

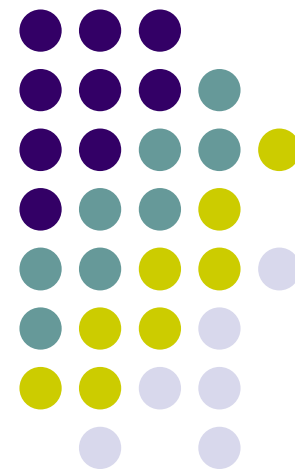


第6章 大量数据的表示

Gig Data Representation

申丽萍

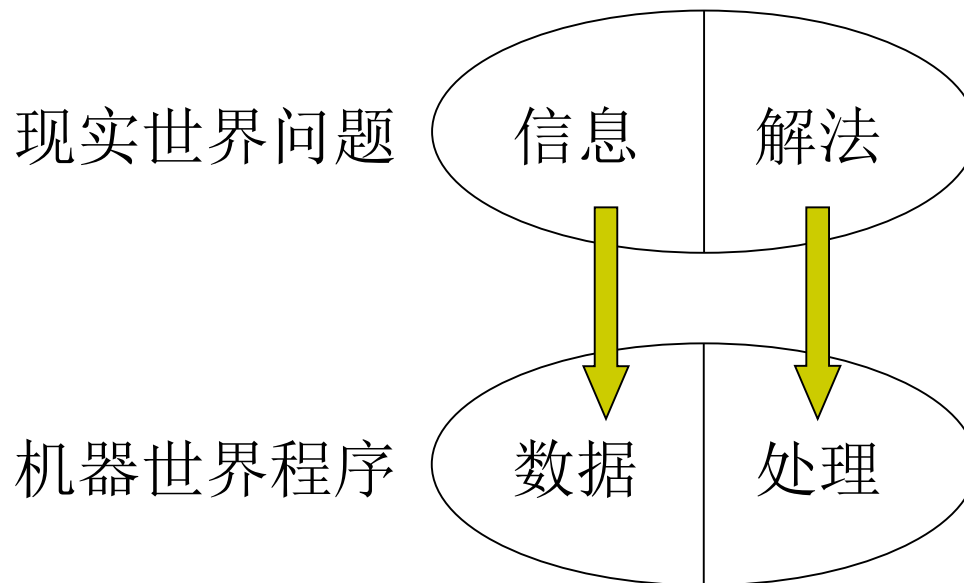
lpshen@sjtu.edu.cn



数据处理



- 计算机=数据处理机器
- 计算=数据+处理
- 问题求解=信息表示+解法表示





第6章 大量数据的表示和处理

- 简单数据类型（回忆）
 - 数值数据类型: `int`, `long`, `float`
 - 布尔类型: `bool`
 - 字符串类型: `str`
- 复杂数据类型
 - 集合体类型
 - 有序集合体: 字符串、列表、元组
 - 无序集合体: 集合、字典
 - 文件
 - 高级数据结构: 链表、堆栈、队列



复杂数据

- 简单数据一般指单个数据，并且没有内部结构，不可分割。
- 复杂数据正相反，可在两方面呈现复杂性：
 - 数量多，即待处理的数据是由大量相互关联的成员数据组成的；
 - 有内部结构，即数据在内部由若干分量组成，每个分量本身可能又由更小的分量组成。对于大量数据，可以用集合体数据类型来表示；对于数据的内部结构，可以用数据结构来刻画。
- 数据结构: 研究如何将大量相关数据按特定的逻辑结构组织起来，以及如何高效地处理这些数据。



第6章 大量数据的表示和处理

- 简单数据类型（回忆）
 - 数值数据类型
 - 布尔类型
 - 字符串类型
- 复杂数据类型
 - 集合体类型
 - 有序集合体：字符串、列表、元组
 - 无序集合体：集合、字典
 - 文件
 - 高级数据结构：链表、堆栈、队列

数据集合体



- 很多程序都需要处理大量类似数据的集合。
 - 文档中的大量单词,
 - 海量的Internet数据
 - 实验得到的数据如DNA序列,
- 原子类型: `int`, `long`, `float`, `bool`都是“原子”值。
- `str`类型是由多个字符组成的序列。
- 有没有一个对象能包含很多数据?
 - 例如: `range(5) = [0,1,2,3,4]`
 - 例如: `string.split("This is it.") = ['This','is','it']`
- 集合体类型: 能够用一个变量来存储大量数据的类型, 包括列表、元组、字典和文件。

序列



- 大量数据排列而形成的有序集合体称为序列（*sequence*）
- Python中的字符串、列表和元组数据类型都是序列
- Python序列其实都是以面向对象方式实现的，因此对序列的处理可以通过对序列对象的方法进行调用而实现。
- 序列支持比较运算。序列s和t的大小按字典序确定：首先通过比较s[0]与t[0]来决定大小，相等时再比较s[1]和t[1]，依次类推。这就是说，两个序列相等当且仅当它们的对应位置上的成员都相等，并且长度相同。

序列通用操作



方法	含义
<code>s1 + s2</code>	序列s1和s2联接成一个序列
<code>s * n</code> 或 <code>n * s</code>	序列s复制n次，即n个s联接
<code>s[i]</code>	序列s中索引为i的成员
<code>s[i:j]</code>	序列s中索引从i到j的子序列
<code>s[i:j:k]</code>	序列s中索引从i到j间隔为k的子序列
<code>len(s)</code>	序列s的长度
<code>min(s)</code>	序列s中的最小数据项
<code>max(s)</code>	序列s中的最大数据项
<code>x in s</code>	检测x是否在序列s中，返回True或False
<code>x not in s</code>	检测x是否不在序列s中，返回True或False

有序集合体: str, list, tuple



- 大量数据按照次序排列的集合体称为序列(sequence)
- 序列通用的操作:
 - 索引: `s[i]`, `s[i:j]`
 - 检测`x`是否在序列: `x in s`, `x not in s`
 - 排序: `sorted(s, cmp, key, reverse)` vs `list.sort(cmp, key, reverse)`
 - 例子: `wordCount=[("a",10),("data",15),("we",20),("the", 16),("key",6)]`

```
>>> wordCount=[("a",10),("data",15),("we",20),("the", 16),("key",6)]
>>> sorted(wordCount,lambda x,y:cmp(x[1],y[1]))
[('key', 6), ('a', 10), ('data', 15), ('the', 16), ('we', 20)]
>>> wordCount
[('a', 10), ('data', 15), ('we', 20), ('the', 16), ('key', 6)]
>>> sorted(wordCount,key=lambda x:x[1])
[('key', 6), ('a', 10), ('data', 15), ('the', 16), ('we', 20)]
>>> wordCount
[('a', 10), ('data', 15), ('we', 20), ('the', 16), ('key', 6)]
>>> import operator
>>> sorted(wordCount,key=operator.itemgetter(1))
[('key', 6), ('a', 10), ('data', 15), ('the', 16), ('we', 20)]
>>> wordCount
[('a', 10), ('data', 15), ('we', 20), ('the', 16), ('key', 6)]
>>> wordCount.sort(key=lambda x:x[1],reverse=True)
>>> wordCount
[('we', 20), ('the', 16), ('data', 15), ('a', 10), ('key', 6)]
```

```
>>> t1=(10,5,8,3,1,7)
>>> t1.sort()
```

```
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    t1.sort()
AttributeError: 'tuple' object has no attribute 'sort'
>>> sorted(t1)
[1, 3, 5, 7, 8, 10]
```

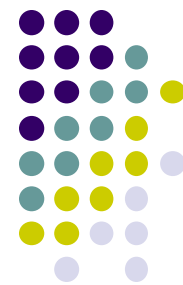


序列的面向对象方法



字符串对象方法	string库函数	含义
s.capitalize()	capitalize(s)	s 首字母大写
s.center(width)	center(s,width)	s 扩展到给定宽度且 s 居中
s.count(sub)	count(s,sub)	sub 在 s 中出现的次数
s.find(sub)	find(s,sub)	sub 在 s 中首次出现的位置
s.ljust(width)	ljust(s,width)	s 扩展到给定宽度且 s 居左
s.lower()	lower(s)	将 s 的所有字母改成小写
s.lstrip()	lstrip(s)	将 s 的所有前导空格删去
s.replace(old,new)	replace(s,old,new)	将 s 中所有 old 替换成 new
s.rfind(sub)	rfind(s,sub)	sub 在 s 中最后一次出现的位置
s.rjust(width)	rjust(s,width)	s 扩展到给定宽度且 s 居右
s.rstrip()	rstrip(s)	将 s 的所有尾部空格删去
s.split()	split(s)	将 s 拆分成子串的列表
s.upper()	upper(s)	将 s 的所有字母改成大写

列表类型



- 列表(List):是一种数据集合体.
 - 是数据项的有序序列
 - 例如: `[]`, `[1, 2, 3]` `[1, "two", 3.0, True]`
- 数据整体用一个名字表示
 - 例如: `seq = ['abc', 2, True]`
- 数据成员通过位置索引引用
 - 例如: `seq[2]=True`
- 列表特点:
 - 列表成员可以由任意类型的数据构成, 不要求各成员具有相同类型;
 - 列表长度是不定的, 随时可以增加和删除成员。
 - 列表是可以修改的, 修改方式包括向列表添加成员、从列表删除成员以及对列表的某个成员进行修改。

列表的修改



修改方式	含义
<code>a[i] = x</code>	将列表a中索引为i的成员改为x
<code>a[i:j] = b</code>	将列表a中索引从i到j（不含）的片段改为列表b
<code>del a[i]</code>	将列表a中索引为i的成员删除
<code>del a[i:j]</code>	将列表a中索引从i到j（不含）的片段删除

面向对象方式的列表操作:

方法	含义
<code><列表>.append(x)</code>	将x添加到<列表>的尾部
<code><列表>.sort()</code>	对<列表>排序（使用缺省比较函数cmp）
<code><列表>.sort(mycmp)</code>	对<列表>排序（使用自定义比较函数mycmp）
<code><列表>.reverse()</code>	将<列表>次序颠倒
<code><列表>.index(x)</code>	返回x在<列表>中第一次出现处的索引
<code><列表>.insert(i,x)</code>	在<列表>中索引i处插入成员x
<code><列表>.count(x)</code>	返回<列表>中x的出现次数
<code><列表>.remove(x)</code>	删除<列表>中x的第一次出现
<code><列表>.pop()</code>	删除<列表>中最后一个成员并返回该成员
<code><列表>.pop(i)</code>	删除<列表>中第i个成员并返回该成员



列表操作

- 类似字符串操作:
 - 合并: `<seq> + <seq>`
 - 重复: `<seq> * <int_expr>`
 - 索引: `<seq>[<index_expr>]`
 - 分段: `<seq>[<start>:<end>]`
 - 长度: `len(<seq>)`
 - 迭代: `for <var> in <seq>: ...`
- 删除列表成员:
 - `del <seq>[<start>:<end>]`



列表操作(续)

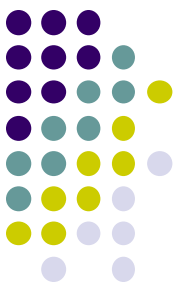
- 专用于列表的方法:
 - 追加: `<list>.append(x)`
 - 排序: `<list>.sort()`
 - 逆转: `<list>.reverse()`
 - 查找`x`的索引: `<list>.index(x)`
 - 在`i`处插入`x`: `<list>.insert(i, x)`
 - 数`x`的个数: `<list>.count(x)`
 - 删除`x`: `<list>.remove(x)`
 - 按索引取出成员: `<list>.pop(i)`
 - 隶属: `x in <list>`



列表操作-索引

- 索引操作和字符串类似
 - 通过在序列中的位置编号来访问成员
<列表>[<位置编号>]
 - 例如

```
>>> x = [1, "two", 3.0, True]
>>> x[0]
1
>>> x[-1]
True
>>> x[1+1]
3.0
```



列表操作-子列表

- 子列表操作和字符串类似
 - 指定序列中的开始和结束位置
<列表>[<开始位置>:<结束位置>]

- 例如

```
>>> x = [1, "two", 3.0, True]
>>> x[0:2]
[1, 'two']
>>> x[1:]
['two', 3.0, True]
>>> x[:-1]
[1, 'two', 3.0]
```

- 列表也有+和*操作,意义和字符串类似

```
>>> [1,3,5]+[2,4]
[1, 3, 5, 2, 4]
>>> 4*[3.0,True]
[3.0, True, 3.0, True, 3.0, True, 3.0, True]
```




与列表有关的几个内建函数

- 求列表长度len()

```
>>> x=4*[3.0,True]
>>> len(x)
8
```

- 删除列表成员del()

```
>>> x=[1,2,3]
>>> del x[1]
>>> x
[1, 3]
```

- 产生整数列表range()

```
>>> range(1,10,2)
[1, 3, 5, 7, 9]
```

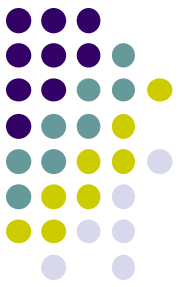
列表的应用



- 以下代码是找到字符串中所有字典顺序的字符串，并存放在字符串列表中，以备排序用

```
# This is demo09 to output the
# longest alphabetical sequences in a string

s=raw_input('Please input the string: ')
start=0
# find all the alphabetical sequences
strList=[]      # for the alphabetical sequences
for i in range(len(s)-1):
    if s[i]>s[i+1]:
        strList=strList+[s[start:i+1]]
        start=i+1
strList=strList+[s[start:]]
```



列表与字符串

- 回顾: 字符串是字符序列,可通过索引引用串的组成部分.
- 列表与字符串的区别:
 - 列表的成员可以是任何数据类型,而字符串中只能是字符;
 - 字符串不能删改,而列表可以

```
>>> x=[1, True, "spring"]
>>> x[0]=6
>>> x
[6, True, 'spring']
>>> del x[1]
>>> x
[6, 'spring']
```



列表与数组

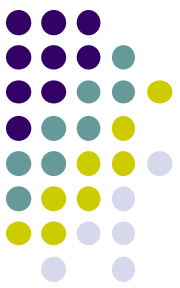
- **list**与其他语言中的数组**array**相似,但不同
 - 列表是动态的,而数组是定长的
 - 列表可以增删成员
 - 不要求各成员都是相同类型的
 - 成员本身也可以是列表
 - 例如

```
[2, "apples"]
```

```
[1, "two", 3.0, True]
```

```
[[1, "apple"], [2, "pears"]]
```

列表例：中位数



```
2 def getInputs():
3     data = []
4     x = raw_input("Enter a number (<Enter> to quit): ")
5     while x != "":
6         data.append(eval(x))
7         x = raw_input("Enter a number (<Enter> to quit): ")
8     return data
9
```

```
10 def sum(aList):
11     s = 0.0
12     for x in aList:
13         s = s + x
14     return s
15
16 def mean(aList):
17     return sum(aList) / len(aList)
18
```

```
19 def median(aList):
20     aList.sort()
21     size = len(aList)
22     mid = size / 2
23     if size % 2 == 1:
24         m = aList[mid]
25     else:
26         m = (aList[mid] + aList[mid-1]) / 2.0
27     return m
28
```

```
29 def main():
30     print "This program computes sum, mean and median."
31     data = getInputs()
32     sigma = sum(data)
33     xbar = mean(data)
34     med = median(data)
35     print "Sum:", sigma
36     print "Average:", xbar
37     print "Median:", med
38
39 main()
```

元组类型



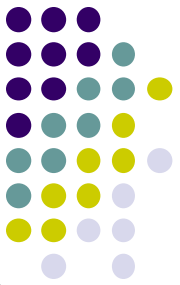
- 元组类型**tuple**

- 用圆括号括起的成员集合体，如(), (8,) (3,"6",True)
- 和列表基本相同,只是不能删改成员
- 如果数据序列不需要改变，则用**tuple**比用**List**效率高
- 元组类型可以用**tuple**作构造器，将一个字符串或列表转换成元组对象。

```
>>> tuple('hello')
('h', 'e', 'l', 'l', 'o')
>>> t=tuple([1,2,3])
>>> t
(1, 2, 3)
>>> t[1]=8
```

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    t[1]=8
TypeError: 'tuple' object does not support item assignment
```

集合：无序集合体



- 集合类型**set**，用于表示大量数据的无序集合体。数据之间没有次序，并且互不相同。
- 集合类型的值有两种创建方式：一种是用一对花括号将多个用逗号分隔的数据括起来；另一种是调用函数**set()**，此函数可以将字符串、列表、元组等类型的数据转换成集合类型的数据。
- Python在创建集合值的时候会自动删除掉重复的数据。

```
>>> s={1,2,2,3,3,3}
>>> s
set([1, 2, 3])
>>> s1={1,2,2,3,3,3}
>>> s1
set([1, 2, 3])
>>> s2=set([1,2,3,4,5,6])
>>> s2
set([1, 2, 3, 4, 5, 6])
>>> s1|s2
set([1, 2, 3, 4, 5, 6])
>>> s1&s2
set([1, 2, 3])
>>> s2-s1
set([4, 5, 6])
```

集合操作



运算	含义
<code>x in <集合></code>	检测 <code>x</code> 是否属于<集合>，返回True或False
<code>s1 s2</code>	并集
<code>s1 & s2</code>	交集
<code>s1 - s2</code>	差集
<code>s1 ^ s2</code>	对称差
<code>s1 <= s2</code>	检测 <code>s1</code> 是否 <code>s2</code> 的子集
<code>s1 < s2</code>	检测 <code>s1</code> 是否 <code>s2</code> 的真子集
<code>s1 >= s2</code>	检测 <code>s1</code> 是否 <code>s2</code> 的超集
<code>s1 > s2</code>	检测 <code>s1</code> 是否 <code>s2</code> 的真超集
<code>s1 = s2</code>	将 <code>s2</code> 的元素并入 <code>s1</code> 中
<code>len(s)</code>	<code>s</code> 中的元素个数

字典:无序集合体



- 列表实现了索引查找:按给定位置检索.
- 很多应用需要“键-值”查找:按给定的键,检索相关联的值.
- 字典类型(dict):存储“键-值对”.
 - 创建: `dict = {k1:v1, k2:v2, ..., kn:vn}`
 - 检索: `dict[<ki>]`返回相关联的<v_i>
 - 值可修改:`dict[<ki>] = <new_value>`
 - 键类型常用字符串,整数;值类型则任意.
 - 存储:按内部最有效的方式,不保持创建顺序.

```
>>> d1=dict(name='George',age=12,hobby=('sport','violin'))
>>> d1
{'hobby': ('sport', 'violin'), 'age': 12, 'name': 'George'}
>>> d2={1:'Mon',2:'Tues',3:'Wedn',4:'Thur',5:'Fri',6:'Sat',7:'Sun'}
>>> d2
{1: 'Mon', 2: 'Tues', 3: 'Wedn', 4: 'Thur', 5: 'Fri', 6: 'Sat', 7: 'Sun'}
```

字典例



- 缩略语字典

`abbr = {'etc': 'cetera', 'cf': 'confer', 'ibid': 'ibidem'}`

```
>>> abbr={'etc': 'cetera', 'cf.': 'confer', 'e.g.': 'for example', 'ibid': 'ibidem'}
>>> abbr['etc']
'cetera'
>>> abbr['etc']='et cetera'
>>> abbr
{'cf.': 'confer', 'etc': 'et cetera', 'ibid': 'ibidem', 'e.g.': 'for example'}
```

- 月份映射表

`month = {1: 'Jan', 2: 'Feb', 3: 'March', 4: 'April'}`

```
>>> month = {1: 'Jan', 2: 'Feb', 3: 'March', 4: 'April'}
>>> month[4]
'April'
>>> month[5]
```

```
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    month[5]
KeyError: 5
>>> month[5]='May'
>>> month
{1: 'Jan', 2: 'Feb', 3: 'March', 4: 'April', 5: 'May'}
```

字典操作



Python将字典实现为对象：

方法	含义
<code><字典>.has_key(<键>)</code>	若<字典>包含<键>，返回 True ； 否则返回 False
<code><字典>.keys()</code>	返回所有键构成的列表
<code><字典>.values()</code>	返回所有值构成的列表
<code><字典>.items()</code>	返回所有(key,value)元组构成的列表
<code><字典>.clear()</code>	删除<字典>的所有条目

```
>>> month.has_key(5)
True
>>> month.keys()
[1, 2, 3, 4, 5]
>>> month.values()
['Jan', 'Feb', 'March', 'April', 'May']
>>> month.items()
[(1, 'Jan'), (2, 'Feb'), (3, 'March'), (4, 'April'), (5, 'May')]
>>> del month[1]
>>> month
{2: 'Feb', 3: 'March', 4: 'April', 5: 'May'}
>>> month.clear()
>>> month
{}
```

编程实例:词频统计 assign3



- 统计文档中单词的出现次数.
- 用字典结构:
 - 用很多累积变量显然不好!
 - `counts:{<单词>:<频度计数>}`
- 分词

```
for ch in ",.;;\n":  
    text.replace(ch, ' ')  
wordlist = string.split(text)
```

- 单词首次出现时字典里查不到会出错:

```
for w in wordlist:  
    try:  
        wordCounts[w]=wordCounts[w]+1  
    except KeyError:  
        wordCounts[w]=1
```

- 如何输出前n个最频繁的单词? 根据频度进行排序生成列表:

```
Countlist=sorted(counts.items(), key=lambda count:count[1],reserse=True)
```



第6章 大量数据的表示和处理

- 简单数据类型（回忆）
 - 数值数据类型: int, long, float
 - 布尔类型: bool
 - 字符串类型: str
- 复杂数据类型
 - 集合体类型
 - 有序集合体: 字符串、列表、元组
 - 无序集合体: 集合、字典
 - 文件
 - 高级数据结构: 链表、堆栈、队列

文件处理



- 文件:对存储在磁盘上的一组数据予以命名.
- 典型的数据组织粒度:
 - 基本数据项
 - 若干数据项构成固定结构的记录
 - 若干记录构成文件
- 例:
 - 基本数据项:学号,姓名,年龄
 - 一个学生的记录:{学号,姓名,年龄}
 - 一个文件:全体学生的记录

文本文件



- 文件中是文本数据
 - 相应地有二进制数据.
- 可视为存储在磁盘上的字符串.
 - 单行字符串
 - 多行字符串
 - 行尾(EOL):用特殊字符,如新行(\n)字符.
 - Python用\n表示新行字符,该字符在显示时被解释成新行字符. 例:

```
print "first line\nsecond line"
```

```
>>> print 'first line\nsecond line'
first line
second line
```



文件处理:打开文件

- 打开文件:将磁盘文件与一个程序变量关联,做好读写准备.
`<filevar> = open(<filename>,<mode>)`
- `<mode>`: 'r','w','a'
- 一个完整的文件标识由磁盘驱动器、目录层次和文件名三部分构成。在Python程序中，路径分隔字符既可以使用“\”，也可以使用“/”
- 例如

```
infile = open("d:\\demo\\demo1.py", "r")  
outfile = open("d:/demo/demo2.py", "w")  
oldfile = open("demo3.py", "a")
```


文件处理:读写文件



- 读文件:读出文件内容

`<filevar>.read()`

`<filevar>.read(n)`

`<filevar>.readline()`

`<filevar>.readlines()`

`<filevar>.seek(n)`

- 点表示法:程序中文件是对象!

- 要有文件当前读写位置的概念!

- 写文件:将新内容写入文件.

`<filevar>.write(<string>)`

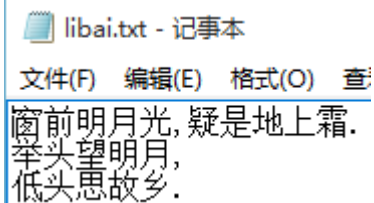
- 若想写多行内容,在string后加\n

- 关闭文件:取消文件变量与磁盘文件的关联.

`<filevar>.close()`

- 系统会将缓存中的文件内容输出到磁盘,并释放资源。

```
>>> f=open('D:/courses/Python/Demo/libai.txt','w')
>>> f.write('窗前明月光,')
>>> f.write('疑是地上霜.\n')
>>> f.write('举头望明月,\n低头思故乡.')
>>> f.close()
```



libai.txt - 记事本

文件(F) 编辑(E) 格式(O) 查

窗前明月光,疑是地上霜.
举头望明月,
低头思故乡.

编程实例:批处理



- 通过文件实现成批数据的输入输出
 - 这种情况不适合用交互方式输入

```
infile = open(infileName, 'r')
outfile = open(outfileName, 'w')
for line in infile.readlines():
    first, last = string.split(line)
    uname = string.lower(first[0]+last[:7])
    outfile.write(uname + '\n')
```

```
infile.close()
outfile.close()
```

```
infile = open("input.dat", "r")↵
for line in infile:↵
    do something with line ...↵
```

这种用法有个好处是无需考虑内存大小，而 **readlines()** 要求内存足够大，以便容纳它返回的列表。

缓冲



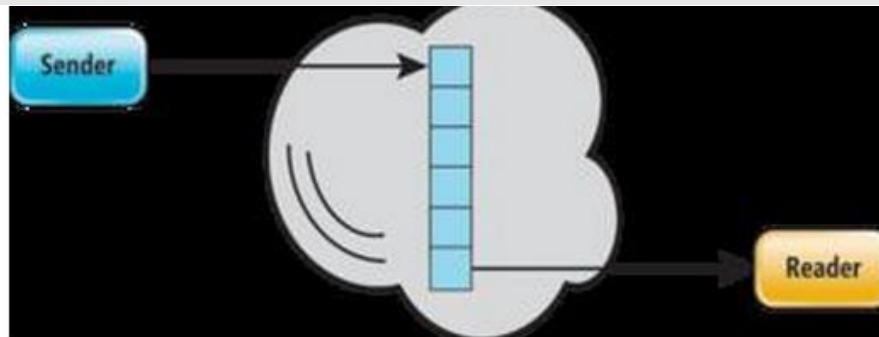
- **缓冲区 (buffer):** 在内存中建立过渡带，解决输入与输出的速率差异。

- 计算机与打印机
- 银行等待去

- **例：文件缓冲**

编写一个文件拷贝程序，功能是将用户指定的文件复制到文件夹d:\backup中。假设内存容量有限或者CPU处理能力有限，导致每次只能处理1024个字符。为此，使用read(n)来读文件，其中参数n表示从文件读取n个字符。

```
def main():  
    fname = raw_input("Enter file name: ")  
    f = open(fname, "r")  
    fcopy = open("d:/backup/"+fname, "w")  
    while True:  
        buffer = f.read(1024)  
        if buffer == "":  
            break  
        fcopy.write(buffer)  
    f.close()  
    fcopy.close()
```



二进制文件与随机存取



- 打开二进制文件时必须指明“以二进制方式打开”，具体就是用"rb"、"wb"和"ab"分别表示读打开、写打开和追加打开。例如：

```
In [81]: bf1 = open("c:/windows/notepad.exe", "rb")
```

```
In [82]: bf1.read(10)
```

```
Out[82]: 'MZ\x90\x00\x03\x00\x00\x00\x04\x00'
```

有时候也需要对文件进行随机读写，即直接定位到文件的特定位置进行读写。Python提供的seek()方法可用于文件的随机存取

<文件对象>.seek(n)

<文件对象>.seek(n,m)



可使用tell()方法确定当前读写位置。

其中，seek(n)的含义是将文件当前位置移到偏移为n的地方，这里的偏移是相对于文件开始位置的，即文件的第1个字节偏移为0，第2个字节偏移为1，依此类推。seek(n,m)的含义是将文件当前位置移到偏移为n的地方，这里的偏移要依m值来定：m为0时相对于文件开始位置，m为1时相对于文件当前位置，m为2时相对于文件末尾。偏移为正数表示朝文件尾方向移动，偏移为负数表示向文件头方向移动。



第6章 大量数据的表示和处理

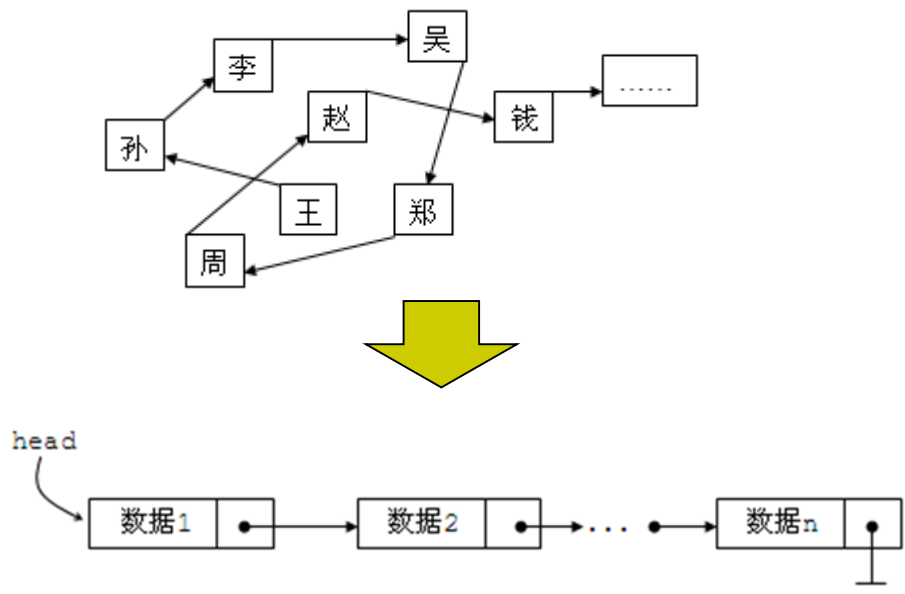
- 简单数据类型（回忆）
 - 数值数据类型: int, long, float
 - 布尔类型: bool
 - 字符串类型: str
- 复杂数据类型
 - 集合体类型
 - 有序集合体: 字符串、列表、元组
 - 无序集合体: 集合、字典
 - 文件
 - 高级数据结构: 链表、堆栈、队列

链表

- 顺序排列
- 优点：仅凭排列次序（或相邻关系）就知道成员数据之间的逻辑关系，而不需要另外存储表示成员间逻辑关系的信息；可以通过位置信息（索引）对任何成员进行随机访问，而不需要从头开始一个一个查看。
- 缺点：如果需要增加新成员，必须移动大量数据以便为新成员腾出空间；如果要删除某个数据，删除后必须移动大量数据以便填补空缺、保持连续性。

王	孙	李	吴	郑	周	赵	钱
---	---	---	---	---	---	---	---	-------

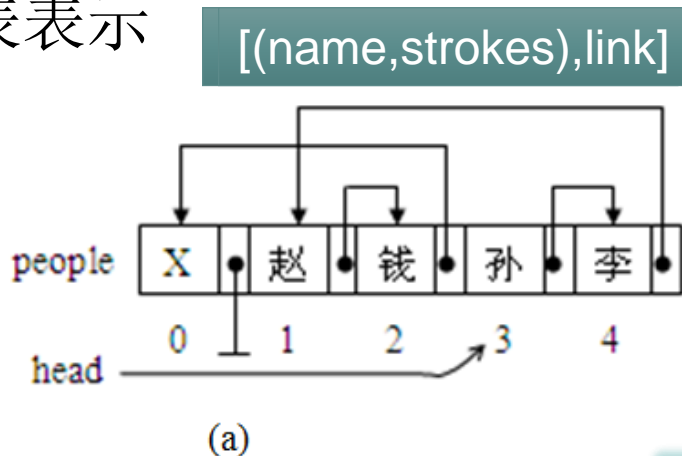
④ 链表 (*linked list*)



链表的表示和处理

- 指针类型
- 列表表示

按姓氏笔划排序



地址	结点
0	[('X', 100), -1]
1	[('赵', 9), 2]
2	[('钱', 10), 0]
3	[('孙', 6), 4]
4	[('李', 7), 1]

(b)

图 6.7 链表的表示

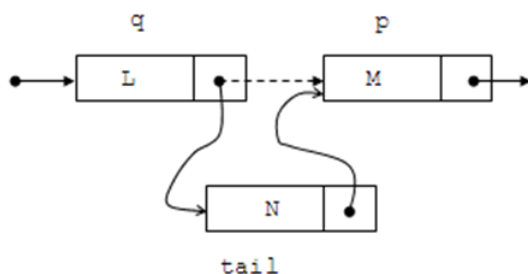


图 6.8 向链表中插入新结点

在链表中插入新结点

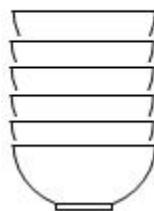
```

p = head
q = -1
while True:
    if people[p][0][1] <= people[tail][0][1]:
        q = p
        p = people[p][1]
    else:
        people[tail][1] = p
        if q >= 0:
            people[q][1] = tail
        else:
            head = tail
        break
    
```

堆栈



- 堆栈是一种数据集合体，具有“后进先出” LIFO 的特点。
- 主要操作：
 - **push(x)**: 在堆栈顶部推入一个新数据x，x即成为新的栈顶元素；
 - **pop()**: 从堆栈中取出栈顶元素，显然被取出的元素只能是最后加入堆栈的元素。
 - **isFull()**: 检查堆栈是否已满。如果堆栈具有固定大小，那么满了之后是无法执行**push()**的；
 - **isEmpty()**: 检查堆栈是否为空。如果堆栈是空的，那么**pop()**操作将出错。



堆栈



- 可以采用多种不同的方式来实现堆栈这个抽象数据类型。
- 用列表来实现:令列表**stack**是存放数据的堆栈，以列表尾为栈顶，那么向堆栈中放入元素就只能在尾部添加，可以用列表**append**方法来实现堆栈**push()**，列表**pop**方法来实现堆栈**pop()**

```
>>> from demo_complexDatatype_listStack import Stack
>>> s=Stack()
>>> s.items
[]
>>> s.push(1)
>>> s.push(2)
>>> s.push(3)
>>> s.items
[1, 2, 3]
>>> s.pop()
3
>>> s.pop()
2
>>> s.items
[1]
>>> s.pop()
1
>>> s.isEmpty()
True
```

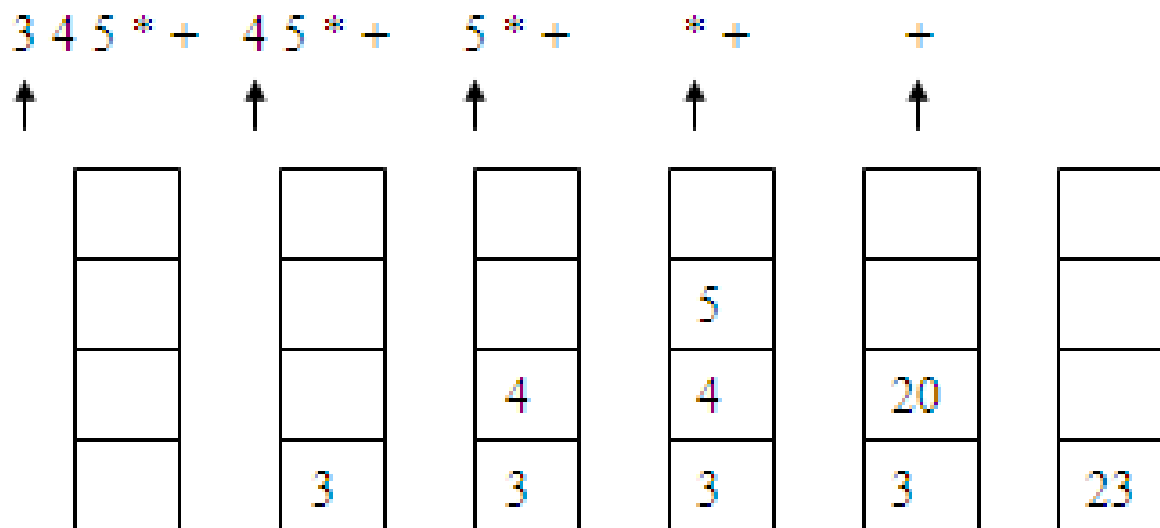
```
# implement stack using list.
```

```
class Stack:
    def __init__(self):
        self.items = []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def getLen(self):
        return len(self.items)
    def isEmpty(self):
        return (self.items == [])
```

堆栈应用



- 算术表达式的中缀形式和后缀形式（无需括号）
- 例如“ $1 + 2$ ”可写成“ $1\ 2\ +$ ”、“ $3 + 4 * 5$ ”可写成“ $3\ 4\ 5\ *\ +$ ”。后缀形式的表达式可以利用堆栈来非常方便地求值。

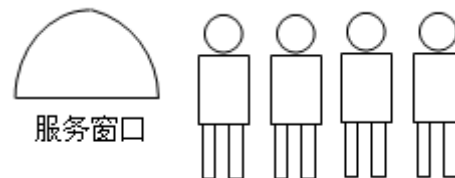


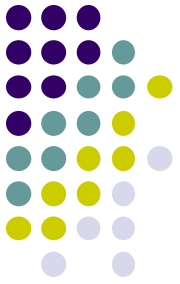
队列



- 队列是一种数据集合体，具有“先进先出” **FIFO** 的特点。
- 主要操作是：
 - **enqueue**: 入队，即在队列尾部添加数据；
 - **dequeue**: 出队，即将队列头部的数据移出队列作为返回值。
- 队列的具体实现有多种方式，例如可以用顺序列表、链表来实现队列。

```
>>> q=Queue()  
>>> q.enqueue(1)  
>>> q.enqueue(2)  
>>> q.enqueue(3)  
>>> q.items  
[1, 2, 3]  
>>> q.dequeue()  
1  
>>> q.items  
[2, 3]
```





End