第二章 流程控制和数据类型

条件判断

循环

列表和元组

字典

集合

文件处理

条件判断

当我们开始编写一个程序时，肯定会遇到要求程序能根据不同的条件做出相应动作的需求，这时候，我们就需要使用计算机语言的最基本的逻辑控制 if..else 语句来实现了，比如我们要写一个验证用户信息的程序，

```
valid_user = 'alex'
user_input = raw_input("Your username:")
if user_input == valid_user:
	print "Welcome %s login to our system!" % user_input
else:
	print "invalid username! Byebye!"
```

根据 if..else 的语法规则，如果 if 语句判断为 True,就会执行它下面的子代码(下层缩进代码)，否则的话，就执行 else 下面的子代码。当然后面的 else 也不是必须要加的，如果不加 else 的话，那 if 条件判断为 False 的话，那它就什么也不做。

所以在上面的代码中，如果你输入的是 alex 的话，那就会打印"Welcome alex login to our system!"，否则就打印"invalid username! Byebye!".

看上面的程序，我们只验证了用户名，正常情况下得用户名和密码一起验证才能确定这个用户是否合法对不对？没错，那如何同时验证 2 个条件呢？即用户名和密码都匹配上才能算验证成功。有的同学灵机一动说，再加一层判断不就得了，如下：

```
valid_user = 'alex'
valid_passwd = 'alex123!'
user_input = raw_input("Your username:")
passwd = raw_input("Your password:")
if user_input == valid_user:
	if passwd == valid_passwd:
		print "Welcome %s login to our system!" % user_input
else:
	print "invalid username! Byebye!"
```

我们在原有的代码基础上新加了3行(已用背景色标注)，不错，这样确实实现了用户和密码同时验证，当用户名正确后，就再继续判断密码，如果密码也正确，就打印Welcome信息，唉呀，但如果用户名正确但密码不正确呢？`if passwd == valid_passwd:`后面没有`else`语句了，那程序就直接退出了对不对？但是我们想要的是只要用户名和密码都不匹配就打印"`invalid username!Byebye!`" 那句呀！怎么办怎么办？刚才那个同学又说了，再在`if passwd == valid_passwd:`后面加个`else`判断不就得了，如下：

```
valid_user = 'alex'
valid_passwd = 'alex123!'
user_input = raw_input("Your username:")
```

```
passwd = raw_input("Your password:")
if user_input == valid_user:
        if passwd == valid_passwd:
                print "Welcome %s login to our system!" % user_input
        else:
                print "invalid username! Byebye!"
else:
        print "invalid username! Byebye!"
```

哈，好聪明，果真实现了！试了一下，这样确实就没问题了！但别高兴太早，这样虽然实现了需求,但却是一种非常 low 的实现方式,这样写程序会被人鄙视的，what? 实现了功能不就得了，再说，这哪 low 了呀？告诉你吧，这个聪明同学犯了写代码的一大忌，重复代码，程序中出现的两个 else 所做的事情是一模一样的，属于重复代码，这无疑使整个程序笨重了很多，记住，你写程序时一定要尽最大的可能避免出现重复代码，这是专业程序员需具备的基本道德之一，为什么呢？重复代码有以下害处：

1. 使程序变的冗长、笨重
2. 使程序变的不易维护，当你的程序出现多处重复代码时，如果突然你需要对一个功能做更改，但这个功能又在程序中重复了多次，那你就必须在各个重复的地方把修改这个功能，太恶心了。计算机语言支持函数、类功能的原因之一就是要减少代码的冗余。

好吧，那既然这样写很 low，快来告诉我们什么是合格的写法！没问题，来看代码：

```
valid_user = 'alex'
valid_passwd = 'alex123!'
user_input = raw_input("Your username:")
passwd = raw_input("Your password:")
if user_input == valid_user and passwd == valid_passwd:
        print "Welcome %s login to our system!" % user_input
else:
        print "invalid username! Byebye!"
```

what the fuck？只加了半句就实现了？yes！只加半句就够了，看我这里写的 `and passwd == valid_passwd` 的意思就是让 `if` 条件必须同时满足 2 个条件了，当然你还可以再后面再写 1 个 `and` ,2 个 `and……` n 个 `and`,那 `if` 条件就必须要满足 n 个条件才会执行它下面的语法了，但是正常人都不写那么多 `and` ,为啥？因为这降低了你程序的可读性，所以一般一个 `if` 判断同时满足 1 至 3 个条件就可以了，条件少则怡情，多则伤身，很多则灰飞烟灭。。。噢，跟 `and` 对应的语法是 `or`，“或”的意思，`and` 是“与”的意思，就是满足 1 个或另 1 个或另 1 个条件或。。。另 n(灰飞烟灭)个条件就会执行它后面的代码。

elif...else

如果的程序想对外开放访客功能，就是没有用户名密码也可以登录我的程序，但

所具有的权限可能就只能是只读了，那这个如何实现呢？假如我们设定所有的访客都用一个统一的 guest 用户，我们只要一遇到用户输入 guest 就跳入另一个判断是不是就可以了呢？来看代码：

```
valid_user = 'alex'
valid_passwd = 'alex123!'
user_input = raw_input("Your username:")
passwd = raw_input("Your password:")
if user_input == valid_user and passwd == valid_passwd:
       print "Welcome %s login to our system!" % user_input
elif user_input == 'guest':
       print "Welcome %s login our system,but you only have read-only access,enjoy!"
else:
       print "invalid username! Byebye!"
```

这里我们用到了 `elif`,意思就是，如果不满足第一个 `if` 条件，那程序就会继续往下走，再判断是否满足 `elif` 条件，如果不满足，就再继续走（这里你可以加多个 `elif` 判断），只要遇到有满足的 `elif` 就停下来执行它后面的代码，然后结束，如果最终没有碰到满足的条件，就最终执行 `else` 语法。另外真的可以写多行噢，如下：

```
if <条件判断 1>:

     <执行 1>

elif <条件判断 2>:

    <执行 2>

elif <条件判断 3>:

    <执行 3>

else:

    <执行 4>
```

好了，if...elif...else 就是这些东西。

循环

刚才我们的验证用户名密码的程序只对用户进行了一次验证就退出了，如果我们想让程序在用户输入验证信息错误后允许用户再进行几次尝试应该如何做呢？这个时候我们就可以用到 for 循环了，首先 for 循环的基本语法如下：

```
for i in range(5):  #循环5次
       print 'Loop', i

执行输出$:python 05for_loop.py
Loop 0
```

```
Loop 1
Loop 2
Loop 3
Loop 4
```

把它套到我们之前用户验证的程序上是什么样子的呢？看代码：

```
valid_user = 'alex'
valid_passwd = 'alex123!'

for i in range(3):
	user_input = raw_input("Your username:")
	passwd = raw_input("Your password:")
	if user_input == valid_user and passwd == valid_passwd:
		print "Welcome %s login to our system!" % user_input
		break
	elif user_input == 'guest':
		print "Welcome %s login our system,but you only have
read-only access,enjoy!"
		break
	else:
		print "invalid username!"
```

这个代码实现了，如果用户名密码错误后，最多让用户尝试 3 次，注意这里用到了 break,它是干什么用的呢？听好了，它是用来帮助跳出整个循环的，就是现在你的程序要循环 3 次，但是用户再尝试了第 2 次时，就验证成功了，这时候你还需要让他再进行一次验证吗？当然不需要，这时候需要让程序在验证成功后直接跳出整个循环，不再需要进行下一次循环操作，Ok, break 就是干这个用的，这个 for 循环中我用了 2 个 break,一个是在验证成功时让它直接跳出，一个是在用户是 guest 访客时。纪住，循环中允许你使用多个 break,但是一旦跳出循环，程序就继续往下走了，你可就不能再返回来了。还有就是 break 只能在循环中使用噢,没有循环的时候用 break 会出错，因为没啥可跳的，哈哈。

好，我想让程序再智能些，就是在用户尝试 3 次失败后，就打印你的 ip 地址已经被锁定，以防止恶意攻击。该咋办呢？其实很简单，直接在 for 循环结尾处加个 else 就搞定：

```
for i in range(3):
	user_input = raw_input("Your username:")
	passwd = raw_input("Your password:")
	if user_input == valid_user and passwd == valid_passwd:
		print "Welcome %s login to our system!" % user_input
		break
	elif user_input == 'guest':
		print "Welcome %s login our system,but you only have
read-only access,enjoy!"
		break
	else:
```

```
            print "invalid username!"
else:
      print "You've retried 3 times,to avoid attack,I will block your
IP address.."
```

哈哈，是不是很简单？这个 else 在什么时候会执行呢？在 for 循环正常循环完所设定的次数，中间没有被 break 中断的话，就会执行 else 啦，但如果中间被 break 了，那 else 语句就不会执行了。

While 循环

刚才的 for 循环我们是不是预先指定了要循环多少次了？那如果我有种需求，要求我的程序每 10s 中循环一次，天长地久、直到永远怎么办？这里就需要用到 while 啦，while 就可以是那种你不 break 它就傻傻的一直死循环的语句，来看下：

```
import time #导入time模块
count = 0    #设置一个计数器，每循环一次加一次，这样就知道循环多少次了。
while True:  #只要为真，就执行下面代码，每循环一次就判断一次
       count += 1 #每循环一次就自加1
       print "Loop ", count
       time.sleep(10) #每循环一次就 sleep 10s 再继续运行
```

输出：

```
python 07while_endless_loop.py
Loop  1
Loop  2
Loop  3
```

上面的程序就是死循环了，只要进程没被杀死，就会一直运行下去，当然你也可设置成循环到一定次数就自动停止，如下：

```
import time
count = 0
run_forever = True  #设置run_forever变量默认为True,这样就会一直运行
while run_forever:
       count += 1
       print "Loop ", count
       if count == 9:
              run_forever = False #当count==9时，就把run_forever改成False,
这样下一次判断时就不成立了，就不再运行了。
              #break
       time.sleep(10)
```

1. 注意上面的代码在 if count==9 后面我现在写的是，在循环到第 9 次后，通过把

run_forever 变量由 True 改成 False 来让下一次循环的执行条件不成立的方式实现终止循环。其实我们在这里也可用 break 对不对？没错，break 也可以让这个循环终止，但是跟改 run_forever=False 的方式有什么不同呢？不同就是，如果你现在用 run_forever=False 的方式的话，它后面的 time.sleep(10)还会继续执行，也就是程序会在 10s 后才会退出，但是如果你直接 break 的话，那后面 time.sleep(10)就不会执行了，因为你已经跳出整个循环了，所以程序就直接立刻退出了。 明白了么？不懂的话自己写一遍。

continue VS break

跟 break 相对应的还有一个负责循环跳出的语法就是 continue，它跟 break 有什么区别呢？我们都知道了 break 是负责跳出整个循环，但 continue 是跳出本次循环，继续下一次循环。就是说，循环过程中，如果遇到 continue,那这一次循环本应该执行的后面的代码就不执行了，直接跳过了，直接进行下一次循环了。还是不明白？呵呵，那来个例子吧：
需求：要求循环 10 次，但每次循环只有在遇到第奇数次循环时才打印循环的次数。

```
#_*_ coding:utf-8 _*_
for i in range(10):
        if i%2 ==0 : #取模运算，只要i除以2没有余数，那这个i肯定是偶数，不能被整除就是奇数
            continue #如果if条件成立，就跳出此次循环，下面的print就不执行了
        print u"这是奇数",i #只要上面 if 条件不成产，那就肯定是奇数喽，就执行这句
```

注意：break 只能跳出一层循环，如里你现在有 2 个循环，第二个循环嵌套在第一个循环里面，如果第 2 个循环被 break 了，那第 1 个循环会继续执行噢，Python 不支持一次性跳出多层循环噢，像 c 语言中的 goto，shell 中的 break 后面跟跳出层次的在方法在 python 中都没有，这可不是因为 python 设计的 low，而是允许一次性跳出多个循环很容易造成程序流程的混乱，使理解和调试程序都产生困难，这也是为什么 goto 在 c 语言中不建议使用的原因。但是确实在有的情况下我们需要一次性跳出多个循环，这怎么办呢？其实很简单，哈，来看：

```
#_*_ coding:utf-8 _*_
loop1 = 0 #设定loop1 and loop2这两个计数器
loop2 = 0
while True:
        loop1 +=1
        print "Loop1:", loop1
        break_flag = False #在父循环中设定一个跳出标志，子循环只要想连父亲一块跳出时，就把这个标志改成True
        while True:
                loop2 +=1
                if loop2 ==5:
```

```
            break_flag = True #让我爹一块往外跳
            break #我先跳出第一层
    print 'Loop2:',loop2
    if break_flag: #儿子跳了没有？
            print u"接到子循环跳出通知，我也得跳了！"  #我擦，儿子真跳了。
            break
```

上面代码的基本逻辑就是，在第一层循环中设置一个是否跳出的标志变量默认为 False,如果子循环在 break 时想连它的上一层一起 break,就可以把这个跳出的标志变量改成 True，当子循环跳出后，父循环会继续往下走，但下在的语句是判断这个跳出变量是否已经被子循环改掉了，如果已经改掉了，那就直接跳出就好了，因此你就实现了同时跳出 2 层循环的功能啦。好了，上面的代码自己练习吧，流程控制就这些的东西了。

列表和元组

列表(List)

列表是在编程中经常用到的一种数据类型，它跟其它语言中所指的数组基本是一样的，列表是指一组有序的数据集合，可以将各种各样的数据有序的存放在列表中，并且可以对其进行增删改查，以及遍历。列表的存在是为了通过一个变量存储更多的信息，比如我想在一个变量里存储一张购物清单，然后程序只需要通过我定义的这个变量就可以找到购物清单中的任意一个或多个商品。如下：

```
>>> shopping_list = ['Iphone', 'Mac', 'Bike','Coffee','Car','Clothes','Food','Gift']
>>> shopping_list
['Iphone', 'Mac', 'Bike', 'Coffee', 'Car', 'Clothes', 'Food', 'Gift']
```

通过 len()内置函数可查看列表中元素的个数

```
>>> len(shopping_list)
8
```

你可以通过索引来找到列表中每个元素的位置，记住索引是从 0 开始的

```
>>> shopping_list[2]  #找Bike
'Bike'
>>> shopping_list[0] #第一个元素取出来
'Iphone'
>>> shopping_list[-1] #-1代表取列表中最后一个元素
'Gift'
>>> shopping_list[-3] #取倒数第3位元素
'Clothes'
>>> shopping_list[-4] #取倒数第4个元素
'Car'
>>> shopping_list[8] #取索引为8的元素
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

#最后一个shopping_list[8]报错了，原因是引用超出了索引范围，不是有8个元素吗？取第8个怎么会出错呢？这里要注意了，列表的索引是从0开始的，你这里写的8其实是取了列表中的第9个位置，但你列表中一共有8个元素，所以肯定取不到喽。

切片(Slice)

你还可以从列表中取出指定多个元素，这种操作叫做切片

```
>>> shopping_list
['Iphone', 'Mac', 'Bike', 'Coffee', 'Car', 'Clothes', 'Food', 'Gift']
>>>
>>> shopping_list[0:3] #取0到第3个元素，不包括第4个
['Iphone', 'Mac', 'Bike']
>>> shopping_list[:3]  #同上，取0到第3个元素，不包括第4个，0可以不写
['Iphone', 'Mac', 'Bike']
>>> shopping_list[2:5]  #取第3至第5个元素
['Bike', 'Coffee', 'Car']
>>> shopping_list[:-3] #取从0至倒数第3个元素
['Iphone', 'Mac', 'Bike', 'Coffee', 'Car']
>>> shopping_list[-3:] #取最后3个元素
['Clothes', 'Food', 'Gift']
>>> shopping_list[1:8:2]  #从1至8隔一个取一个，后面的2是步长，即每隔几个元素取一个
['Mac', 'Coffee', 'Clothes', 'Gift']
>>> shopping_list[::2] #从头到位每隔一个取一个
['Iphone', 'Bike', 'Car', 'Food']
```

增删改查

```
>>> shopping_list.append('MovieTicket')  #向列表后面追加一个元素
>>> shopping_list
['Iphone', 'Mac', 'Bike', 'Coffee', 'Car', 'Clothes', 'Food', 'Gift', 'MovieTicket']
>>> shopping_list.pop() #删除最后一个元素
'MovieTicket'
>>> shopping_list.remove('Mac') #删除叫'Mac'的元素，如果有多个'Mac',那会删除从左边数找到的第一个
>>> shopping_list[2]
'Coffee'
>>> shopping_list[2] = 'COFFEE' #将索引为2的元素改为"COFFEE"，原来是小写
>>> shopping_list.insert(3,"Toy") #插入一个新元素，索引为3
>>> shopping_list
```

```
['Iphone', 'Bike', 'COFFEE', 'Toy', 'Car', 'Clothes', 'Food', 'Gift']
>>> shopping_list.index('Toy') #返回'Toy'元素的索引值，如果有多个相同元素，则返回匹配的第一个
3
>>> shopping_list.append('Food')
>>> shopping_list.count('Food') #统计'Food'的元素的个数，刚添加了一个，所以现在是2个
2
>>> shopping_list
['Iphone', 'Bike', 'COFFEE', 'Toy', 'Car', 'Clothes', 'Food', 'Gift', 'Food']
>>> list2= ['Banana','Apple'] #创建一个新列表
>>> shopping_list.extend(list2) #把上面的新列表合并到shopping_list中
>>> shopping_list
['Iphone', 'Bike', 'COFFEE', 'Toy', 'Car', 'Clothes', 'Food', 'Gift', 'Food', 'Banana',
'Apple']
>>> shopping_list.sort() #将列表排序
>>> shopping_list
['Apple', 'Banana', 'Bike', 'COFFEE', 'Car', 'Clothes', 'Food', 'Food', 'Gift', 'Iphone',
'Toy']
>>> shopping_list.reverse() #将列表反转
>>> shopping_list
['Toy', 'Iphone', 'Gift', 'Food', 'Food', 'Clothes', 'Car', 'COFFEE', 'Bike', 'Banana',
'Apple']
>>> del shopping_list[3:8] #删除索引3至8的元素，不包括8
>>> shopping_list
['Toy', 'Iphone', 'Gift', 'Bike', 'Banana', 'Apple']
>>> for i in shopping_list: #遍历列表
...   print i
```

元组（Tuple）

另一种有序列表叫元组：tuple。tuple 和 list 非常类似，但是 tuple 一旦初始化就不能修改，比如同样是列出同学的名字：

```
>>> classmates = ('Michael', 'Bob', 'Tracy')
```

现在，classmates 这个 tuple 不能变了，它也没有 append()，insert()这样的方法。其他获取元素的方法和 list 是一样的，你可以正常地使用 `classmates[0]`，`classmates[-1]`，但不能赋值成另外的元素。

不可变的 tuple 有什么意义？因为 tuple 不可变，所以代码更安全。如果可能，能用 tuple 代替 list 就尽量用 tuple。

tuple 的陷阱：当你定义一个 tuple 时，在定义的时候，tuple 的元素就必须被确定下来，比如：

```
>>> t = (1, 2) >>> t (1, 2)
```

如果要定义一个空的 tuple，可以写成 `()`：

```
>>> t = () >>> t ()
```

但是，要定义一个只有 1 个元素的 tuple，如果你这么定义：

```
>>> t = (1) >>> t 1
```

定义的不是 tuple，是 `1` 这个数！这是因为括号 `()` 既可以表示 tuple，又可以表示数学公式中的小括号，这就产生了歧义，因此，Python 规定，这种情况下，按小括号进行计算，计算结果自然是 `1`。

所以，只有 1 个元素的 tuple 定义时必须加一个逗号 `,`，来消除歧义：

```
>>> t = (1,) >>> t (1,)
```

Try

Python 在显示只有 1 个元素的 tuple 时，也会加一个逗号 `,`，以免你误解成数学计算意义上的括号。

程序练习

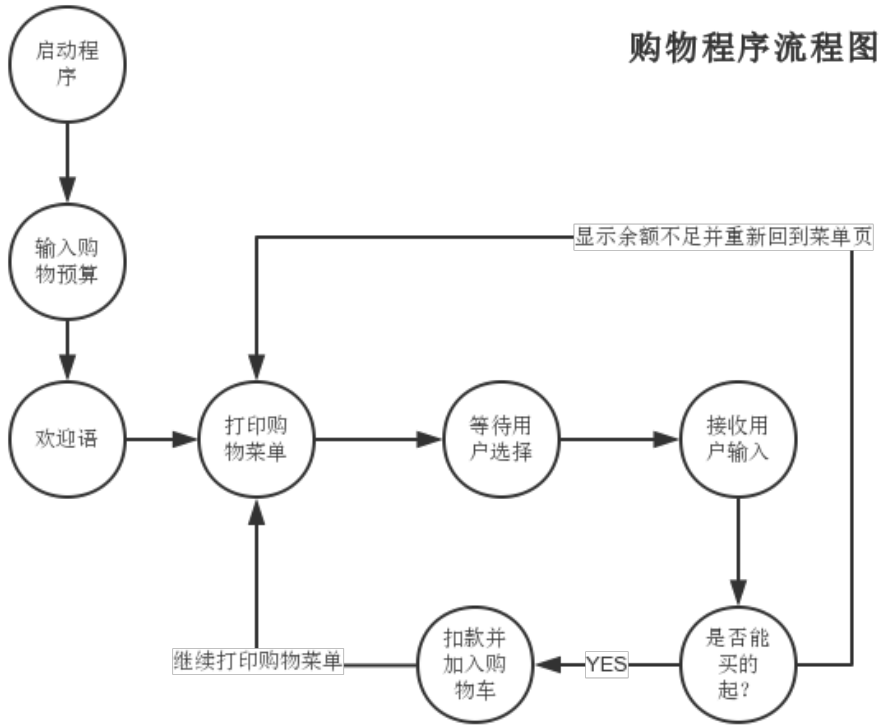
既然列表和元组的用法你都学会了，那我们接下来做个程序练习下吧，看看你是真会了，还是只是看上去会了。。。
题目：购物小程序
需求：程序启动后，要求用户输入购物预算，然后打印购物菜单，菜单格式如下：
Welcome to Alex's shopping mall, below are the things we are selling:

1. MacBook Air 7999
2. Starbucks Coffee 33
3. Iphone 6Plus 6188
4. ...

用户可以不断的购买商品，程序要实时的把购买的商品添加到购物车，并且从预算金额中扣掉相应商品的价格，可购买的商品总值不能超过预算总值，用户选择退出后，打印他已购商品及所剩金额。

首先这个程序很简单，但对于之前完全没有写过程序的同学来讲，就可能不知如何下手了，觉得没思路，很苦恼。呵呵，这个不用担心，很多初学者都要经历过这个阶段，就是语法都会，看别人写的代码也基本能看懂，但一到自己写就不知怎么办了，这种场景有没有在曾经上学时学英语的过程中遇到过？肯定有，好多同学现在依然是英语文档能看懂，也能写一些，但就是说不出来，为什么，因为练的少呀！！！你写程序跟学英语是一样的，你不会写就是因为你练的少，怎么办呢？无它法，只能硬憋！憋不出来就不睡觉！你不这么搞，你永远也学不会编程！

当然了，憋也不是瞎憋，要有方法，我给大家的建议就是在开始写程序前先画流程图，画流程图的过程中能帮你理清思路，流程图画出来了，再写就会容易的多，因为只需要照着流程图的步骤去做就可以了。针对这个购物程序，我先给大家画个图做为示例(是用在线画图软件 processOn 画的)：



请你以后在写程序前务必要画一下流程图，不要上来就写代码，流程图除了帮你理清思路外，还能帮你避免错误，因为仅靠脑子变想变写，很容易就把程序写走了样。

此程序的代码我就不再这展示了，请各位在自己写完后，可以拿我们的示例答案来对比，看自己是否哪里写的不合理、不专业，看哪里是否可以改进，但一定不要自己还没写出来呢，就看我写的答案，那你很难培养好你的学习能力。

字典（Dict）

列表允许你通过一个变量存储大量的信息，但试想以下场景，用列表实现就可能效率较低了：

1. 存储的信息量越来越多，有的时候找一个数据可能要循环整个列表，耗时较长。
2. 单个元素包含的信息量变多时，比如，之前只是存储姓名列表，现在是要存储姓名、年龄、身份证号、地址、工作等这个人的很多信息，用列表去存储很费劲
3. 要求存储的数据是不重复，我们知道列表是允许的重复值的，当然想存储时就让我的数据默认就是唯一的话，用列表就不可以了

以上这些是列表不擅长的地方，却恰恰是我们接下来要讲的 dict 所擅长的方面，dict 使用 key-value 的形式存储数据，dict 的 key 是唯一的，所以你可以通过 key 来唯一的定位到你的数据。之所以叫字典(在其它语言中称为 map)，是因为 dict 的数据结构跟我们生活中用的字典是一样的，查英文字典时，输入单词，就可以定位到这个单词意思的详细解释，其中这个单词就是 key,对应的词义解释就是 value.字典有如下特点：

1. key-value 格式，key 是唯一的
2. 无序，与列表有序的特点不同，字典是无序的，列表只所以有序是因为你需要通过索引来定位相应元素，而字典已经可以通过 key 来定位相应 value,因此为了避免浪费存储空间，字典不会对数据的位置进行纪录，当然如果你想让其变成有序的，也是有方法的，这个我们以后再讲。
3. 查询速度很快，dict 是基于 hash 表的原理实现的，是根据关键字（Key value）而直接访问在内存存储位置的数据结构。也就是说，它通过把键值通过一个函数的计算，映射到表中一个位置来访问记录，这加快了查找速度。这个映射函数称做散列函数，存放记录的数组称做散列表。由于通过一个 key 的索引表就直接定位到了内存地址，所以查询一个只有 100 条数据的字典和一个 100 万条数据的字典的速度是查不多的。

好了，来看看 dict 的语法：

```
>>> info = {'name':'alex',
             'job':'engineer',
             'age': 29,
             'company':'AUTOHOME'
            }
>>> info
{'age': 29, 'job': 'engineer', 'company': 'AUTOHOME', 'name': 'alex'}
```

增删改查

```
>>> info['name'] #查看key为’name’的value
'alex'
>>> info['job'] = 'Boss' #将key 的value 改为’Boss’
>>> info
{'age': 29, 'job': 'Boss', 'company': 'AUTOHOME', 'name': 'alex'}
>>> info['city'] = 'BJ' #如果dict中有key为’city’,就将其值改为’BJ’,如果没有这个key,就创建一条新纪录
>>> info
{'age': 29, 'job': 'Boss', 'company': 'AUTOHOME', 'name': 'alex', 'city': 'BJ'}
>>> info.pop('age') #删除key为’age’的数据，跟del info[‘age’] 一样
29
>>> info
{'job': 'Boss', 'company': 'AUTOHOME', 'name': 'alex', 'city': 'BJ'}
>>> info.popitem() #随机删除一条数据，dict为空时用此语法会报错
```

```
('job', 'Boss')
>>> info.items() #将dict的key,value转换成列表的形式显示
[('company', 'AUTOHOME'), ('name', 'alex'), ('city', 'BJ')]
>>> info.has_key('name') #判断字典中是否有个叫'name'的key
True
>>> info['age'] #查找一个不存在的key报错，因为'age'刚才已经删除了，所以报错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'age'
>>> info.get('age') #查找key,如果存在则返回其value,否则则返回None
>>> info.get('name')
'alex'
>>> info.clear() #清空dict
>>> info
{}
>>> info.fromkeys(['a','b','c'],'Test')  #根据列表['a','b','c']来创建dict里的key,后面的'Test'是默认value,如果不指定的话则为None
{'a': 'Test', 'c': 'Test', 'b': 'Test'}
>>> info
{}
>>> info =info.fromkeys(['a','b','c'],'Test')
>>> info
{'a': 'Test', 'c': 'Test', 'b': 'Test'}
>>> info.setdefault('d','Alex') #找一个key为'd'的纪录,如果这个key不存在,那就创建一个叫'd'的key,并且将其value设置为'Alex'， 如果这个key存在，就直接返回这个key的value，见下一条
'Alex'
>>> info.setdefault('c','Alex')
'Test'
>>> info
{'a': 'Test', 'c': 'Test', 'b': 'Test', 'd': 'Alex'}
>>> dict2 = {'e':'fromDict2','a':'fromDict2'} #创建一个新字典
>>> info.update(dict2) #拿这个新字典去更新info,注意dict2中有一个key值'a'与dict info相冲突，这时dict2的值会覆盖info中的a,如果dict2的key在info中不存在，则创建相应的纪录
>>> info
{'a': 'fromDict2', 'c': 'Test', 'b': 'Test', 'e': 'fromDict2', 'd': 'Alex'}
```

遍历 dict

（于 2015/2/24 早 5 点，倒时差睡不着，起来写会。。。）

遍历 dict 与遍历列表差不多，只不过要记得 dict 是 key-value 的结构，要想在遍历时同时打印这 key 和 value，需要这样写：

```
info = {
    'name': 'Alex Li',
```

```
    'age': 29,
    'job': 'Engineer',
    'phone': 1493335345
}


for item in info:
    print item, info[item] #print item 只会打印 key,如果想同时打印 value,需要再通过 item 去取
```

还有一种遍历的方式：

```
for key,val in info.items():
print key,val
```

以上是默认把字典转换成了一个大列表，并且把每对 key-value 值转换成了元组，所以你可以在每次循环时赋 2 个变量进去，因为循环的数据格式如下：
[('job', 'Engineer'), ('phone', 1493335345), ('age', 29), ('name', 'Alex Li')]
因此每循环一次，其实就是把相应元组中的 2 个值赋值 key,val 这两个变量并打印。

深潜 copy

我们之前写的 dict 的例子中，在给 value 赋值是都是简单的字符串或数字，那能否是其它的数据类型呢？value 可不可以是一个列表呢？可不可以又是一个字典呢？当然可以，这个 value 你可以存任何数据类型噢， 但 key 不是噢，因为要保证 key 是唯一的(可被 hash 的),所以 key 可允许的数据类型一般只有 string 和 int，好了，下面我们往 value 中多存点东西看看：

```
staff_contacts = {
    0023 : {'name':'Alex Li', 'age':29, 'department':'IT','phone':3423},
    3951 : {'name':'Jack Ma', 'age':51, 'department':'Management Team','phone':5563},
    5342 : {'name':'Rain Wang', 'age':24, 'department':'HR','phone':2942},
}
```

good,你看 dict 里面还可以再嵌套 dict，子 dict 还可以继续往下嵌套，无穷尽也，当然也包括 list 啥的，那好，前面都是铺垫，接下来要讲重要的了，注意看，我现在准备把这个 staff_contacts 拷贝一份并在拷贝后将原来的 dict 中 key 0023 里的 age 改掉，如下；

```
contacts2 = staff_contacts
staff_contacts[0023]['age'] = 28
print 'staff_contacts:', staff_contacts[0023]
print 'contacts2:' , contacts2[0023]
```

这时我再打印这 2 个 dict，还记得我们第一章讲过的，当你将一个变量 a 直接赋值给另一个变量 b 时，其实只是把变量 a 所指向的内存地址 copy 给了变量 b 对吧，所以你再更改变量 a 的值时，变量 b 是不会受影响的对吧！那根据这个理论，我们上面把 staff_contacts 赋值给了 contacts2,并且把 staff_contacts 里的 0023 中的'age'的 value 从 29 改成了 28，那 contacts2 肯定不会受影响对吧？看下面执行结果：

```
staff_contacts: {'department': 'IT', 'phone': 3423, 'age': 28, 'name': 'Alex Li'}
contacts2: {'department': 'IT', 'phone': 3423, 'age': 28, 'name': 'Alex Li'}
```

唉呀我去，结果是这 2 个 dict 都把 key 0023 的 age 值改掉了，这是为什么？我们来看下这两个 dict 中 0023 对应的内存地址是否变了：

```
print 'id of staff_contacts: ', id(staff_contacts[0023])
print 'id of contacts2: ', id(contacts2[0023])
```

输出

```
id of staff_contacts:    140543549137920
id of contacts2:    140543549137920
```

结果竟然一样唉！不是说好的嘛，如果 copy 后，更改其中一个变量的值，会在内存里创建一份新的数据，怎么又不对了？呵呵，没错，对于字符串、数字等一些简单的数据类型，python 确实会像我们第一章讲的那样处理。但对于 dict、list、tuple 以及我们稍后讲的 set 集合。Python 默认不会再这样做了，而只是简单的做了一个别名，就像 Linux 系统中对文件做软连接一样，这样你无论是更改原来的 dict，还是 copy 出来的新 dict，其实最终改的都是同一份数据，所以才会出现上面 2 个 dict 的数据都改了的情况。为啥要这么干呢？所谓事出必有因，这么干是为了避免不必要的内存浪费，因为一般来讲，一个变量存字符串、数字等简单的数据类型时，存的信息不会太多，但如果你用来存一个 list 或 dict,这个变量最终会变成多大可就说不准了，你一个 dict 可以存上百万条甚至上亿条数据呀，想象下上亿条的数据存在一个 dict 中，怎么也得吃掉上 GB 的内存吧，那这时你突然将这个 dict 拷贝了给了另一个变量，并且对旧的 dict 中的其中一条数据做了更改，如果 python 还按原来默认的方式 copy 一份独立数据出来的话，那岂不是要立刻再吃掉数 GB 的内存？只因为更改了其中一个 dict 中的一条数据？这。。。这是不是不太合理？没错，这样做的话，那效率太低了，并且极度浪费资源，所以为解决这样的问题，python 直接给你做了一个软连接，对于复杂的数据类型像 dict or list 等，无论你怎么复制，最终改的其实只是一份数据。

这时有同学要问了，那如果我真的想 copy 一份独立的新数据出来该如何办呢 good,想一下，这样的需要也是合理的，如果你真要这么干的话，只需用这个 copy 语句:

```
contacts2 = staff_contacts.copy()
```

用帮助来看下这个语法的解释：

```
help(staff_contacts.copy)
copy(...)
    D.copy() -> a shallow copy of D
```

A shallow copy 就是浅 copy 的意思，啥叫浅 copy 呀？呵呵，就是浅浅的 copy

一层，为了看出效果，我们在 copy 之前先在原来 staff_contacts 中再加一条 3333, 最后 staff_contacts 的数据如下：

```
staff_contacts = {
    0023 : {'name':'Alex Li', 'age':29, 'department':'IT','phone':3423},
     #3951 : {'name':'Jack Ma', 'age':51, 'department':'Management Team','phone':5563},
    #5342 : {'name':'Rain Wang', 'age':24, 'department':'HR','phone':2942},
    3333   :' nothing'
}
```

此时我们再将 0023 的 age 改掉，并且把 3333 的值也改掉，注意 3333 对应的 value 是字符串。

```
staff_contacts[0023]['age'] = 28
staff_contacts[3333] = 'change this by original dict'
```

直接打印 2 个 dict 看结果：

```
staff_contacts: {19: {'department': 'IT', 'phone': 3423, 'age': 28, 'name': 'Alex Li'}, 3333: 'change this by original dict'}
contacts2: {19: {'department': 'IT', 'phone': 3423, 'age': 28, 'name': 'Alex Li'}, 3333: ' nothing'}
```

可以看到，3333 经过浅 copy 后，两份数据已经独立了，但 0023 age 的值依然是在 2 个 dict 中都改了，这说明它俩肯定还是共享的一个内存地址。但为啥 3333 改了但 0023 的 age 值没改呢？都说过了嘛，人家这是浅 copy,只是肤浅的 copy 一下 dict 下的第一层数据，再深入的就不管了，还是用原来的。 这么搞还是为了节省内存空间和提高效率，因为很多时间你并不需要真的 copy 一份完全独立的 dict 数据，可能只是想让它们第一层的值保持独立而已。 但刚才提问的那个同学很固执的就想要一份完完全全独立的数据，我内存多，我任性，我不在乎！alright 好吧。如果真想这么干，也不是没办法，但你就得导入一个模块来干了。

```
import copy
contacts2 = copy.deepcopy(staff_contacts)
```

这就是深 copy 喽，无论你的 dict or list 下面有多少层，它都会给你 copy 了，这就是真正的完全 copy 一份独立数据了。不过建议刚才那位同学，还是慎用这个方法。有钱任性也不禁作呀，哈哈！好了，就是这些，自己试下吧。

集合（Set）

python 的 set 和其他语言类似, 是一个无序不重复元素集, 基本功能包括关系测试和消除重复元素. 集合对象还支持 union(联合), intersection(交), difference(差)和 sysmmetric difference(对称差集)等数学运算.

sets 支持 x in set, len(set),和 for x in set。作为一个无序的集合，sets 不记录元素位置或者插入点。因此，sets 不支持 indexing, slicing, 或其它类序列（sequence-like）的操作。

基本语法

```
s = set([3,5,9,10])        #创建一个数值集合

t = set("Hello")           #创建一个唯一字符的集合
```

与列表和元组不同，集合是无序的，也无法通过数字进行索引。此外，集合中的元素不能重复。例如，如果检查前面代码中 t 集合的值，结果会是：

```
>>> t

set(['H', 'e', 'l', 'o'])
```

注意只出现了一个'l'。

集合支持一系列标准操作，包括并集、交集、差集和对称差集，例如：

```
a = t | s          # t 和 s 的并集
b = t & s          # t 和 s 的交集
c = t - s          # 求差集（项在 t 中，但不在 s 中）
d = t ^ s          # 对称差集（项在 t 或 s 中，但不会同时出现在二者中）
```

基本操作：

```
t.add('x')            # 添加一项
s.update([10,37,42])  # 在 s 中添加多项

t.remove('H')     #从集合中删除'H'，如果不存在则引发 KeyError
len(s)
set 的长度
x in s    #测试 x 是否是 s 的成员
x not in s    #测试 x 是否不是 s 的成员

s.discard(x)    #如果在 set “s”中存在元素 x，则删除
s.pop()    #删除并且返回 set “s”中的一个不确定的元素，如果为空则引发 KeyError

s.clear()    #删除 set “s”中的所有元素

s.issubset(t)
s <= t
#测试是否 s 中的每一个元素都在 t 中

s.issuperset(t)
s >= t
```

```
#测试是否 t 中的每一个元素都在 s 中

s.union(t)
s | t
#返回一个新的 set 包含 s 和 t 中的每一个元素

s.intersection(t)
s & t
#返回一个新的 set 包含 s 和 t 中的公共元素

s.difference(t)
s - t
#返回一个新的 set 包含 s 中有但是 t 中没有的元素

s.symmetric_difference(t)
s ^ t
#返回一个新的 set 包含 s 和 t 中不重复的元素

s.copy()
#返回 set “s”的一个浅复制
```

set 的去重和关系测试是 2 个非常有用的功能，以后你会经常用到的，先把基本语法练熟吧。

文件操作

很多时候我们写的程序都需要对文件进行读、写操作，比如你要分析日志、存储数据等，和其它语言一样，Python 也内置了对文件进行操作的函数，下面一起来学习一下。

读写文件前，我们先必须了解一下，在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）

读文件

要以读文件的模式打开一个文件对象，使用 Python 内置的 `open()` 函数，传入文件名和标示符：

```
>>> f = open('/Users/michael/test.txt', 'r')
```

标示符'r'表示读，这样，我们就成功地打开了一个文件。

如果文件不存在，`open()` 函数就会抛出一个 `IOError` 的错误，并且给出错误码和详细的信息告诉你文件不存在：

```
>>> f=open('/Users/michael/notfound.txt', 'r')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: '/Users/michael/notfound.txt'
```

如果文件打开成功，接下来，调用 `read()` 方法可以一次读取文件的全部内容，Python 把内容读到内存，用一个 `str` 对象表示：

```
>>> f.read() 'Hello, world!'
```

最后一步是调用 `close()` 方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

```
>>> f.close()
```

由于每次处理完文件后都需要调用 close()方法关闭文件，但是很多时候你可能会忘记写这句话，所以，Python 引入了 `with` 语句来自动帮我们调用 `close()` 方法：

```
with open('/path/to/file', 'r') as f:
        print f.read()
```

通过这种方法可使代码更佳简洁，并且不必调用 `f.close()` 方法，这里的 open()和 file()是一样的，属于别名关系，用哪个都行。

调用 `read()` 会一次性读取文件的全部内容，如果文件有 10G，内存就爆了，所以，要保险起见，可以反复调用 `read(size)` 方法，每次最多读取 size 个字节的内容。另外，调用

`readline()`可以每次读取一行内容，调用`readlines()`一次读取所有内容并按行返回`list`。因此，要根据需要决定怎么调用。

如果文件很小，`read()`一次性读取最方便；如果不能确定文件大小，反复调用`read(size)`比较保险；如果是配置文件，调用`readlines()`最方便：

```
for line in f.readlines():
        print(line.strip()) # 把末尾的'\n'删掉
```

写文件

对文件进行写操作的语法

```
#_*_coding:utf-8_*_

f = file('MyNewfile.txt','w')     # ‘w’ 代表以写模式创建一个新文件
f.write('This is frist line\n')   #  往文件里写入内容
f.write('second line\n')
f.close()
```

上面要注意的是，w 模式是以写模式创建一个新文件，是创建噢，不是打开，什么区别？就是如果你要创建的这个文件所在路径中正好有一个文件也叫 MyNewFile.txt 的话，这个文件会被覆盖，里面的内容会被冲掉噢，所以要创建文件里要谨慎，确保不要不小心覆盖掉同名文件中的内容。那如果你只对现有文件做更改，而不是创建新文件的话，应该如何做呢？我们来想一下，要对现有文件做修改，一般有 2 种情况，一个是更改原文件内容，一个是在原文件内容的基础上追加新内容。我们接下来分别来看下：

追加

```
f = file('MyNewfile.txt','a')   #追加模式打开文件
f.write('This is third line, added in append mode\n')   #往文件中追加一行
f.close()

$ more MyNewfile.txt
This is frist line
second line
This is third line, added in append mode
```

最下面那行就是我们新追加的内容喽。这里有一点要注意的是，当用 a 模式打开文件时，但如果这个文件本身不存在，那 a 模式就会创建这个新文件，并且把内容写进去。

修改内容

当你想通过 PYTHON 对文件进行修改时，你需要先打开这个文件，并且把里面的内容读到内存，修改完后，再把内存中的数据写回到文件里，所以你需要打开 2 个文件，一是原文件，一个是用于把改完的内容保存起来的新文件。看下面代码：

```
原文件：$ more MyNewfile.txt
This is frist line
second line

This is third line, added in append mode
fouth line
fifth line ..
haha

修改文件的代码：$ more file_update.py

f = file('MyNewfile.txt','r')   #先打开原文件
new_f = file('MyNewfile_updated.txt','w')   #创建一个新文件，用于存储改过的内容
for line in f:          #循环原文件
    if 'fifth' in line:     #判断每一行，如果行里包含'fifth'这个词，就把这行换掉
        line = line.replace(line,'----haha replace test----')     #通过 relpace 方法把这行换掉，并把换掉的值再赋值给 line,这样下面只需要一个 write 方法，就可以把改过的和没改过的内容都写到新文件里了
    new_f.write(line)       #把内容写到 new_f 里去
f.close()
new_f.close()
```

此时，原文件是没有更改的，改过的内容写到 MyNewfile_updated.txt 里去了，我们来查看下：

```
$ more MyNewfile_updated.txt
This is frist line
second line

This is third line, added in append mode
fouth line
----haha replace test----
haha
```

可以看到内容已经被更换了，这时有人说了，我是想改原文件的内容呀，你现在等于又给我弄出了一个新文件来呀，呵呵，别急，你再把新文件保存后，再把原文件覆盖掉不就得了。
再上面的代码最后加入下面 2 行，就可以把新文件重命名为旧文件，并把旧的文件内容覆盖。

```
import os
os.rename('MyNewfile_updated.txt','MyNewfile.txt')
```

1. 除了 r\w\a 三种最常用处理文件的模式之外，还有以下几种：
2. r+ 以读写模式打开，其实跟追加的效果是一样的，即能读，又能写，但写其实是追加内容，但是如果文件不存在的话，不会像 a 一样创建文件，而是报错
3. w+ 以写读模式打开，
4. a+ 以追加和读的模式打开
5. rb,wb,ab 是指以二进制的模式打开并处理文件，当处理的文件是非文本文件时，就应该以二进制的格式打开文件，但有的同学说了，我不加 b 也不出错呀，没错，即使不加 b，file 也能正常处理二进制文件，但是当你的程序涉及到跨平台时，就可能有问题了，因为在 Linux 和 Windows 的换行标志位是不一样的，Linux 是”\n”，Windows 是”\r\n”，所以 Windows 上的文件在 Linux 下一般会在换行处显示^M，需要特别转换一下才能正常处理。 当你把 Windows 的文件 copy 到 Linux 并用 Python file 方法处理时，你在打开模式上加上 b,Python 就会帮你把^M 转成\n，否则你的程序就不知道该在哪换行了喽！所以，我们建议，如果你预计可能会处理 Windows 上产生的文件，那还是默认就加上 b 吧。

文件其它方法

1. f.mode 显示文件打开格式
2. f.flush() 把缓冲区中的数据刷到硬盘，当你往文件里写数据时，python 会先把你写的内容写到内存中的缓冲区，等缓冲区满了再统一自动写入硬盘，因此减少了对硬盘的操作次数，毕竟硬盘的速度比内存慢多了。
3. f.read() 把文件一次性读入内存
4. f.readline() 读一行内容
5. f.readlines() 把文件都读入内存，并且每行转成列表中的一个元素
6. f.tell() 显示程序光标所在该文件中的当前的位置
7. f.seek() 跳到指定位置，f.seek(0) 是返回文件开始
8. f.truncate() f.truncate(10)从文件开头截取 10 个字符，超出的都删除。
9. f.writelines() 参数需为一个列表，将一个列表中的每个元素都写入文件
10. f.xreadlines() 以迭代的形式循环文件，在处理大文件时效率极高，只记录文件开头和结尾，每循环一次，只读一行，因此不需要将整个文件都一次性加载到内存，而如果用 readlines(),则需要把整个文件都加载到内存，处理

大文件不适合，容易把内存撑爆。

好了，第二章就是这些，为了把这些知识都充分掌握，下面给大家布置一个作业，作业一定要做，否则过不了两天，你学的就全忘了，跟没学一样，实践出真知呀，通过做作业，不但能复习刚学的，说不定还能学到很多我们没有涉及到的细节。

作业：员工信息查询表

要求：

1. 开发员工信息查询程序，可以支持模糊查询员工信息
2. 匹配到的员工信息，需高亮显示，并且显示匹配条数
3. 支持员工信息的增删改查
4. 用户在启动程序后，可以不断的循环查询

信息表素材：staff.txt

Alex Li, Engineer, IT, 13651054608, alex@126.com

Jack Zhang, Salesman, Sales Dep., 18546304334, jack@sina.com

Rain Wang, HR, HR Dep., 13345634253, rain@autohome.com.cn

注：每个字段用逗号区分

好了，想想怎么做吧，做之前一定要先画一下流程图，因为各位现在还没什么写程序的经验，光靠想估计会把脑子想炸了，画图可以帮你理清思路，加油！写不出来不准继续看下一章！