling  
6 #<p class="story">Once upon a time there were three little sisters; and their names were  
7 #<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>,  
8 #<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a> and  
9 #<a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>;  
10 #and they lived at the bottom of a well.</p>  
11 #下一个节点的下一个兄弟节点是我们可以看到的节点
```

(8) 全部兄弟节点

知识点：.next_siblings .previous_siblings 属性

通过.next_siblings 和.previous_siblings 属性可以对当前节点的兄弟节点迭代输出

Python

```
for sibling in  
soup.a.next_siblings:  
  
1 for sibling in soup.a.next_siblings:  
2   print(repr(sibling))  
3   # u'\n'  
4   # <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>  
5   # u' and\n'  
6   # <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>  
7   # u'; and they lived at the bottom of a well.'  
8   # None
```

(9) 前后节点

知识点：.next_element .previous_element 属性

与.next_sibling .previous_sibling 不同，它并不是针对于兄弟节点，而是在所有节点，不分层次
比如 head 节点为

Python

```
<head><title>The  
Dormouse's story</title>
```

```
1 <head><title>The Dormouse's story</title></head>
```

那么它的下一个节点便是 title，它是不分层次关系的

Python

```
print  
soup.head.next_elemen
```

```
1 print soup.head.next_element  
2 #<title>The Dormouse's story</title>
```

(10) 所有前后节点

知识点：.next_elements .previous_elements 属性

通过 .next_elements 和 .previous_elements 的迭代器就可以向前或向后访问文档的解析内容,就好像文档正在被解析一样

Python

```
for element in  
last_a_tag.next_elemen
```

```
1 for element in last_a_tag.next_elements:  
2     print(repr(element))  
3 # u'Tillie'  
4 # u';\nand they lived at the bottom of a well.'  
5 # u'\n\n'  
6 # <p class="story">...</p>  
7 # u'...'  
8 # u'\n'  
9 # None
```

7. 搜索文档树

(1) find_all(name , attrs , recursive , text , **kwargs)

find_all() 方法搜索当前tag的所有tag子节点,并判断是否符合过滤器的条件

1) name 参数

name 参数可以查找所有名字为 name 的tag,字符串对象会被自动忽略掉

A. 传字符串

最简单的过滤器是字符串.在搜索方法中传入一个字符串参数,Beautiful Soup会查找与字符串完整匹配的内容,下面的例子用于查找文档中所有的****标签

Python

```
soup.find_all('b')  
# [<b>The Dormouse's  
1 soup.find_all('b')  
2 # [<b>The Dormouse's story</b>]
```

Python

```
print soup.find_all('a')  
#[<a class="sister"  
 print soup.find_all('a')  
1 #<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>, <a class="sister"  
2 href="http://example.com/lacie" id="link2">Lacie</a>, <a class="sister" href="http://example.com/tillie"  
 id="link3">Tillie</a>]
```

B.传正则表达式

如果传入正则表达式作为参数,Beautiful Soup会通过正则表达式的 `match()` 来匹配内容.下面例子中找出所有以b开头的标签,这表示`<body>`和``标签都应该被找到

Python

```
import re  
for tag in  
1 import re  
2 for tag in soup.find_all(re.compile("^b")):  
3     print(tag.name)  
4 # body  
5 # b
```

C.传列表

如果传入列表参数,Beautiful Soup会将与列表中任一元素匹配的内容返回.下面代码找到文档中所有`<a>`标签和``标签

Python

```
soup.find_all(["a", "b"])  
# [<b>The Dormouse's  
1 soup.find_all(["a", "b"])  
2 # [<b>The Dormouse's story</b>,  
3 # <a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,  
4 # <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,  
5 # <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

D.传 True

`True` 可以匹配任何值,下面代码查找到所有的tag,但是不会返回字符串节点

Python

```
for tag in  
soup.find_all(True):  
1 for tag in soup.find_all(True):  
2     print(tag.name)
```

```
3 # html
4 # head
5 # title
6 # body
7 # p
8 # b
9 # p
10 # a
11 # a
```

E. 传方法

如果没有合适过滤器,那么还可以定义一个方法,方法只接受一个元素参数 [4] ,如果这个方法返回 True 表示当前元素匹配并且被找到,如果不是则返回 False

下面方法校验了当前元素,如果包含 class 属性却不包含 id 属性,那么将返回 True:

Python

```
def has_class_but_no_id(tag):
    return tag.has_attr('class') and not tag.has_attr('id')
```

将这个方法作为参数传入 find_all() 方法,将得到所有<p>标签:

Python

```
soup.find_all(has_class_but_no_id)
soup.find_all(has_class_but_no_id)
# [<p class="title"><b>The Dormouse's story</b></p>,
# <p class="story">Once upon a time there were...</p>,
# <p class="story">...</p>]
```

2) keyword 参数

注意: 如果一个指定名字的参数不是搜索内置的参数名,搜索时会把该参数当作指定名字 tag 的属性来搜索,如果包含一个名字为 id 的参数,Beautiful Soup会搜索每个tag的"id"属性

Python

```
soup.find_all(id='link2')
soup.find_all(id='link2')
# [<a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

如果传入 href 参数,Beautiful Soup会搜索每个tag的"href"属性

Python

```
soup.find_all(href=re.compile("elsie"))
soup.find_all(href=re.compile("elsie"))
```

```
2 # [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>]
```

使用多个指定名字的参数可以同时过滤tag的多个属性

Python

```
soup.find_all(href=re.co  
mpile("elsie"), id='link1')  
1 soup.find_all(href=re.compile("elsie"), id='link1')  
2 # [<a class="sister" href="http://example.com/elsie" id="link1">three</a>]
```

在这里我们想用 class 过滤，不过 class 是 [python](#) 的关键词，这怎么办？加个下划线就可以

Python

```
soup.find_all("a",  
class_="sister")  
1 soup.find_all("a", class_="sister")  
2 # [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,  
3 # <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,  
4 # <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

有些tag属性在搜索不能使用,比如HTML5中的 data-* 属性

Python

```
soup.find_all("a",  
class_="sister")  
1 soup.find_all("a", class_="sister")  
2 # [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,  
3 # <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>,  
4 # <a class="sister" href="http://example.com/tillie" id="link3">Tillie</a>]
```

但是可以通过 `find_all()` 方法的 `attrs` 参数定义一个字典参数来搜索包含特殊属性的tag

Python

```
data_soup.find_all(attrs  
={"data-foo": "value"})  
1 data_soup.find_all(attrs={"data-foo": "value"})  
2 # [<div data-foo="value">foo!</div>]
```

3) text 参数

通过 `text` 参数可以搜搜文档中的字符串内容.与 `name` 参数的可选值一样, `text` 参数接受 字符串 , 正则表达式 , 列表, `True`

Python

```
soup.find_all(text="Elsie")  
1 soup.find_all(text="Elsie")  
2 # [u'Elsie']  
3
```

```
4 soup.find_all(text=["Tillie", "Elsie", "Lacie"])
5 # [u'Elsie', u'Lacie', u'Tillie']
6
7 soup.find_all(text=re.compile("Dormouse"))
8 [u"The Dormouse's story", u"The Dormouse's story"]
```

4) limit 参数

`find_all()` 方法返回全部的搜索结构,如果文档树很大那么搜索会很慢.如果我们不需要全部结果,可以使用 `limit` 参数限制返回结果的数量.效果与SQL中的`limit`关键字类似,当搜索到的结果数量达到 `limit` 的限制时,就停止搜索返回结果.

文档树中有3个tag符合搜索条件,但结果只返回了2个,因为我们限制了返回数量

Python

```
soup.find_all("a",
limit=2)
1 soup.find_all("a", limit=2)
2 # [<a class="sister" href="http://example.com/elsie" id="link1">Elsie</a>,
3 # <a class="sister" href="http://example.com/lacie" id="link2">Lacie</a>]
```

5) recursive 参数

调用tag的 `find_all()` 方法时,Beautiful Soup会检索当前[tag](#)的所有子孙节点,如果只想搜索tag的直接子节点,可以使用参数 `recursive=False` .

一段简单的文档:

Python

```
<html>
<head>
1 <html>
2 <head>
3 <title>
4 The Dormouse's story
5 </title>
6 </head>
7 ...
```

是否使用 `recursive` 参数的搜索结果:

Python

```
soup.html.find_all("title")
# [<title>The
1 soup.html.find_all("title")
2 # [<title>The Dormouse's story</title>]
3
4 soup.html.find_all("title", recursive=False)
5 # []
```

(2) `find(name , attrs , recursive , text , **kwargs)`

它与 `find_all()` 方法唯一的区别是 `find_all()` 方法的返回结果是值包含一个元素的列表,而 `find()` 方法直接返回结果

(3) `find_parents()` `find_parent()`

`find_all()` 和 `find()` 只搜索当前节点的所有子节点,孙子节点等. `find_parents()` 和 `find_parent()` 用来搜索当前节点的父辈节点,搜索方法与普通tag的搜索方法相同,搜索文档搜索文档包含的内容

(4) `find_next_siblings()` `find_next_sibling()`

这2个方法通过 `.next_siblings` 属性对当 tag 的所有后面解析的兄弟 tag 节点进行迭代,`find_next_siblings()` 方法返回所有符合条件的后面的兄弟节点,`find_next_sibling()` 只返回符合条件的后面的第一个tag节点

(5) `find_previous_siblings()` `find_previous_sibling()`

这2个方法通过 `.previous_siblings` 属性对当前 tag 的前面解析的兄弟 tag 节点进行迭代,
`find_previous_siblings()` 方法返回所有符合条件的前面的兄弟节点, `find_previous_sibling()` 方法返回第一个符合条件的前面的兄弟节点

(6) `find_all_next()` `find_next()`

这2个方法通过 `.next_elements` 属性对当前 tag 的之后的 tag 和字符串进行迭代,`find_all_next()` 方法返回所有符合条件的节点, `find_next()` 方法返回第一个符合条件的节点

(7) `find_all_previous()` 和 `find_previous()`

这2个方法通过 `.previous_elements` 属性对当前节点前面的 `tag` 和字符串进行迭代, `find_all_previous()` 方法返回所有符合条件的节点, `find_previous()` 方法返回第一个符合条件的节点

注: 以上 (2) (3) (4) (5) (6) (7) 方法参数用法与 `find_all()` 完全相同, 原理均类似, 在此不再赘述。

8.CSS选择器

我们在写 CSS 时, 标签名不加任何修饰, 类名前加点, id名前加 #, 在这里我们也可以利用类似的方法来筛选元素, 用到的方法是 `soup.select()`, 返回类型是 `list`

(1) 通过标签名查找

Python

```
print soup.select('title')  
#[<title>The Dormouse's  
1 print soup.select('title')  
2 #[<title>The Dormouse's story</title>]
```

Python

```
print soup.select('a')
#[<a class="sister">
  print soup.select('a')
1 # [<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>, <a class="sister"
2 href="http://example.com/lacie" id="link2">Lacie</a>, <a class="sister" href="http://example.com/tillie"
  id="link3">Tillie</a>]
```

Python

```
print soup.select('b')
#[<b>The Dormouse's
  print soup.select('b')
1 # [<b>The Dormouse's story</b>]
```

(2) 通过类名查找

Python

```
print
soup.select('.sister')
  print soup.select('.sister')
1 # [<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>, <a class="sister"
2 href="http://example.com/lacie" id="link2">Lacie</a>, <a class="sister" href="http://example.com/tillie"
  id="link3">Tillie</a>]
```

(3) 通过 id 名查找

Python

```
print
soup.select('#link1')
  print soup.select('#link1')
1 # [<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>]
```

(4) 组合查找

组合查找即和写 class 文件时，标签名与类名、id名进行的组合原理是一样的，例如查找 p 标签中，id 等于 link1 的内容，二者需要用空格分开

Python

```
print soup.select('p
#link1')
  print soup.select('p #link1')
1 # [<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>]
```

直接子标签查找

Python

```
print soup.select("head
> title")
```

```
1 print soup.select("head > title")
2 #[<title>The Dormouse's story</title>]
```

(5) 属性查找

查找时还可以加入属性元素，属性需要用中括号括起来，注意属性和[标签](#)属于同一节点，所以中间不能加空格，否则会无法匹配到。

Python

```
print soup.select("head
> title")
```



```
1 print soup.select("head > title")
2 #[<title>The Dormouse's story</title>]
```

Python

```
print
soup.select('a[href="http
"]')
```



```
1 print soup.select('a[href="http://example.com/elsie"]')
2 #[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>]
```

同样，属性仍然可以与上述查找方式组合，不在同一节点的空格隔开，同一节点的不加空格

Python

```
print soup.select('p
a[href="http://example.c
"]')
```



```
1 print soup.select('p a[href="http://example.com/elsie"]')
2 #[<a class="sister" href="http://example.com/elsie" id="link1"><!-- Elsie --></a>]
```

好，这就是另一种与 `find_all` 方法有异曲同工之妙的查找方法，是不是感觉很方便？

总结

本篇内容比较多，把 BeautifulSoup 的方法进行了大部分整理和总结，不过这还不算完全，仍然有 BeautifulSoup 的修改删除功能，不过这些功能用得比较少，只整理了查找提取的方法，希望对大家有帮助！小伙伴们加油！

熟练掌握了 BeautifulSoup，一定会给你带来太多方便，加油吧！

打赏支持我写出更多好文章，谢谢！

[打赏作者](#)

打赏支持我写出更多好文章，谢谢！

任选一种支付方式

微信扫一扫转账



向崔庆才转账

赞赏伯乐在线的文章

¥2.00

支付宝扫一扫，向我付款



赞赏你发在伯乐在线的文章

¥2.00

1 赞 13 收藏 [3 评论](#)

关于作者：[崔庆才](#)



静觅 静静寻觅生活的美好个人站点 cuiqingcai.com [个人主页](#) · [我的文章](#) · 13 ·

Python爬虫入门（7）：正则表达式

本文作者：[伯乐在线 - 崔庆才](#)。未经作者许可，禁止转载！

欢迎加入伯乐在线[专栏作者](#)。

- [Python爬虫入门（1）：综述](#)
- [Python爬虫入门（2）：爬虫基础了解](#)
- [Python爬虫入门（3）：Urllib库的基本使用](#)
- [Python爬虫入门（4）：Urllib库的高级用法](#)
- [Python爬虫入门（5）：URLError异常处理](#)
- [Python爬虫入门（6）：Cookie的使用](#)
- [Python爬虫入门（7）：正则表达式](#)
- [Python爬虫入门（8）：Beautiful Soup的用法](#)

在前面我们已经搞定了怎样获取页面的内容，不过还差一步，这么多杂乱的代码夹杂文字我们怎样把它提取出来整理呢？下面就开始介绍一个十分强大的工具，正则表达式！

1.了解正则表达式

正则表达式是对字符串操作的一种逻辑公式，就是用事先定义好的一些特定字符、及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。

正则表达式是用来匹配字符串非常强大的工具，在其他编程语言中同样有正则表达式的概念，Python同样不例外，利用了正则表达式，我们想要从返回的页面内容提取出我们想要的内容就易如反掌了。

正则表达式的大致匹配过程是：

- 1.依次拿出表达式和文本中的字符比较，
- 2.如果每一个字符都能匹配，则匹配成功；一旦有匹配不成功的字符则匹配失败。
- 3.如果表达式中有量词或边界，这个过程会稍微有一些不同。

2.正则表达式的语法规则

下面是Python中正则表达式的一些匹配规则，图片资料来自CSDN

语法	说明	表达式实例	完整匹配的字符串
字符			
一般字符	匹配自身	abc	abc
.	匹配任意除换行符"\n"外的字符。 在DOTALL模式中也能匹配换行符。	a.c	abc
\	转义字符，使后一个字符改变原来的意思。 如果字符串中有字符*需要匹配，可以使用*或者字符集[*]。	a\.c a\\c	a.c a\c
[...]	字符集（字符类）。对应的位置可以是字符集中任意字符。 字符集中的字符可以逐个列出，也可以给出范围，如[abc]或 [a-c]。第一个字符如果是^则表示取反，如[^abc]表示不是 abc的其他字符。 所有的特殊字符在字符集中都失去其原有的特殊含义。在字 符集中如果要使用]、-或^，可以在前面加上反斜杠，或把] 、-放在第一个字符，把^放在非第一个字符。	a[bcd]e	abe ace ade

预定义字符集 (可以与在字符集[...]中)

\d	数字 : [0-9]	a\dc	a1c
\D	非数字 : [^\d]	a\Dc	abc
\s	空白字符 : [<空格>\t\r\n\f\v]	a\sc	a c
\S	非空白字符 : [^\s]	a\Sc	abc
\w	单词字符 : [A-Za-z0-9_]	a\wc	abc
\W	非单词字符 : [^\w]	a\Wc	a c

数量词 (用在字符或(...)之后)

*	匹配前一个字符0或无限次。	abc*	ab abccc
+	匹配前一个字符1次或无限次。	abc+	abc abccc
?	匹配前一个字符0次或1次。	abc?	ab abc
{m}	匹配前一个字符m次。	ab{2}c	abbc
{m,n}	匹配前一个字符m至n次。 m和n可以省略：若省略m，则匹配0至n次；若省略n，则匹配m至无限次。	ab{1,2}c	abc abbc
*? +? ?? {m,n}?	使 * + ? {m,n}变成非贪婪模式。	示例将在下文中介绍。	

边界匹配 (不消耗待匹配字符串中的字符)

^	匹配字符串开头。 在多行模式中匹配每一行的开头。	^abc	abc
\$	匹配字符串末尾。 在多行模式中匹配每一行的末尾。	abc\$	abc
\A	仅匹配字符串开头。	\Aabc	abc
\Z	仅匹配字符串末尾。	abc\Z	abc
\b	匹配\w和\W之间。	a\b!bc	a!bc
\B	[^\b]	a\Bbc	abc

逻辑、分组

	代表左右表达式任意匹配一个。 它总是先尝试匹配左边的表达式，一旦成功匹配则跳过匹配右边的表达式。 如果 没有被包括在()中，则它的范围是整个正则表达式。	abc def	abc def
--	--	---------	------------

(...)	被括起来的表达式将作为分组，从表达式左边开始每遇到一个分组的左括号'('，编号+1。 另外，分组表达式作为一个整体，可以后接数量词。表达式中的 仅在该组中有效。	(abc){2}	abcabc
(?P<name>...)	分组，除了原有的编号外再指定一个额外的别名。	(?P<id>abc){2}	abcabc

\<number>	引用编号为<number>的分组匹配到的字符串。	(\d)abc\1	1abc1 5abc5
(?P=name)	引用别名为<name>的分组匹配到的字符串。	(?P<id>\d)abc(?P=id)	1abc1 5abc5

特殊构造 (不作为分组)

(?:...)	(...)的不分组版本，用于使用' '或后接数量词。	(?:abc){2}	abcabc
---------	----------------------------	------------	--------

(?i)Lmsux)	iLmsux的每个字符代表一个匹配模式，只能用在正则表达式的开头，可选多个。匹配模式将在下文中介绍。	(?i)abc	AbC
(?#...)	#后的內容将作为注释被忽略。	abc(?#comment)123	abc123
(?=...)	之后的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	a(?=\\d)	后面是数字的a
(?!...)	之后的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	a(?!\\d)	后面不是数字的a
(?<=...)	之前的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	(?<=\\d)a	前面是数字的a
(?<!...)	之前的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	(?<!\\d)a	前面不是数字的a
(?i(id/name) yes-pattern no-pattern)	如果编号为id/别名为name的组匹配到字符，则需要匹配 yes-pattern，否则需要匹配no-pattern。 no-patern可以省略。	(\\d)abc(?(1)\\d abc)	1abc2 abcabc

3. 正则表达式相关注解

(1) 数量词的贪婪模式与非贪婪模式

正则表达式通常用于在文本中查找匹配的字符串。Python里数量词默认是贪婪的（在少数语言里也可能是默认非贪婪），总是尝试匹配尽可能多的字符；非贪婪的则相反，总是尝试匹配尽可能少的字符。例如：正则表达式“ab*”如果用于查找“abbbc”，将找到“abbb”。而如果使用非贪婪的数量词“ab*？”，将找到“a”。

注：我们一般使用非贪婪模式来提取。

(2) 反斜杠问题

与大多数编程语言相同，正则表达式里使用“\”作为转义字符，这就可能造成反斜杠困扰。假如你需要匹配文本中的字符“\”，那么使用编程语言表示的正则表达式里将需要4个反斜杠“\\\\”：前两个和后两个分别用于在编程语言里转义成反斜杠，转换成两个反斜杠后再在正则表达式里转义成一个反斜杠。

Python里的原生字符串很好地解决了这个问题，这个例子中的正则表达式可以使用r“\\”表示。同样，匹配一个数字的“\\d”可以写成“r\\d”。有了原生字符串，妈妈也不用担心是不是漏写了反斜杠，写出来的表达式也更直观勒。

4. Python Re模块

Python自带了re模块，它提供了对正则表达式的支持。主要用到的方法列举如下

Python

```
#返回pattern对象
re.compile(string[,flag])
1 #返回pattern对象
2 re.compile(string[,flag])
3 #以下为匹配所用函数
4 re.match(pattern, string[, flags])
5 re.search(pattern, string[, flags])
6 re.split(pattern, string[, maxsplit])
7 re.findall(pattern, string[, flags])
8 re.finditer(pattern, string[, flags])
9 re.sub(pattern, repl, string[, count])
10 re.subn(pattern, repl, string[, count])
```

在介绍这几个方法之前，我们先来介绍一下pattern的概念，pattern可以理解为一个匹配模式，那么我们怎么获得这个匹配模式呢？很简单，我们需要利用re.compile方法就可以。例如

Python

```
pattern =  
re.compile(r'hello')  
1 pattern = re.compile(r'hello')
```

在参数中我们传入了原生字符串对象，通过compile方法编译生成一个pattern对象，然后我们利用这个对象来进行进一步的匹配。

另外大家可能注意到了另一个参数 flags，在这里解释一下这个参数的含义：

参数[flag](#)是匹配模式，取值可以使用按位或运算符'|'表示同时生效，比如re.I | re.M。

可选值有：

Python

- re.I(全拼: IGNORECASE): 忽略大小写
- re.M(全拼: MULTILINE): 多行模式，改变'^'和'\$'的行为 (参见上图)
- re.S(全拼: DOTALL): 点任意匹配模式，改变'.'的行为
- re.L(全拼: LOCALE): 使预定字符类 \w \W \b \B \s \S 取决于当前区域设定
- re.U(全拼: UNICODE): 使预定字符类 \w \W \b \B \s \S \d \D 取决于unicode定义的字符属性
- re.X(全拼: VERBOSE): 详细模式。这个模式下正则表达式可以是多行，忽略空白字符，并可以加入注释。

在刚才所说的另外几个方法例如 re.match 里我们就需要用到这个pattern了，下面我们一一介绍。

注：以下七个方法中的[flags](#)同样是代表匹配模式的意思，如果在pattern生成时已经指明了flags，那么在下面的方法中就不需要传入这个参数了。

(1) re.match(pattern, string[, flags])

这个方法将会从string（我们要匹配的字符串）的开头开始，尝试匹配pattern，一直向后匹配，如果遇到无法匹配的字符，立即返回 None，如果匹配未结束已经到达string的末尾，也会返回None。两个结果均表示匹配失败，否则匹配pattern成功，同时匹配终止，不再对 string 向后匹配。下面我们通过一个例子理解一下

Python

```
author__ = 'CQC'  
# -*- coding: utf-8 -*-  
1 author__ = 'CQC'  
2 # -*- coding: utf-8 -*-  
3  
4 # 导入re模块  
5 import re  
6  
7 # 将正则表达式编译成Pattern对象，注意hello前面的r的意思是“原生字符串”  
8 pattern = re.compile(r'hello')  
9  
10 # 使用re.match匹配文本，获得匹配结果，无法匹配时将返回None  
11 result1 = re.match(pattern, 'hello')  
12 result2 = re.match(pattern, 'heloo CQC!')  
13 result3 = re.match(pattern, 'heoo CQC!')  
14 result4 = re.match(pattern, 'heoo CQC!')  
15  
16 # 如果1匹配成功
```

```
17 if result1:  
18     # 使用Match获得分组信息  
19     print result1.group()  
20 else:  
21     print '1匹配失败！'  
22  
23 #如果2匹配成功  
24 if result2:  
25     # 使用Match获得分组信息  
26     print result2.group()  
27 else:  
28     print '2匹配失败！'  
29  
30 #如果3匹配成功  
31 if result3:  
32     # 使用Match获得分组信息  
33     print result3.group()  
34 else:  
35     print '3匹配失败！'  
36  
37 #如果4匹配成功  
38 if result4:  
39     # 使用Match获得分组信息  
40     print result4.group()  
41 else:  
42     print '4匹配失败！'
```

运行结果

Python

```
hello  
hello  
  
1 hello  
2 hello  
3 3匹配失败！  
4 hello
```

匹配分析

- 1.第一个匹配，pattern正则表达式为'hello'，我们匹配的目标字符串string也为hello，从头至尾完全匹配，匹配成功。
- 2.第二个匹配，string为helloo CQC，从string头开始匹配pattern完全可以匹配，pattern匹配结束，同时匹配终止，后面的o CQC不再匹配，返回匹配成功的信息。
- 3.第三个匹配，string为heloo CQC，从string头开始匹配pattern，发现到'o'时无法完成匹配，匹配终止，返回None
- 4.第四个匹配，同第二个匹配原理，即使遇到了空格符也不会受影响。

我们还看到最后打印出了result.[group\(\)](#)，这个是什么意思呢？下面我们说一下关于match对象的属性和方法。Match对象是一次匹配的结果，包含了很多关于此次匹配的信息，可以使用Match提供的可读属性或方法来获取这些信息。

属性：

- 1.string: 匹配时使用的文本。
- 2.re: 匹配时使用的Pattern对象。
- 3.pos: 文本中正则表达式开始搜索的索引。值与Pattern.match()和Pattern.seach()方法的同名参数相同。
- 4.endpos: 文本中正则表达式结束搜索的索引。值与Pattern.match()和Pattern.seach()方法的同名参

数相同。

5.lastindex: 最后一个被捕获的分组在文本中的索引。如果没有被捕获的分组，将为None。

6.lastgroup: 最后一个被捕获的分组的别名。如果这个分组没有别名或者没有被捕获的分组，将为None。

方法：

1.group([group1, ...]):

获得一个或多个分组截获的字符串；指定多个参数时将以元组形式返回。group1可以使用编号也可以使用别名；编号0代表整个匹配的子串；不填写参数时，返回group(0)；没有截获字符串的组返回None；截获了多次的组返回最后一次截获的子串。

2.groups([default]):

以元组形式返回全部分组截获的字符串。相当于调用group(1,2,...last)。default表示没有截获字符串的组以这个值替代，默认为None。

3.groupdict([default]):

返回以有别名的组的别名为键、以该组截获的子串为值的字典，没有别名的组不包含在内。default含义同上。

4.start([group]):

返回指定的组截获的子串在string中的起始索引（子串第一个字符的索引）。group默认值为0。

5.end([group]):

返回指定的组截获的子串在string中的结束索引（子串最后一个字符的索引+1）。group默认值为0。

6.span([group]):

返回(start(group), end(group))。

7.expand(template):

将匹配到的分组代入template中然后返回。template中可以使用\id或\g、\g引用分组，但不能使用编号0。\\id与\\g是等价的；但\\10将被认为是第10个分组，如果你想表达\\1之后是字符'0'，只能使用\\g0。

下面我们用一个例子来体会一下

Python

```
# -*- coding: utf-8 -*-
#一个简单的match实例

1 # -*- coding: utf-8 -*-
2 #一个简单的match实例
3
4 import re
5 # 匹配如下内容： 单词+空格+单词+任意字符
6 m = re.match(r'(\w+) (\w+)(?P.*)', 'hello world!')
7
8 print "m.string:", m.string
9 print "m.re:", m.re
10 print "m.pos:", m.pos
11 print "m.endpos:", m.endpos
12 print "m.lastindex:", m.lastindex
13 print "m.lastgroup:", m.lastgroup
14 print "m.group():", m.group()
15 print "m.group(1,2):", m.group(1, 2)
16 print "m.groups():", m.groups()
17 print "m.groupdict():", m.groupdict()
18 print "m.start(2):", m.start(2)
19 print "m.end(2):", m.end(2)
20 print "m.span(2):", m.span(2)
21 print r'm.expand(r"\g \g\g"):', m.expand(r'\2 \1\3')
22
23 ### output ###
24 # m.string: hello world!
25 # m.re:
26 # m.pos: 0
27 # m.endpos: 12
```

```
28 # m.lastindex: 3
29 # m.lastgroup: sign
30 # m.group(1,2): ('hello', 'world')
31 # m.groups(): ('hello', 'world', '!')
32 # m.groupdict(): {'sign': '!'}
33 # m.start(2): 6
34 # m.end(2): 11
35 # m.span(2): (6, 11)
36 # m.expand(r'\2 \1\3'): world hello!
```

(2) re.search(pattern, string[, flags])

search方法与match方法极其类似，区别在于match()函数只检测re是不是在string的开始位置匹配，search()会扫描整个string查找匹配，match () 只有在0位置匹配成功的话才有返回，如果不是开始位置匹配成功的话，match()就返回None。同样，search方法的返回对象同样match()返回对象的方法和属性。我们用一个例子感受一下

Python

```
#导入re模块
import re
1 #导入re模块
2 import re
3
4 # 将正则表达式编译成Pattern对象
5 pattern = re.compile(r'world')
6 # 使用search()查找匹配的子串，不存在能匹配的子串时将返回None
7 # 这个例子中使用match()无法成功匹配
8 match = re.search(pattern,'hello world!')
9 if match:
10    # 使用Match获得分组信息
11    print match.group()
12 ### 输出 ###
13 # world
```

(3) re.split(pattern, string[, maxsplit])

按照能够匹配的子串将string分割后返回列表。maxsplit用于指定最大分割次数，不指定将全部分割。我们通过下面的例子感受一下。

Python

```
import re
1 import re
2
3 pattern = re.compile(r'\d+')
4 print re.split(pattern,'one1two2three3four4')
5
6 ### 输出 ###
7 # ['one', 'two', 'three', 'four', '']
```

(4) re.findall(pattern, string[, flags])

搜索string，以列表形式返回全部能匹配的子串。我们通过这个例子来感受一下

Python

```
import re
1 import re
```

```
2
3 pattern = re.compile(r'\d+')
4 print re.findall(pattern,'one1two2three3four4')
5
6 ### 输出 ###
7 # [1, '2', '3', '4']
```

(5) **re.finditer(pattern, string[, flags])**

搜索string，返回一个顺序访问每一个匹配结果（Match对象）的迭代器。我们通过下面的例子来感受一下

Python

```
import re
1 import re
2
3 pattern = re.compile(r'\d+')
4 for m in re.finditer(pattern,'one1two2three3four4'):
5     print m.group(),
6
7 ### 输出 ###
8 # 1 2 3 4
```

(6) **re.sub(pattern, repl, string[, count])**

使用repl替换string中每一个匹配的子串后返回替换后的字符串。

当repl是一个字符串时，可以使用\1或\g、\g引用分组，但不能使用编号0。

当repl是一个方法时，这个方法应当只接受一个参数（Match对象），并返回一个字符串用于替换（返回的字符串中不能再引用分组）。

count用于指定最多替换次数，不指定时全部替换。

Python

```
import re
1 import re
2
3 pattern = re.compile(r'(\w+) (\w+)')
4 s = 'i say, hello world!'
5
6 print re.sub(pattern,r'\2 \1', s)
7
8 def func(m):
9     return m.group(1).title() + ' ' + m.group(2).title()
10
11 print re.sub(pattern,func, s)
12
13 ### output ###
14 # say i, world hello!
15 # I Say, Hello World!
```

(7) **re.subn(pattern, repl, string[, count])**

返回 (sub(repl, string[, count]), 替换次数)。

Python

```
import re
```

```
1 import re
2
3 pattern = re.compile(r'(\w+) (\w+)')
4 s = 'i say, hello world!'
5
6 print re.subn(pattern,r'\2 \1', s)
7
8 def func(m):
9     return m.group(1).title() + ' ' + m.group(2).title()
10
11 print re.subn(pattern,func, s)
12
13 ### output ###
14 # ('say i, world hello!', 2)
15 # ('I Say, Hello World!', 2)
```

5.Python Re模块的另一种使用方式

在上面我们介绍了7个工具方法，例如match，search等等，不过调用方式都是re.match，re.search的方式，其实还有另外一种调用方式，可以通过pattern.match，pattern.search调用，这样调用便不用将pattern作为第一个参数传入了，大家想怎样调用皆可。

函数API列表

Python

match(string[, pos[, endpos]])
1 match(string[, pos[, endpos]]) re.match(pattern, string[, flags]) 2 search(string[, pos[, endpos]]) re.search(pattern, string[, flags]) 3 split(string[, maxsplit]) re.split(pattern, string[, maxsplit]) 4 findall(string[, pos[, endpos]]) re.findall(pattern, string[, flags]) 5 finditer(string[, pos[, endpos]]) re.finditer(pattern, string[, flags]) 6 sub(repl, string[, count]) re.sub(pattern, repl, string[, count]) 7 subn(repl, string[, count]) re.sub(pattern, repl, string[, count])

具体的调用方法不必详说了，原理都类似，只是参数的变化不同。小伙伴们尝试一下吧~

小伙伴们加油，即使这一节看得云里雾里的也没关系，接下来我们会通过一些实战例子来帮助大家熟练掌握正则表达式的。

参考文章：此文章部分内容出自 [CNBlogs](#)

打赏支持我写出更多好文章，谢谢！

[打赏作者](#)

打赏支持我写出更多好文章，谢谢！

任选一种支付方式

微信扫一扫转账



向崔庆才转账

赞赏伯乐在线的文章

¥2.00

支付宝扫一扫，向我付款



¥2.00

赞赏你发在伯乐在线的文章

1 赞 9 收藏 [3 评论](#)

关于作者：[崔庆才](#)



静觅 静静寻觅生活的美好个人站点 cuiqingcai.com [个人主页](#) · [我的文章](#) · [13](#) ·

Python爬虫入门（6）：Cookie的使用

本文作者：[伯乐在线 - 崔庆才](#)。未经作者许可，禁止转载！

欢迎加入伯乐在线[专栏作者](#)。

- [Python爬虫入门（1）：综述](#)
- [Python爬虫入门（2）：爬虫基础了解](#)
- [Python爬虫入门（3）：Urllib库的基本使用](#)
- [Python爬虫入门（4）：Urllib库的高级用法](#)
- [Python爬虫入门（5）：URLError异常处理](#)
- [Python爬虫入门（6）：Cookie的使用](#)
- [Python爬虫入门（7）：正则表达式](#)
- [Python爬虫入门（8）：Beautiful Soup的用法](#)

大家好哈，上一节我们研究了一下爬虫的异常处理问题，那么接下来我们一起来看一下Cookie的使用。

为什么要使用Cookie呢？

Cookie，指某些网站为了辨别用户身份、进行session跟踪而储存在用户本地终端上的数据（通常经过加密）

比如说有些网站需要登录后才能访问某个页面，在登录之前，你想抓取某个页面内容是不允许的。那么我们可以利用Urllib2库保存我们登录的Cookie，然后再抓取其他页面就达到目的了。

在此之前呢，我们必须先介绍一个opener的概念。

1.Opener

当你获取一个URL你使用一个opener(一个urllib2.OpenerDirector的实例)。在前面，我们都是使用的默认的opener，也就是urlopen。它是一个特殊的opener，可以理解成opener的一个特殊实例，传入的参数仅仅是url，[data](#)，[timeout](#)。

如果我们要用到Cookie，只用这个opener是不能达到目的的，所以我们需要创建更一般的opener来实现对Cookie的设置。

2.Cookielib

cookielib模块的主要作用是提供可存储cookie的对象，以便于与urllib2模块配合使用来访问Internet资源。Cookielib模块非常强大，我们可以利用本模块的CookieJar类的对象来捕获cookie并在后续连接请求时重新发送，比如可以实现模拟登录功能。该模块主要的对象有CookieJar、FileCookieJar、MozillaCookieJar、LWPCookieJar。

它们的关系：CookieJar ——派生——> FileCookieJar ——派生——> MozillaCookieJar和LWPCookieJar

1) 获取Cookie保存到变量

首先，我们先利用CookieJar对象实现获取cookie的功能，[存储](#)到变量中，先来感受一下

Python

```
import urllib2
import cookielib
1 import urllib2
2 import cookielib
3 #声明一个CookieJar对象实例来保存cookie
4 cookie = cookielib.CookieJar()
5 #利用urllib2库的HTTPCookieProcessor对象来创建cookie处理器
6 handler=urllib2.HTTPCookieProcessor(cookie)
7 #通过handler来构建opener
8 opener = urllib2.build_opener(handler)
9 #此处的open方法同urllib2的urlopen方法，也可以传入request
10 response = opener.open('http://www.baidu.com')
11 for item in cookie:
12     print 'Name = '+item.name
13     print 'Value = '+item.value
```

我们使用以上方法将cookie保存到变量中，然后打印出了cookie中的值，运行结果如下

Python

```
Name = BAIDUID  
1 Name = BAIDUID  
2 Value = B07B663B645729F11F659C02AAE65B4C:FG=1  
3 Name = BAIDUPSID  
4 Value = B07B663B645729F11F659C02AAE65B4C  
5 Name = H_PS_PSSID  
6 Value = 12527_11076_1438_10633  
7 Name = BDSVRTM  
8 Value = 0  
9 Name = BD_HOME  
10 Value = 0
```

2) 保存Cookie到文件

在上面的方法中，我们将cookie保存到了cookie这个变量中，如果我们想将cookie保存到文件中该怎么做呢？这时，我们就要用到

FileCookieJar这个对象了，在这里我们使用它的子类MozillaCookieJar来实现Cookie的保存

Python

```
import cookielib  
import urllib2  
1 import cookielib  
2 import urllib2  
3  
4 #设置保存cookie的文件，同级目录下的cookie.txt  
5 filename = 'cookie.txt'  
6 #声明一个MozillaCookieJar对象实例来保存cookie，之后写入文件  
7 cookie = cookielib.MozillaCookieJar(filename)  
8 #利用urllib2库的HTTPCookieProcessor对象来创建cookie处理器  
9 handler = urllib2.HTTPCookieProcessor(cookie)  
10 #通过handler来构建opener  
11 opener = urllib2.build_opener(handler)  
12 #创建一个请求，原理同urllib2的urlopen  
13 response = opener.open("http://www.baidu.com")  
14 #保存cookie到文件  
15 cookie.save(ignore_discard=True, ignore_expires=True)
```

关于最后save方法的两个参数在此说明一下：

官方解释如下：

ignore_discard: save even cookies set to be discarded.

ignore_expires: save even cookies that have expiredThe file is overwritten if it already exists

由此可见，ignore_discard的意思是即使cookies将被丢弃也将它保存下来，ignore_expires的意思是如果在该文件中cookies已经存在，则覆盖原文本写入，在这里，我们将这两个全部设置为True。运行之后，cookies将被保存到cookie.txt文件中，我们查看一下内容，附图如下

```
# Netscape HTTP Cookie File
# http://curl.haxx.se/rfc/cookie_spec.html
# This is a generated file! Do not edit.

.baidu.com TRUE / FALSE 3571490773 BAIDUID 1618055EDDD7B032E1AE449E9B001492:FG=1
.baidu.com TRUE / FALSE 3571490773 BAIDUPSID 1618055EDDD7B032E1AE449E9B001492
.baidu.com TRUE / FALSE H_PS_PSSID 1447
www.baidu.com FALSE / FALSE BDSVRTM 0
www.baidu.com FALSE / FALSE BD_HOME 0
```

3) 从文件中获取Cookie并访问

那么我们已经做到把Cookie保存到文件中了，如果以后想使用，可以利用下面的方法来读取cookie并访问网站，感受一下

Python

```
import cookielib
import urllib2

1 import cookielib
2 import urllib2
3
4 #创建MozillaCookieJar实例对象
5 cookie = cookielib.MozillaCookieJar()
6 #从文件中读取cookie内容到变量
7 cookie.load('cookie.txt', ignore_discard=True, ignore_expires=True)
8 #创建请求的request
9 req = urllib2.Request("http://www.baidu.com")
10 #利用urllib2的build_opener方法创建一个opener
11 opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cookie))
12 response = opener.open(req)
13 print response.read()
```

设想，如果我们的 cookie.txt 文件中保存的是某个人登录百度的cookie，那么我们提取出这个cookie文件内容，就可以用以上方法模拟这个人的账号登录百度。

4) 利用cookie模拟网站登录

下面我们以我们学校的[教育系统](#)为例，利用cookie实现模拟登录，并将cookie信息保存到文本文件中，来感受一下cookie大法吧！

注意：密码我改了啊，别偷偷登录本宫的选课系统 o(╯□╰)o

Python

```
import urllib
import urllib2

1 import urllib
2 import urllib2
3 import cookielib
4
5 filename = 'cookie.txt'
6 #声明一个MozillaCookieJar对象实例来保存cookie，之后写入文件
7 cookie = cookielib.MozillaCookieJar(filename)
8 opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cookie))
9 postdata = urllib.urlencode({
10 'stuid':'201200131012',
11 'pwd':'23342321',
12 })
13 #登录教务系统的URL
14 loginUrl = 'http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bks_login2.login'
15 #模拟登录，并把cookie保存到变量
16 result = opener.open(loginUrl,postdata)
```

```
17 #保存cookie到cookie.txt中
18 cookie.save(ignore_discard=True, ignore_expires=True)
19 #利用cookie请求访问另一个网址，此网址是成绩查询网址
20 gradeUrl = 'http://jwxt.sdu.edu.cn:7890/pls/wwwbks/bkscjcx.curscope'
21 #请求访问成绩查询网址
22 result = opener.open(gradeUrl)
23 print result.read()
```

以上程序的原理如下

创建一个带有cookie的opener，在访问登录的URL时，将登录后的cookie保存下来，然后利用这个cookie来访问其他网址。

如登录之后才能查看的成绩查询呀，本学期课表呀等等网址，模拟登录就这么实现啦，是不是很酷炫？

好，小伙伴们要加油哦！我们现在可以顺利获取网站信息了，接下来就是把网站里面有效内容提取出来，下一节我们去会会正则表达式！

打赏支持我写出更多好文章，谢谢！

[打赏作者](#)

打赏支持我写出更多好文章，谢谢！

任选一种支付方式



1 赞 4 收藏 2 评论

关于作者：崔庆才



静觅 静静寻觅生活的美好个人站点 cuiqingcai.com [个人主页](#) · [我的文章](#) · 13 ·

Python爬虫入门（5）：URLError异常处理

本文作者：[伯乐在线 - 崔庆才](#)。未经作者许可，禁止转载！
欢迎加入伯乐在线[专栏作者](#)。

- [Python爬虫入门（1）：综述](#)
- [Python爬虫入门（2）：爬虫基础了解](#)
- [Python爬虫入门（3）：Urllib库的基本使用](#)
- [Python爬虫入门（4）：Urllib库的高级用法](#)
- [Python爬虫入门（5）：URLError异常处理](#)
- [Python爬虫入门（6）：Cookie的使用](#)
- [Python爬虫入门（7）：正则表达式](#)
- [Python爬虫入门（8）：Beautiful Soup的用法](#)

大家好，本节在这里主要说的是URLError还有HTTPError，以及对它们的一些处理。

1.URLError

首先解释下URLError可能产生的原因：

- 网络无连接，即本机无法上网
- 连接不到特定的服务器
- 服务器不存在

在代码中，我们需要用try-except语句来包围并捕获相应的异常。下面是一个例子，先感受下它的风骚

Python

```
import urllib2  
  
1 import urllib2  
2  
3 request = urllib2.Request('http://www.xxxxx.com')  
4 try:  
5     urllib2.urlopen(request)  
6 except urllib2.URLError, e:  
7     print e.reason
```

我们利用了urlopen方法访问了一个不存在的网址，运行结果如下：

Python

```
[Errno 11004] getaddrinfo  
failed  
1 [Errno 11004] getaddrinfo failed
```

它说明了错误代号是11004，错误原因是getaddrinfo failed

2. HTTPError

HTTPError是URLError的子类，在你利用urlopen方法发出一个请求时，服务器上都会对应一个应答对象response，其中它包含一个数字“状态码”。举个例子，假如response是一个“重定向”，需[定位](#)到别的地址获取文档，urllib2将对此进行处理。

其他不能处理的，urlopen会产生一个HTTPError，对应相应的状态吗，HTTP状态码表示HTTP协议所返回的响应的状态。下面将状态码归结如下：

100：继续 客户端应当继续发送请求。客户端应当继续发送请求的剩余部分，或者如果请求已经完成，忽略这个响应。

101：转换协议 在发送完这个响应最后的空行后，[服务器](#)将会切换到在Upgrade消息头中定义的那些协议。只有在切换新的协议更有好处的时候才应该采取类似措施。

102：继续处理 由WebDAV（RFC 2518）扩展的状态码，代表处理将被继续执行。

200：请求成功 处理方式：获得响应的内容，进行处理

201：请求完成，结果是创建了新资源。新创建资源的URI可在响应的实体中得到 处理方式：[爬虫](#)中不会遇到

202：请求被接受，但处理尚未完成 处理方式：阻塞等待

204：服务器端已经实现了请求，但是没有返回新的信息。如果客户是[用户代理](#)，则无须为此更新自身的文档视图。 处理方式：丢弃

300：该状态码不被HTTP/1.0的应用程序直接使用，只是作为3XX类型回应的默认解释。存在多个可用的被请求资源。 处理方式：若程序中能够处理，则进行进一步处理，如果程序中不能处理，则丢弃

301：请求到的资源都会分配一个永久的URL，这样就可以在将来通过该URL来访问此资源 处理方式：重定向到分配的URL

302：请求到的资源在一个不同的URL处临时保存 处理方式：重定向到临时的URL

304：请求的资源未更新 处理方式：丢弃

400：非法请求 处理方式：丢弃

401：未授权 处理方式：丢弃

403：禁止 处理方式：丢弃

404：没有找到 处理方式：丢弃

500：服务器内部错误 服务器遇到了一个未曾预料的状况，导致了它无法完成对请求的处理。一般来说，这个问题都会在[服务器端](#)的源代码出现错误时出现。

501：服务器无法识别 服务器不支持当前请求所需要的某个功能。当服务器无法识别请求

的方法，并且无法支持其对任何资源的请求。

502：错误网关 作为网关或者[代理](#)工作的服务器尝试执行请求时，从上游服务器接收到无效的响应。

503：服务出错 由于临时的[服务器](#)维护或者过载，服务器当前无法处理请求。这个状况是临时的，并且将在一段时间以后恢复。

HTTPError实例产生后会有一个code属性，这就是是服务器发送的相关错误号。

因为urllib2可以为你处理重定向，也就是3开头的代号可以被处理，并且100-299范围的号码指示成功，所以你只能看到400-599的错误号码。

下面我们写一个例子来感受一下，捕获的异常是HTTPError，它会带有一个code属性，就是错误代号，另外我们又打印了reason属性，这是它的父类URLError的属性。

Python

```
import urllib2  
1 import urllib2  
2  
3 req = urllib2.Request('http://blog.csdn.net/cqcre')  
4 try:  
5     urllib2.urlopen(req)  
6 except urllib2.HTTPError, e:  
7     print e.code  
8     print e.reason
```

运行结果如下

Python

```
403  
Forbidden
```

```
1 403  
2 Forbidden
```

错误代号是403，错误原因是Forbidden，说明服务器禁止访问。

我们知道，HTTPError的父类是URLError，根据[编程](#)经验，父类的异常应当写到子类异常的后面，如果子类捕获不到，那么可以捕获父类的异常，所以上述的代码可以这么改写

Python

```
import urllib2  
1 import urllib2  
2  
3 req = urllib2.Request('http://blog.csdn.net/cqcre')  
4 try:  
5     urllib2.urlopen(req)  
6 except urllib2.HTTPError, e:  
7     print e.code
```

```
8 except urllib2.URLError, e:  
9     print e.reason  
10 else:  
11     print "OK"
```

如果捕获到了HTTPError，则输出code，不会再处理URLError异常。如果发生的不是HTTPError，则会去捕获URLError异常，输出错误原因。

另外还可以加入 hasattr属性提前对属性进行判断，代码改写如下

Python

```
import urllib2  
  
1 import urllib2  
2  
3 req = urllib2.Request('http://blog.csdn.net/cqcre')  
4 try:  
5     urllib2.urlopen(req)  
6 except urllib2.URLError, e:  
7     if hasattr(e,"code"):  
8         print e.code  
9     if hasattr(e,"reason"):  
10        print e.reason  
11 else:  
12     print "OK"
```

首先对异常的属性进行判断，以免出现属性输出报错的现象。

以上，就是对URLError和HTTPError的相关介绍，以及相应的错误处理办法，小伙伴们加油！

打赏支持我写出更多好文章，谢谢！

[打赏作者](#)

打赏支持我写出更多好文章，谢谢！

任选一种支付方式

微信扫一扫转账



向崔庆才转账

赞赏伯乐在线的文章

¥2.00

支付宝扫一扫，向我付款



赞赏你发在伯乐在线的文章

¥2.00

2 赞 4 收藏 [5 评论](#)

关于作者：[崔庆才](#)



静觅 静静寻觅生活的美好个人站点 cuiqingcai.com [个人主页](#) · [我的文章](#) · [13](#) ·