python3 指南

用 *python3* 玩转电脑

万泽[①] | 德山书生[②]

版本: 0.1

---

① 作者:

② 编者：编者邮箱：a358003542@gmail.com。

前言

本书讨论了 python3 语言的基础和进阶, 并对 python3 中的一些常用模块进行了介绍。其中有关 python3 语言本身的基础和进阶知识为本书的着重点, 将求全面和详细的讨论。

目 录

前言 i

目录 ii

I python3 基础 1

1 beginning 2

1.1 python 简介 . . . 2
1.2 进入 python 的 REPL 环境 . . . 2
1.3 python3 命令行用法 . . . 3
1.3.1 python 执行脚本参数的传递 . . . 3
1.4 geany 的相关配置 . . . 3
1.5 代码注释 . . . 4
1.6 Unicode 码支持 . . . 4
1.7 代码多行表示一行 . . . 5
1.7.1 一行表示多行 . . . 5
1.8 输入和输出 . . . 5
1.8.1 最基本的 input 和 print 命令 . . . 5
1.9 __main__ 和 __name__ . . . 6

2 程序中的操作对象 7

2.1 赋值 . . . 7
2.1.1 序列赋值 . . . 8
2.1.2 同时赋相同的值 . . . 9
2.1.3 增强赋值语句 . . . 11

2.1.4 序列解包赋值 11
2.1.5 可迭代对象的迭代赋值 11
2.2 数值 12
2.2.1 二进制八进制十六进制 12
2.2.2 数学幂方运算 14
2.2.3 数值比较 14
2.2.4 相除取商或余 15
2.2.5 复数 15
2.2.6 abs 函数 15
2.2.7 round 函数 15
2.2.8 min，max 和 sum 函数 16
2.2.9 位操作 17
2.2.10 math 模块 17
2.2.11 random 模块 18
2.2.12 statistics 模块 20
2.3 序列 21
2.3.1 len 函数 21
2.3.2 调出某个值 22
2.3.3 调出多个值 23
2.3.4 序列反转 24
2.3.5 序列的可更改性 25
2.3.6 序列的加法和减法 26
2.4 字符串 26
2.4.1 三单引号和三双引号 26
2.4.2 startswith 方法 27
2.4.3 find 方法 27
2.4.4 replace 方法 27
2.4.5 upper 方法 28
2.4.6 isdigit 方法 28
2.4.7 split 方法 29

2.4.8 join 方法 29
2.4.9 strip 方法 30
2.4.10 format 方法 30
2.4.11 转义和不转义 31
2.4.12 count 方法 31
2.5 列表 31
2.5.1 列表的插入操作 31
2.5.2 append 方法 32
2.5.3 reverse 方法 33
2.5.4 copy 方法 33
2.5.5 sort 方法 33
2.5.6 删除某个元素 36
2.5.7 count 方法 37
2.5.8 index 方法 37
2.5.9 列表解析 37
2.5.10 for 语句中列表可变的影响 40
2.5.11 列表元素替换 40
2.5.12 列表元素去重 41
2.6 字典 41
2.6.1 创建字典 42
2.6.2 字典里面有字典 43
2.6.3 字典遍历操作 43
2.6.4 字典的 in 语句 45
2.6.5 字典对象的 get 方法 45
2.6.6 update 方法 46
2.6.7 pop 方法 46
2.6.8 字典解析 46
2.7 集合 47
2.7.1 集合添加元素 48
2.7.2 集合去掉某个元素 49

2.7.3 两个集合之间的关系 49
2.7.4 两个集合之间的操作 50
2.7.5 clear 方法 50
2.7.6 copy 方法 51
2.7.7 pop 方法 51
2.8 元组 51
2.8.1 生成器表达式 51
2.9 bytes 类型 52
2.9.1 基本编码知识 52
2.9.2 使用方法 53
2.10 bytearray 类型 55
2.11 文件 55
2.11.1 写文件 55
2.11.2 读文件 56
2.11.3 open 函数的处理模式 56
2.11.4 用 with 语句打开文件 57
2.11.5 除字符串外其他类型的读取 57

3 程序中的逻辑 59

3.1 布尔值 59
3.1.1 其他逻辑小知识 59
3.1.2 None 59
3.2 if 条件判断 60
3.2.1 逻辑与或否 61
3.2.2 稍复杂的条件判断 61
3.2.3 try 语句捕捉错误 64
3.2.4 in 语句 67
3.3 for 迭代语句 68
3.3.1 else 分句 68
3.3.2 range 函数 69
3.3.3 迭代加上操作 70

3.3.4 enumerate 函数 70
3.4 while 循环 70
3.4.1 break 命令 71
3.4.2 continue 命令 71
3.4.3 pass 命令 71

4 操作或者函数 **73**
4.1 自定义函数 74
4.2 参数传递问题 74
4.3 变量作用域问题 76
4.3.1 内置作用域 77
4.3.2 global 命令 78
4.3.3 nonlocal 命令 78
4.4 参数和默认参数 82
4.5 不定参量函数 82
4.5.1 序列解包赋值 83
4.5.2 函数中的通配符 84
4.5.3 mysum 函数 84
4.5.4 任意数目的可选参数 84
4.5.5 解包可迭代对象传递参数 85
4.5.6 解包字典成为关键字参数 86
4.6 参数的顺序 86
4.7 递归函式 87
4.7.1 什么时候用递归? 88
4.7.2 lisp 的 car-cdr 递归技术 89
4.8 lambda 函式 92
4.9 print 函数 92

5 类 **93**
5.1 python 中类的结构 93
5.2 类的最基础知识 94

5.2.1 类的创建 94
5.2.2 根据类创建实例 95
5.2.3 类的属性 95
5.2.4 类的方法 96
5.3 类的继承 97
5.4 类的内置方法 98
5.4.1 __init__ 方法 99
5.4.2 self 意味着什么 100
5.4.3 类的操作第二版 100
5.5 类的操作第三版 102
5.5.1 构造函数的继承和重载 104
5.5.2 __str__ 函数的继承和重载 104
5.5.3 类的高级知识 104

6 模块 **105**
6.1 找到模块文件 105
6.2 编写模块 106
6.3 import 语句 107
6.4 from 语句 107
6.5 reload 函数 108

II python3 进阶 **109**

7 类的高级知识 **110**
7.1 类内部的字典 110
7.1.1 __dict__ 值 110
7.1.2 类按键取值 111
7.1.3 类按键赋值 111
7.1.4 字典的索引删除 112
7.2 属性管理 113
7.2.1 属性赋值 113

7.2.2 属性删除 113
7.2.3 属性获取 113
7.3 数学运算 113
7.3.1 一般加法 113
7.3.2 右侧加法 114
7.3.3 增强加法 114
7.3.4 一般减法 114
7.3.5 其他数学运算符一览 114
7.4 逻辑运算 115
7.4.1 bool 函数支持 115
7.4.2 类之间的相等判断 115
7.4.3 比较判断操作 119
7.4.4 in 语句 119
7.5 强制类型变换 120
7.6 len 函数 120
7.7 copy 方法和 deepcopy 方法 120
7.8 with 语句支持 120
7.9 函数调用 121
7.10 和迭代操作有关 122
7.10.1 __next__ 方法 122
7.10.2 next 函数 122
7.10.3 iter 函数 122
7.10.4 重构字典的 iter 函数 124
7.11 当对象内存存储回收时的操作 125
7.12 静态方法 125
7.13 装饰器 126
7.13.1 自定义装饰器 127
7.13.2 多个装饰器 127
7.13.3 装饰器带上参数 128
7.14 类方法 129

7.15 多重继承的顺序问题 130
7.15.1 菱形难题 132
7.15.2 super() 134
7.16 给某个对象动态加载一个方法 135

8 深入理解 python3 的迭代 139
8.1 生成器函数 141
8.2 map 和 filter 函数 143
8.2.1 map 函数 144
8.2.2 filter 函数 144
8.2.3 zip 函数 145

9 模块包 148
9.1 __init__ 文件 148
9.1.1 如果里面有 import 语句 149
9.1.2 如果里面有 from 语句 150
9.1.3 __all__ 变量 150
9.2 模块中的帮助信息 152

10 文件处理高级知识 154
10.1 一行行的操作 154
10.2 整个文件的列表解析 155
10.2.1 readlines 方法 155
10.2.2 文本所有某个单词的替换 156

11 与 c 语言或 c++ 语言编写的模块集成 157
11.1 安装和配置 157
11.1.1 通过 apt 安装 157
11.1.2 从 github 下载最新版安装 157
11.1.3 安装后的配置 158
11.2 beginning 158
11.2.1 手工编译 158

11.2.2 通过 setuptools 安装 159

III 常用的模块 161

12 pickle 模块 163

12.1 将对象存入文件 163

12.2 从文件中取出对象 164

13 shelve 模块 166

13.1 存入多个对象 166

13.2 读取这些对象 167

14 time 模块 168

14.1 time 函数 168

14.2 gmtime 函数 168

14.3 localtime 函数 169

14.4 ctime 函数 169

14.5 strftime 函数 170

14.6 sleep 函数 170

15 sys 模块 171

15.1 sys.argv 171

15.2 exit 函数 172

15.3 sys.platform 172

15.4 sys.path 172

15.5 标准输入输出错误输出文件 172

15.6 sys.version 173

15.7 sys.maxsize 173

15.8 sys.stdin.isatty() 173

16 fileinput 模块 174

16.1 input 函数 174

17 os.path 模块 176

17.1 abspath 函数 177
17.2 dirname 函数 177
17.3 basename 函数 178
17.4 split 函数 178
17.5 splitext 函数 178
17.6 join 函数 179
17.7 expanduser 函数 179
17.8 exists 函数 180
17.9 isfile 和 isdir 还有 islink 180
17.10 samefile 函数 180
17.11 getmtime 函数 180
17.12 getctime 函数 181

18 glob 模块 **182**

19 subprocess 模块 **183**
19.1 call 函数 183
19.2 getoutput 函数 184
19.3 getstatusoutput 函数 184
19.4 Popen 类 185

20 os 模块 **186**
20.1 getcwd 函数 186
20.2 mkdir 函数 187
20.3 chdir 函数 187
20.4 删除文件 187
20.5 os.rename 187
20.6 os.repalce 188
20.7 删除空目录 188
20.8 listdir 命令 188
20.9 遍历目录树 189
20.10 environ 函数 191

20.11 getpid 函数 191
20.12 stat 函数 191
20.12.1 st_size 属性 191
20.12.2 st_mtime 属性 192
20.12.3 st_ctime 属性 193
20.13 给进程发送信号 193

21 shutil 模块 **194**
21.1 复制文件 194
21.2 复制文件夹 194
21.3 删除整个目录 195
21.4 移动文件夹 195
21.5 chown 函数 195
21.6 which 函数 195

22 tarfile 和 zipfile 模块 **197**
22.1 制作 gz 压缩文件 197
22.1.1 TarFile 的 add 方法 198
22.2 解压缩 gz 压缩文件 198
22.2.1 TarFile 的 extractall 方法 198
22.3 提取 egg 文件中的内容 198
22.4 制作 zip 压缩文件 199

23 collections 模块 **200**
23.1 namedtuple 函数 200
23.2 Counter 计数类 200

24 re 模块 **201**
24.1 re 模块中的元字符集 201
24.2 re 模块中的特殊字符类 202
24.3 转义问题 203
24.4 re 模块的使用 203

24.4.1 匹配和查找 203
24.4.2 分割操作 205
24.4.3 替换操作 206

25 itertools 模块 **207**
25.1 repeat 函数 207
25.2 starmap 函数 207

26 multiprocessing 模块 **209**
26.1 Pool 类 209
26.1.1 starmap 方法 210
26.2 ThreadPool 类 211

IV python3 高级篇 **212**

27 python 的多任务处理 **213**
27.1 进程 fork 213
27.2 子进程和父进程分开 214
27.3 线程入门 215
27.4 多线程: 一个定时器 218
27.5 多线程下载大文件 220

28 python 的元类编程 **225**

V 附录 **226**

A 更多的 python 信息 **227**

B linux 及时了解 **228**

C 古怪的 python **229**
C.1 字符串比较大小 229
C.1.1 中文比较大小? 230
C.1.2 ord 和 chr 函数 230

D 其他 **232**

D.1 exec 和 eval 232

D.1.1 如果执行 import 语句 234

D.2 assert 语句 235

D.3 属性管理的函数 235

参考资料 **236**

python3 基础

beginning

python 简介

Python 是个成功的脚本语言。它最初由 Guido van Rossum 开发，在 1991 年第一次发布。Python 由 ABC 和 Haskell 语言所启发。Python 是一个高级的、通用的、跨平台、解释型的语言。一些人更倾向于称之为动态语言。它很易学，Python 是一种简约的语言。它的最明显的一个特征是，不使用分号或括号，Python 使用缩进。现在，Python 由来自世界各地的庞大的志愿者维护。

python 现在主要有两个版本区别，python2 和 python3。作为新学者推荐完全使用 python3 编程，本文档完全基于 python3。

完全没有编程经验的人推荐简单学一下 c 语言和 scheme 语言（就简单学习一下这个语言的基本概念即可）。相信我学习这两门语言不会浪费你任何时间，其中 scheme 语言如果你学得深入的话甚至编译器的基本原理你都能够学到。了解了这两门语言的核心理念，基本上任何语言在你看来都大同小异了。

进入 python 的 REPL 环境

在 ubuntu13.10 下终端中输入 python 即进入 python 语言的 REPL 环境，目前默认的是 python2。你可以运行：

```
python  --version
```

来查看。要进入 python3 在终端中输入 python3 即可。

python3 命令行用法

命令行的一般格式就是：

```
python3  [可选项]  test.py  [可选参数 1 可选参数 2]
```

同样类似的运行`python3  --help` 即可以查看 python3 命令的一些可选项。比如加入 **-i** 选项之后，python 执行完脚本之后会进入 REPL 环境继续等待下一个命令，这个在最后结果一闪而过的时候有用。后面的 -c，-m 选项还看不明白。

python 执行脚本参数的传递

上面的命令行接受多个参数都没有问题的，不会报错，哪怕你在 py 文件并没有用到他们。在 py 文件中要使用他们，首先导入 sys 模块，然后 sys.argv[0] 是现在这个 py 文件在系统中的文件名，接下来的 sys.argv[1] 就是之前命令行接受的第一个参数，后面的就依次类推了。

geany 的相关配置

geany 的其他配置这里不做过多说明，就自动执行命令默认的应该是 python2，修改成为：

```
python3  -i  %f
```

即可。

代码注释

python 语言的注释符号和 bash 语言（linux 终端的编程语言）一样用的是 # 符号来注释代码。然后 py 文件开头一般如下面代码所示：

```
#!/usr/bin/env python3
#-*-coding:utf-8-*-
```

其中代码第一行表示等下如果 py 文件可执行模式执行那么将用 python3 来编译[①]，第二行的意思是 py 文件编码是 utf-8 编码的，python3 直接支持 utf-8 各个符号，这是很强大的一个更新。

多行注释可以利用编辑器快速每行前面加上 # 符号。

Unicode 码支持

前面谈及 python3 是可以直接支持 Unicode 码的，如果以可执行模式加载，那么第二行需要写上：

```
#-*-coding:utf-8-*-
```

这么一句。

读者请实验下面这个小例子，这将打印一个笑脸符号：

① 也就是用 chmod 加上可执行权限那么可以直接执行了。第一行完整的解释是什么通过 *env* 程序来搜索 python 的路径，这样代码更具可移植性。

```
#!/usr/bin/env python3
#-*-coding:utf-8-*-
print('\u263a')
```

☺

>>>

上面的数字就是笑脸符号具体的 Unicode 码（十六进制）。

代码多行表示一行

这个技巧防止代码越界所以经常会用到。用反斜线 \ 即可。不过更常用的是将表达式用圆括号 ( ) 括起来，这样内部可以直接换行并继续。在 python 中任何表达式都可以包围在圆括号中。

一行表示多行

python 中一般不用分号，但是分号的意义大致和 bash 或者 c 语言中的意义类似，表示一行结束的意思。其中 c 语言我们知道是必须使用分号的。

输入和输出

最基本的 input 和 print 命令

input 函数请求用户输入，并将这个值赋值给某个变量。注意赋值之后类型是字符串，但后面你可以用强制类型转换——int 函数（变成整数），float 函数（变成实数），str 函数（变成字符串）——将其转变过来。print 函数就是一般的输出函数。

读者请运行下面的例子：

```
x=input(' 请输入一个实数：')
string=' 你输入的这个实数乘以 2 等于：'+ str(float(x)*2)
print(string)
```

__main__ 和 __name__

按照这个网站的讲解，如果当前这个 py 文件是被执行的，那么__name__ 在本 py 文件中的值是__main__，如果这个 py 文件是被作为模块引入的，那么__name__ 在那个 py 文件中的值是本 py 文件作为模块的模块名。比如说你随便新建一个 test.py 文件，这个 py 文件里面就简单打印__name__ 的值，这个时候你会发现__name__ 的值是字符”test”，如果是 mymodule 模块里的 mymod.py 文件，那么在这个 py 文件里面其__name__ 的值是”mymodule.mymod”。

程序中的操作对象

python 和 c 语言不同，c 是什么`int x = 3`，也就是这个变量是整数啊，字符啊什么的都要明确指定，python 不需要这样做，只需要声明`x = 3` 即可。但是我们知道任何程序语言它到最后必然要明确某一个变量（这里也包括后面的更加复杂的各个结构对象）的内存分配，只是 python 语言帮我们将这些工作做了，所以就让我们省下这份心吧。

```
''' 这是一个多行注释
    你可以在这里写上很多废话
    '''
x = 10
print(x,type(x))
```

python 程序由各个模块（modules）组成，模块就是各个文件。模块由声明（statements）组成，声明由表达式（expressions）组成，表达式负责创造和操作对象（objects）。在 python 中一切皆对象。python 语言内置对象（数值、字符串、列表、数组、字典、文件、集合、其他内置对象。）后面会详细说明之。

赋值

python 中的赋值语法非常的简单，x=1，就是一个赋值语句了。和 c 语言不同，c 是必须先声明 int x 之类，开辟一个内存空间，然后才能给这个 x 赋

值。而 python 的 x=1 语句实际上至少完成了三个工作：一，判断 1 的类型（动态类型语言必须要这步）；二，把这个类型的对象存储在内存里面；三，创建 x 这个名字和这个名字指向这个内存，x 似乎可以称之为对应 c 语言的指针对象。

序列赋值

```
x,y=1,'a'
[z,w]=['b',10]
print(x,y,z,w)
```

```
1 a b 10
>>>
```

我们记得 python 中表达式可以加上圆括号，所以这里x,y 产生的是一个数组(x,y)，然后是对应的数组平行赋值，第二行是列表的平行赋值。这是一个很有用的技巧。

在其他语言里面常常会介绍 swap 函数，就是接受两个参数然后将这两个参数的值交换一下，交换过程通常要用到临时变量。而在 python 中不需要再创建一个临时变量了，因为序列赋值会自动生成一个临时的右边的序列（其中的变量都对应原来的原始值），然后再一一对应赋值（这里强调一一对应是指两边的序列长度要一致。）

交换两个元素

在 python 中交换两个元素用序列赋值形式是很便捷的：

```
>>> x = 1
>>> y = 2
>>> x,y = y,x
>>> print(x,y)
2 1
```

这个过程显然不是先执行 x=y 然后执行 y=x，如上所述的，程序首先右边创建一个临时的序列，其中的变量都对应原来的值，即x,y=(2,1)，然后再进行序列赋值。

同时赋相同的值

```
x=y='a'
z=w=2
print(x,y,z,w)
```

```
a a 2 2
>>>
```

这种语句形式 c 语言里面也有，不过内部实现机制就非常的不一样了。python 当声明 x=y 的时候，x 和 y 是相同的指针值，然后相同的指针值都指向了'a' 这个字符串对象，也可以说 x 和 y 就是一个东西，只是取的名字不同罢了。

我们用 is 语句[①]来测试，显示 x 和 y 就是一个东西。

① is 语句用来测试对象的同一性，就是真正是内存里的同一个东西，而不仅仅是值相同而已。== 只是确保值相同。

```
>>> x=y='a'
>>> x is y
True
>>> x == y
True
```

但如果写成这种形式：

```
>>> x = 'a'
>>> y = 'a'
>>> x is y
True
```

x 和 y 还是指向的同一个对象，关于这点 python 内部是如何实现的我还不太清楚（似乎有点神奇）。为了说明 is 语句功能正常这里再举个例子吧：

```
>>> x = [1,2,3]
>>> y = [1,2,3]
>>> x == y
True
>>> x is y
False
```

我们看到这里就有了两个列表对象，我的一个推测是可变的对象会多次生成，而不可变的对象多个变量是共用的。那么我们看一下元组的情况：

```
>>> x = (1,2,3)
>>> y = (1,2,3)
>>> x is y
False
```

元组不可变，不过他们也不是两个共用的，打住了，这个问题到此吧，有点偏题了。

增强赋值语句

x=x+y 可以写作 x += y。类似的还有：

+=	&=	»=
-=	|=	«=
*=	^=	**=
/=	%=	//=

序列解包赋值

具体内容请参看后面的序列解包赋值这一小节4.5.1。

可迭代对象的迭代赋值

在我们对 python 语言有了深入的了解之后，我们发现 python 中迭代思想是深入骨髓的。我们在前面接触了序列的赋值模式之后，发现似乎这种赋值除了临时创建右边的序列之外，还似乎与迭代操作有关，于是我们推测 python 的这种平行赋值模式可以扩展到可迭代对象，然后我们发现确实如此！

```
>>> x,y,z= map(lambda x : x+2,[-1,0,1])
>>> print(x,y,z)
1 2 3
```

最后要强调一点的是确保变量名和后面的可迭代对象的输出元素数目是一致的，当然进一步扩展的序列解包赋值也是支持的：

```
>>> x,y,*z= map(lambda x : x+2,[-1,0,1,2])
>>> print(x,y,z)
1 2 [3, 4]
```

通配赋值，我喜欢这样称呼了，通配之后收集的元素在列表里面；而函数参数的通配传递，收集的元素是在元组里面。

最后我们总结到，可迭代对象的赋值就是迭代操作加上各个元素的一对一的赋值操作。

数值

python 的数值的内置类型有：int，float，complex 等[①]。python 的基本算术运算操作有加减乘除（+ - * /）。然后‘=’表示赋值，类似数学书上的中缀表达式和优先级和括号法则等，这些都是一般编程语言说到烂的东西了。

```
print((1+2)*(10-5)/2)
print(2**100)
```

二进制八进制十六进制

二进制的数字以 0b（零比）开头，八进制的数字以 0o（零哦）开头，十六进制的数字以 0x（零艾克斯）开头。

```
0b101010, 0o177, 0x9ff
```

① 这些 int、float 等命令都是强制类型转换命令

以二进制格式查看数字使用 bin 命令，以十六进制查看数字使用 hex 命令。

```
>>> bin(42)
'0b101010'
>>> hex(42)
'0x2a'
```

进制转换小程序

```
number=input(" 请输入一个数字: ")
number= eval(number)
#
radix= input(''' 请输入你想转换的进制系统
2   表示   二进制
8   表示   八进制
16  表示   十六进制
''')
radix =eval(radix)

while True:
    if radix == 2:
        print(bin(number))
        break
    elif radix == 8:
        print(oct(number))
        break
    elif radix == 16:
        print(hex(number))
```

```
        break
    else:
        print("sorry you input the wrong radix")
```

程序运行的情况如下所示：

```
请输入一个数字：20
请输入你想转换的进制系统
2    表示    二进制
8    表示    八进制
16   表示    十六进制
8
0o24
```

此外基于字符串的进制转换可以用字符串的 format 方法来处理之。

数学幂方运算

x^y，x 的 y 次方如上面第二行所述就是用x**y 这样的形式即可。此外 pow 函数作用是一样的，pow(x,y)。

数值比较

数值比较除了之前提及的 >，<，== 之外，>=，<=，!= 也是有的（大于等于，小于等于，不等于）。此外 python 还支持连续比较，就是数学格式 $a < x < b$，x 在区间 (a, b) 的判断。在 python 中可以直接写成如下形式：a<x<b。这实际实现的过程就是两个比较操作的进一步与操作。

相除取商或余

就作为正整数相除使用`x//y` 得到的值意义还是很明显的就是**商**。带上负号感觉有点怪了，这里先略过。相关的还有**取余**数，就是`x%y`，这样就得到 x 除以 y 之后的余数了，同样带上负号情况有变，这里先略过。

复数

python 直接支持复数，复数的写法是类似`1+2j` 这样的形式，然后如果 z 被赋值了一个复数，这样它就是一个复数类型，那么这个类具有两个属性量，**real** 和 **imag**。也就是使用`z.real` 就给出这个复数的实数部。imag 是 imaginary number 的缩写，虚数，想像出来的数。

abs 函数

大家都知道 abs 函数是绝对值函数，这个 python 自带的，不需要加载什么模块。作用于复数也是可以的：

```
z=3+4j
print(z.real,z.imag)
print(abs(z))
```

这个和数学中复数绝对值的定义完全一致，也就是复数的模：
$|z| = \sqrt{a^2 + b^2}$

round 函数

简单的理解就是这个函数实现了对数值的四舍五入功能。

```
>>> round(3.1415926)
3
>>> round(3.1415926,0)
3.0
>>> round(3.1415926,1)
3.1
>>> round(3.1415926,2)
3.14
>>> round(3.1415926,4)
3.1416
```

这里第二个参数接受 0 或者负数多少有点没意义了，一般使用还是取 1 或大于 1 的数吧，意思就是保留几位小数。

min，max 和 sum 函数

min，max 函数的用法和 sum 的用法稍微有点差异，简单起见可以认为 min，max，sum 都接受一个元组或者列表（还有其他？），然后返回这个元组或者列表其中的最小值，最大值或者相加总和。此外 min 和 max 还支持 min(1,2,3) 这样的形式，而 sum 不支持。

```
>>> min((1,6,8,3,4))
1
>>> max([1,6,8,3,4])
8
>>> sum([1,6,8,3,4])
22
>>> min(1,6,8,3,4)
1
```

位操作

python 支持位操作的，这里简单说一下：位左移操作 «，位与操作 &，位或操作 |，位异或操作 ^。

```
>>> x=0b0001
>>> bin(x << 2)
'0b100'
>>> bin(x | 0b010)
'0b11'
>>> bin(x & 0b1)
'0b1'
>>> bin(x ^ 0b101)
'0b100'
```

math 模块

在`from math import *`之后，可以直接用符号 pi 和 e 来引用圆周率和自然常数。此外 math 模块还提供了很多数学函数，比如：

sqrt 开平方根函数，sqrt(x)。

sin 正弦函数，类似的还有 cos，tan 等，sin(x)。

degrees 将弧度转化为角度，三角函数默认输入的是弧度值。

radians 将角度转化位弧度，radians(30)。

log 开对数，log(x,y)，即 $\log_y x$，y 默认是 e。

exp 指数函数，exp(x)。

pow 扩展了内置方法，现在支持 float 了。pow(x,y)

这里简单写个例子:

```
>>> from math import *
>>> print(pi)
3.141592653589793
>>> print(sqrt(85))
9.219544457292887
>>> print(round(sin(radians(30)),1))#sin(30°)
0.5
```

更多内容请参见官方文档。

random 模块

random 模块提供了一些函数来解决随机数问题。

random random 函数产生 0 到 1 之间的随机实数（包括 0）。

random()->[0.0, 1.0)。

uniform uniform 函数产生从 a 到 b 之间的随机实数（a，b 的值指定，包括 a。）。

uniform(a,b)->[a.0, b.0)。

randint randint 函数产生从 a 到 b 之间的随机整数，包含 a 和 b。

randint(a,b)->[a,b]

choice choice 随机从一个列表或者字符串中取出一个元素。

randrange randrange 函数产生从 a 到 b 之间的随机整数，步长为 c（a，b，c 的值指定，相当于 choice(range(a,b,c))。整数之间就用 randint 函数吧，这里函数主要是针对 range 函数按照步长从而生成一些整数序列的情况。

sample(p,k) sample 函数从 p 中随机选取唯一的元素（p 一般是 range(n) 或集合之类的，这里所谓的唯一的意思就是不放回抽样的意思，但如果 p 样品里面有重复的元素，最后生成的列表还是会有重复的元素的。）然后组成 k 长度的列表返回。

下面是一个简单的例子：

```
>>> from random import *
>>> print(random())
0.36882919781549717
>>> print(uniform(1,10))
2.771065174892699
>>> print(randrange(1,6))
1
>>> print(randint(1,10))
3
>>> print(choice('abcdefghij'))
j
>>> print(choice(['①','②','③']))
①
```

作为随机实数，所谓开始包含的那个临界值可能数学意义大于实际价值，你可以写一个类似下面的小脚本看一下，随机实数是很难随机到某个具体的数的。

```
from random import *
i = 0
while True:
    x = uniform(0,2)
    if x == 0:
```

```
            print(i)
            break
        else:
            print(x)
            i += 1
```

从上一个例子我们看到，虽然我不确定随具体随机到某个实数的概率是不是永远也没有可能，但肯定很小很小。所以如果我们要解决某个问题，需要某个确定的概率的话还是用随机整数好一些。

更多内容请参见官方文档。

statistics 模块

这个模块 python3.4 才加入进来。

上面的那个例子这里稍作修改，使之成为一个骰子模拟器。其中`i_list` 这个列表收集多次实验中掷多少次骰子才遇到 6 的次数。

```
from random import *
i_list = []
while len(i_list) < 100:
    i = 1
    while True:# 一次实验
        x = randint(1,6)
        if x == 6:
            print('times:' , i)
            break
        else:
            print(x)
            i += 1
```

```
    i_list.append(i)


print(i_list)
from statistics import *
print(mean(i_list))# 平均值
print(median(i_list))# 中位数，去掉最高最低...
```

statistics 模块中的 **mean** 函数接受一组数值列表，然后返回这组数值的平均值。而 **median** 函数返回的是统计学上所谓的中位数，你可以简单看作一组数字不断的去掉一个最高和最低，然后剩下来的一个或者两个（两个要取平均值）的数值的值。

更多内容请参见官方文档。

序列

字符串，列表，元组（tuple，这里最好翻译成元组，因为里面的内容不一定是数值。）都是序列（sequence）的子类，所以序列的一些性质他们都具有，最好在这里一起讲方便理解记忆。

len 函数

len 函数返回序列所含元素的个数：

```
string001='string'
list001=['a','b','c']
tuple001=(1,2,3,4)


for x in [string001,list001,tuple001]:
```

```
    print(len(x))
```

```
6
3
4
>>>
```

调出某个值

对于序列来说后面跟个方括号，然后加上序号（程序界的老规矩，从 0 开始计数。），那么调出对应位置的那个值。还以上面那个例子来说明。

```
string001='string'
list001=['a','b','c']
tuple001=(1,2,3,4)

for x in [string001,list001,tuple001]:
    print(x[2])
```

```
r
c
3
>>>
```

倒着来

倒着来计数 -1 表示倒数第一个，-2 表示倒数第二个。依次类推。

code2-1

```
string001='string'
list001=['a','b','c']
tuple001=(1,2,3,4)

for x in [string001,list001,tuple001]:
    print(x[-1],x[-2])
```

```
g n
c b
4 3
```

调出多个值

前面不写表示从头开始，后面不写表示到达尾部。中间加个冒号的形式表示从那里到那里。这里**注意**后面那个元素是不包括进来，看来 python 区间的默认含义都是包头不包尾。这样如果你想要最后一个元素也进去，只有使用默认的不写形式了。

code2-2

```
string001='string'
list001=['a','b','c']
tuple001=(1,2,3,4)

for x in [string001,list001,tuple001]:
    print(x[1:3],x[-2:-1],x[:-1],x[1:],x[1:-1])
```

```
tr n strin tring trin
['b', 'c'] ['b'] ['a', 'b'] ['b', 'c'] ['b']
```

(2, 3) (3,) (1, 2, 3) (2, 3, 4) (2, 3)

用数学半开半闭区间的定义来理解这里的包含关系还是很便捷的。

1. 首先是数学半开半闭区间，左元素和右元素都是之前叙述的对应的定位点。左元素包含右元素不包含。
2. 其次方向应该是从左到右，如果定义的区间是从右到左，那么将产生空值。
3. 如果区间超过，那么从左到右包含的所有元素就是结果，不会返回错误。
4. 最后如果左右元素定位点相同，那么将产生空值，比如：`string001[2:-4]`，其中 2 和 -4 实际上是定位在同一个元素之上的。额外值得一提的列表插入操作，请参看列表的插入操作这一小节。2.5.1

序列反转

这是 python 最令人叹为观止的地方了，其他的语言可能对列表啊什么的反转要编写一个复杂的函数，我们 python 有一种令人感动的方法。

code2-3

```
string001='string'
list001=['a','b','c']
tuple001=(1,2,3,4)

for x in [string001,list001,tuple001]:
    print(x[::-1])
```

gnirts

['c', 'b', 'a']

(4, 3, 2, 1)

之前在 range 函数的介绍时提及序列的索引和 range 函数的参数设置很是类似，这是我们可以参考理解之，序列（列表，字符串等）的索引参数 [start:end:step] 和 range 函数的参数设置一样，第一个参数是起步值，第二个参数是结束值，第三个参数是步长。这里 end 不填都好理解，就是迭代完即可，不过如果 step 是负数，似乎起点不填默认的是 -1。

然后 range 函数生成的迭代器对象同样接受这种索引参数语法，看上去更加的怪异了：

```
>>> range(1,10,2)
range(1, 10, 2)
>>> range(1,10,2)[::-2]
range(9, -1, -4)

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

我们可以看到对 range 函数进行切片操作之后返回的仍然是一个 range 对象，经过了一些修正。似乎这种切片操作和类的某个特殊方法有关，和 python 的 slice 对象有关。

序列的可更改性

字符串不可以直接更改，但可以组合成为新的字符串；列表可以直接更改；元组不可以直接更改。

序列的加法和减法

两个字符串相加就是字符串拼接了。乘法就是加法的重复，所以一个字符串乘以一个数字就是自己和自己拼接了几次。列表还有元组和字符串一样大致情况类似。

code2-4

```
print('abc'+'def')
print('abc'*3)
print([1,2,3]+[4,5,6])
print((0,'a')*2)
```

```
abcdef
abcabcabc
[1, 2, 3, 4, 5, 6]
(0, 'a', 0, 'a')
```

字符串

python 语言不像 c 语言字符和字符串是不分的，用单引号或者双引号包起来就表示一个字符串了。单引号和双引号的区别是一般用单引号，如果字符串里面有单引号，那么就使用双引号，这样单引号直接作为字符处理而不需要而外的转义处理——所谓转义处理和其他很多编程语言一样用 \ 符号。比如要显示' 就输入\'。

三单引号和三双引号

在单引号或者双引号的情况下，你可以使用\n 来换行，其中 \n 表示换行。此外还可以使用三单引号''' 或者三双引号""" 来包围横跨多行的字符串，其中换

行的意义就是换行，不需要似前面那样的处理。

```
print('''\
这是一段测试文字
  this is a test line
      其中空白和      换行都所见所得式的保留。''')
```

startswith 方法

```
>>> x = 'helloABC'
>>> x
'helloABC'
>>> x.startswith('hello')
True
>>> x.endswith('ABC')
True
```

startswith 测试字符串是否以某个子字符串开始

endswith 测试某个字符串是否以某个子字符串结束

find 方法

字符串的 find 方法可用来查找某个子字符串，没有找到返回 -1，找到了返回字符串的偏移量。用法就是：`s.find('d')`。

replace 方法

字符串的 replace 方法进行替换操作，接受两个参数：第一个参数是待匹配的子字符串，第二个参数是要替换成为的样子。

```
>>> print('a b 11 de'.replace('de','ding'))
a b 11 ding
>>> print('1,1,5,4,1,6'.replace('1','replaced'))
replaced,replaced,5,4,replaced,6
```

upper 方法

将字符串转换成大写形式。

```
>>> str='str'
>>> str.upper()
'STR'
```

类似的还有:

lower 都变成小写

capitalize 首字母大写，其它都小写。

isdigit 方法

isdigit 测试是不是数字

isalpha 测试是不是字母

isalnum 测试是不是数字或字母

值得注意的是就算是字母组成的语句，如果中间有空格 isalpha 方法也会返回 False。

split 方法

字符串的 split 方法可以将字符串比如有空格或者逗号等分隔符分割而成，可以将其分割成子字符串列表。默认是空格是分隔符。

```
>>> string='a=1,b=2,c=3'
>>> string.split(',')
['a=1', 'b=2', 'c=3']
```

splitline 方法

把一个字符串按照行分开。这个可以用上面的 split 方法然后接受\n 参数来实现，所不同的是 splitline 方法不需要接受参数：

```
>>> string
'this is line one\nthis is line two\nthis is line three'
>>> string.splitlines()
['this is line one', 'this is line two', 'this is line three']
>>> string.split('\n')
['this is line one', 'this is line two', 'this is line three']
```

join 方法

字符串的 join 方法非常有用，严格来说它接受一个迭代器参数，不过最常见的是列表。将列表中的多个字符串连接起来，我们看到他采用了一种非常优雅的方式，就是只有两个字符串之间才插入某个字符，这正是我们所需要的。具体例子如下所示：

```
>>> list001=['a','b','c']
>>> "".join(list001)
'abc'
>>> ','.join(list001)
'a,b,c'
```

strip 方法

rstrip 方法

字符串右边的空格都删除。换行符也会被删除掉。

lstrip 方法

类似 rstrip 方法，字符串左边的空格都删除。换行符也会被删除掉。

format 方法

字符串的 format 方法方便对字符串内的一些变量进行替换操作，其中花括号不带数字跟 format 方法里面所有的替换量，带数字 0 表示第一个替换量，后面类推。此外还可以直接用确定的名字引用。

```
>>> print('1+1={0}, 2+2={1}'.format(1+1,2+2))
1+1=2, 2+2=4
>>> print('my name is {name}'.format(name='Jim T Kirk'))
my name is Jim T Kirk
```

转义和不转义

\n　　\t　这是一般常用的转义字符，换行和制表。此外还有\\ 输出 \ 符号。

如果输出字符串不想转义那么使用如下格式：

```
>>> print(r'\t \n \test')
\t \n \test
```

count 方法

统计字符串中某个字符或某一连续的子字符串出现的次数。

```
>>> string = 'this is a test line.'
>>> string.count('this')
1
>>> string.count('t')
3
```

列表

方括号包含几个元素就是列表。

列表的插入操作

字符串和数组都不可以直接更改所以不存在这个问题，列表可以。其中列表还可以以一种定位在相同元素的区间的方法来实现插入操作，这个和之前理解的

区间多少有点违和，不过考虑到定位在相同元素的区间本来就概念模糊，所以在这里就看作特例，视作在这个定位点相同元素之前插入吧。

code2-5

```
list001=['one','two','three']
list001[1:-2]=['four','five']
print(list001)
```

```
['one', 'four', 'five', 'two', 'three']
```

extend 方法似乎和列表之间的加法重合了，比如 list.extend([4,5,6]) 就和 list=list+[4,5,6] 是一致的，而且用加法表示还可以自由选择是不是覆盖原定义，这实际上更加自由。

insert 方法也就是列表的插入操作：

```
>>> list = [1,2,3,4]
>>> list.insert(0,5)
>>> list
[5, 1, 2, 3, 4]
>>> list.insert(2,'a')
>>> list
[5, 1, 'a', 2, 3, 4]
```

append 方法

python 的 append 方法就是在最后面加**一个元素**，如果你 append 一个列表那么这一个列表整体作为一个元素。然后 append 方法会永久的改变了该列表对象的值。

记住，append 等等原处修改列表的方法都是没有返回值的。

```
>>> list = [1,2,3,4]
>>> list.append(5)
>>> list
[1, 2, 3, 4, 5]
```

如果你希望不改动原列表的附加，请使用加法来操作列表。

reverse 方法

reverse 方法不接受任何参数，直接将一个列表永久性地翻转过来。如果你希望不改变原列表的翻转，有返回值，请使用如下方法：

```
>>> list
[1, 2, 3, 4, 5]
>>> listNew = list[::-1]
>>> list
[1, 2, 3, 4, 5]
>>> listNew
[5, 4, 3, 2, 1]
```

copy 方法

copy 方法复制返回本列表。

sort 方法

也就是排序，永久性改变列表。默认是递增排序，可以用 **reverse=True** 来调成递减排序。

默认的递增排序顺序如果是数字那么意思是数字越来越大，如果是字符那么（似乎）是按照 ACSII 码编号递增来排序的。如果列表一些是数字一些是字符会报错。

```
>>> list = ['a','ab','A','123','124','5']
>>> list.sort()
>>> list
['123', '124', '5', 'A', 'a', 'ab']
```

sort 方法很重要的一个可选参数 **key=function**，这个 function 函数就是你定义的函数（或者在这里直接使用 lambda 语句。），这个函数只接受一个参数，就是排序方法（在迭代列表时）接受的当前的那个元素。下面给出一段代码，其中 tostr 函数将接受的对象返回为字符，这样就不会出错了。

code2-6

```
def tostr(item):
    return str(item)

list001 = ['a','ab','A',123,124,5]

list001.sort(key=tostr)

print(list001)
```

```
[123, 124, 5, 'A', 'a', 'ab']
```

sorted 函数

sorted 函数在这里和列表的 sort 方法最大的区别是它返回的是一个新的列表而不是原处修改。其次 sorted 函数的第一个参数严格来说是所谓的可迭代对

象，也就是说它还可以接受除了列表之外的比如元组字典等可迭代对象。至于用法他们两个差别不大。

```
>>> sorted((1,156,7,5))
[1, 5, 7, 156]
>>> sorted({'andy':5,'Andy':1,'black':9,'Black':55},key=str.lower)
['Andy', 'andy', 'black', 'Black']
```

上面第二个例子调用了 **str.lower** 函数，从而将接受的 item，这里比如说'Andy'，转化为 andy，然后参与排序。也就成了对英文字母大小写不敏感的排序方式了。

字典按值排序　同样类似的有字典按值排序的方法[①]：

```
>>> sorted({'andy':5,'Andy':1,'black':9,'Black':55}.items(),key=lambda i: i[1])
[('Andy', 1), ('andy', 5), ('black', 9), ('Black', 55)]
```

这个例子先用字典的 items 方法处理返回 (key,value) 对的可迭代对象，然后用后面的 lambda 方法返回具体接受 item 的值，从而根据值来排序。

中文排序　下面这个例子演示了如何对中文名字排序。整个函数的思路就是用pypinyin（一个第三方模块），将中文姓名的拼音对应出来，然后组成一个列表，然后根据拼音对这个组合列表排序，然后生成目标列表。

① 参考网站

```
list001=[' 张三',' 李四',' 王二',' 麻子',' 李二',' 李一']
def zhsort(lst):
    from pypinyin import  lazy_pinyin
    pinyin=[lazy_pinyin(lst[i]) for i in range(len(lst))]
    lst0=[(a,b) for (a,b) in zip(lst,pinyin)]
    lst1= sorted(lst0, key=lambda d:d[1])
    return [x[0] for x in lst1]
print(zhsort(list001))
```

```
[' 李二', ' 李四', ' 李一', ' 麻子', ' 王二', ' 张三']
```

删除某个元素

- 赋空列表值，相当于所有元素都删除了。
- pop 方法：接受一个参数，就是列表元素的定位值，然后那个元素就删除了，方法并返回那个元素的值。如果不接受参数默认是删除最后一个元素。
- remove 方法：移除第一个相同的元素，如果没有返回相同的元素，返回错误。
- del 函数：删除列表中的某个元素。

```
>>> list001=['a','b','c','d','e']
>>> list001.pop(2)
'c'
>>> list001
['a', 'b', 'd', 'e']
>>> list001.pop()
'e'
```

```
>>> list001
['a', 'b', 'd']
>>> list001.remove('a')
>>> list001
['b', 'd']
>>> del list001[1]
>>> list001
['b']
```

count 方法

统计某个元素出现的次数。

```
>>> list001=[1,'a',100,1,1,1]
>>> list001.count(1)
4
```

index 方法

index 方法返回某个相同元素的偏移值。

```
>>> list001=[1,'a',100]
>>> list001.index('a')
1
```

列表解析

我们来看下面这个例子:

code2-7

```
def square(n):
    return n*n

print(list(map(square,[1,2,3,4,5])))
print([square(x) for x in [1,2,3,4,5]])
```

```
[1, 4, 9, 16, 25]
[1, 4, 9, 16, 25]
```

map 函数将某个函数应用于某个列表的元素中并生成一个 map 对象（可迭代对象），需要外面加上 list 函数才能生成列表形式。第二种方式更有 python 风格，是推荐使用的列表解析方法。

在 python 中推荐多使用迭代操作和如上的列表解析风格，因为 python 中的迭代操作是直接用 c 语言实现的。

列表解析加上过滤条件

for 语句后面可以跟一个 if 子句表示过滤条件，看下面的例子来理解吧：

```
>>> [s*2 for s in ['hello','abc','final','help'] if s[0] == 'h']
['hellohello', 'helphelp']
```

这个例子的意思是列表解析，找到的元素进行乘以 2 的操作，其中过滤条件为字符是 h 字母开头的，也就是后面 if 表达式不为真的元素都被过滤掉了。

完整的列表解析结构

下面给出一个完整的列表解析结构，最常见的情况一般就一两个 for 语句吧，这里 if 外加个括号是可选项的意思。

```
[ expression for var1 in iterable1 [if condition1 ]
                    for var2 in iterable2 [if condition2 ]
                    ........
                           ]
```

这里的逻辑是从左到右第一个 for 语句就是最先执行的 for 语句，然后是第二个 for 语句跟着执行。

这里的 iterable1 是指某个可迭代对象，也就是说那些能够返回可迭代对象的函数比如 map，filter，zip，range 等函数都可以放进去。不过我们要克制自己在这里别写出太过于晦涩的程序了。还有 for 循环语句也别嵌套太多了，这样就极容易出错的。

下面这个程序大家看看：

```
>>> [x+str(y) for x in ['a','b','c'] for y in [1,2,3,4,5,6] if y & 1]
['a1', 'a3', 'a5', 'b1', 'b3', 'b5', 'c1', 'c3', 'c5']
>>> [x+str(y) for x in ['a','b','c'] for y in [1,2,3,4,5,6] if not  y & 1]
['a2', 'a4', 'a6', 'b2', 'b4', 'b6', 'c2', 'c4', 'c6']
```

列表解析的好处

在熟悉列表解析的语句结构之后，一两个 for 语句不太复杂的情况下，还是很简单明了的。同时语法也更加精炼，同时运行速度较 for 循环要至少快上一倍。最后 python 的迭代思想深入骨髓，以后 python 的优化工作很多都围绕迭代展开，也就是多用列表解析会让你的代码以后可能运行的更快。

有这么多的好处，加上这么 cool 的 pythonic 风格，推荐大家多用列表解析风格解决问题。

元组的生成

这个时候需要明确加个括号表示这是一个元组对象。

```
>>> [(x,x**2) for x in range(5)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16)]
```

for 语句中列表可变的影响

一般情况 for 迭代某个可迭代对象就是可迭代对象返回一个值然后利用这个值赋值并进行下面的操作，但是列表却是一个可变的东西，如果列表在操作中被修改了，情况会怎样呢？

```
lst = [1,2,3,4,5]
index = 0
for x in lst:
    lst.pop(index)
    print(x)
```

```
1
3
5
```

具体这个过程的细节我不清楚，但确定的是在这里 for 语句并没有记忆原列表，而只是记忆了返回次数或者偏移值。

列表元素替换

推荐用列表解析方法来实现列表元素的替换功能。

```
def replace(x,a,b):
    if x == a:
        return b
    else:
        return x


lst=[1,5,4,1,6]
```

```
>>> [replace(i,1,'replaced') for i in lst]
['replaced', 5, 4, 'replaced', 6]
```

列表元素去重

列表元素去重推荐用后来的 set 集合对象来处理之，其会自动去除重复的元素。请参看下面的集合一小节2.7。

```
>>> lst = [1,2,3,4,5,1,2,3,4,5]
>>> [i for i in set(lst)]
[1, 2, 3, 4, 5]
```

字典

与列表一样字典是可变的，可以像列表一样引用然后原处修改，del 语句也适用。

创建字典

字典是一种映射，并没有从左到右的顺序，只是简单地将键映射到值。字典的声明格式如下：

```
dict001={'name':'tom','height':'180','color':'red'}
dict001['name']
```

或者创建一个空字典，然后一边赋值一边创建对应的键：

```
dict002={}
dict002['name']='bob'
dict002['height']=195
```

所以对字典内不存在的键赋值是可行的。

根据列表创建字典

如果是 [['a',1],['b',2],['c',3]] 这样的形式，那么直接用 dict 函数处理就变成字典了，如果是 ['a','b','c'] 和 [1,2,3] 这样的形式那么需要用 zip 函数处理一下，然后用 dict 函数处理一次就变成字典了：

```
>>> lst
[['a', 1], ['b', 2], ['c', 3]]
>>> dict001=dict(lst)
>>> dict001
{'a': 1, 'b': 2, 'c': 3}
```

zip 函数的例子请参看后面的2.6.8。

字典里面有字典

和列表的不同就在于字典的索引方式是根据“键”来的。

```
dict003={'name':{'first':'bob','second':'smith'}}
dict003['name']['first']
```

字典遍历操作

字典特定顺序的遍历操作的通用做法就是通过字典的 keys 方法收集键的列表，然后用列表的 sort 方法处理之后用 for 语句遍历，如下所示：

```
dict={'a':1,'c':2,'b':3}
dictkeys=list(dict.keys())
dictkeys.sort()
for key in dictkeys:
    print(key,'->',dict[key])
```

警告：上面的例子可能对 python 早期版本并不使用，因为 python 中一大规则是对对象的原处修改的函数并没有返回值。上面的语句只是到了 python3 后期才能适用，保险起见，推荐使用 sorted 函数，sorted 函数是默认对字典的键进行排序并返回键的值组成的列表。

```
dict={'a':1,'c':3,'b':2}
>>> for key in sorted(dict):
...   print(key,'->',dict[key])
...
a -> 1
```

```
b -> 2
c -> 3
```

如果你对字典遍历的顺序没有要求，那么就可以简单的这样处理：

```
>>> for key in dict:
...     print(key,'->',dict[key])
...
c -> 2
a -> 1
b -> 3
```

keys 方法

收集键值，返回可迭代对象。

values 方法

和 keys 方法类似，收集的值，返回可迭代对象。

```
>>> dict001.values()
dict_values([3, 1, 2])
>>> list(dict001.values())
[3, 1, 2]
```

items 方法

和 keys 和 values 方法类似，不同的是返回的是 (key,value) 对的可迭代对象。

```
>>> dict001.items()
dict_items([('c', 3), ('a', 1), ('b', 2)])
>>> list(dict001.items())
[('c', 3), ('a', 1), ('b', 2)]
```

字典的 in 语句

可以看到 in 语句只针对字典的键，不针对字典的值。

```
>>> dict001={'a':1,'b':2,'c':3}
>>> 2 in dict001
False
>>> 'b' in dict001
True
```

字典对象的 get 方法

get 方法是去找某个键的值，为什么不直接引用呢，get 方法的好处就是某个键不存在也不会出错。

```
>>> dict001={'a':1,'b':2,'c':3}
>>> dict001.get('b')
2
>>> dict001.get('e')
```

update 方法

感觉字典就是一个小型数据库，update 方法将另外一个字典里面的键和值覆盖进之前的字典中去，称之为更新，没有的加上，有的覆盖。

```
>>> dict001={'a':1,'b':2,'c':3}
>>> dict002={'e':4,'a':5}
>>> dict001.update(dict002)
>>> dict001
{'c': 3, 'a': 5, 'e': 4, 'b': 2}
```

pop 方法

pop 方法类似列表的 pop 方法，不同引用的是键，而不是偏移地址，这个就不多说了。

字典解析

这种字典解析方式还是很好理解的。

```
>>> dict001={x:x**2 for x in [1,2,3,4]}
>>> dict001
{1: 1, 2: 4, 3: 9, 4: 16}
```

zip 函数创建字典

可以利用 zip 函数来通过两个可迭代对象平行合成一个配对元素的可迭代对象，然后用 dict 函数将其变成字典对象。具体的理解请参看深入理解 python3 的迭代这一章8。

```
>>> dict001=zip(['a','b','c'],[1,2,3])
>>> dict001
<zip object at 0xb7055eac>
>>> dict001=dict(dict001)
>>> dict001
{'c': 3, 'b': 2, 'a': 1}
```

集合

python 实现了数学上的无序不重复元素的集合概念，在前面讨论列表去重元素的时候我们提到过正好可以利用集合的这一特性。

```
>>> list001=[1,2,3,1,2,4,4,5,5,5,7]
>>> {x for x in list001}
{1, 2, 3, 4, 5, 7}
>>> set(list001)
{1, 2, 3, 4, 5, 7}
```

用集合解析的形式表示出来就是强调 set 命令可以将任何可迭代对象都变成集合类型。当然如果我们希望继续使用列表的话使用 list 命令强制类型转换为列表类型即可，不过如果我们在应用中确实一致需要元素不重复这一特性，就可以考虑直接使用集合作为主数据操作类型。

集合也是可迭代对象。关于可迭代对象可以进行的列表解析操作等等就不啰嗦了。下面介绍集合的一些操作。

集合添加元素

值得一提的是如果想创建一个空的集合，需要用 set 命令，用花括号系统会认为你创建的是空字典。然后我们看到用集合的 **add** 方法添加，那些重复的元素是添加不进来的。

警告：值得一提的是集合只能包括不可变类型，因此列表和字典不能作为集合内部的元素。元组不可变，所以可以加进去。还有集合也是不可以包括进去的，觉得这点好逊啊，数学里面的集合概念能够包含集合那是基本的特性啊，感觉这点不修正好还是用列表方便些。

```
>>> set001=set()
>>> set001.add(1)
>>> set001
{1}
>>> set001.add(2)
>>> set001
{1, 2}
>>> set001.add(1)
>>> set001
{1, 2}
```

或者使用 update 方法一次更新多个元素：

```
>>> set001=set('a')
>>> set001.update('a','b','c')
>>> set001
{'b', 'a', 'c'}
```

集合去掉某个元素

有两个集合对象的方法可以用于去掉集合中的某个元素，discard 方法和 remove 方法，其中 discard 方法如果删除集合中没有的元素那么什么都不会发生，而 remove 方法如果删除某个不存在的元素那么会产生 KeyError。

```
>>> set001=set('hello')
>>> set001.discard('h')
>>> set001
{'e', 'o', 'l'}
>>> set001.discard('l')
>>> set001
{'e', 'o'}
```

remove 方法与之类似就不做演示了。

两个集合之间的关系

子集判断

集合对象有一个 issubset 方法用于判断这个集合是不是那个集合的子集。

```
>>> set001=set(['a','b'])
>>> set002=set(['a','b','c'])
>>> set001.issubset(set002)
True
```

还有更加简便的方式比较两个集合之间的关系，那就是 >，<，>=，<=，== 这样的判断都是适用的。也就是 set001 是 set002 的子集，它

的元素 set002 都包含，那么 set001<=set002，然后真子集的概念就是 set001<set002 即不等于即可。

两个集合之间的操作

下面的例子演示的是两个集合之间的交集：&，并集：|，差集：-。

```
>>> set001=set('hello')
>>> set002=set('hao')
>>> set001 & set002 # 交集
{'o', 'h'}
>>> set001 | set002 # 并集
{'h', 'l', 'a', 'e', 'o'}
>>> set001 - set002 # 差集
{'e', 'l'}
```

类似的集合对象还有 intersection 方法，union 方法，difference 方法：

```
>>> set001=set('hello')
>>> set002=set('hao')
>>> set001.intersection(set002) # 交集
{'h', 'o'}
>>> set001.union(set002) # 并集
{'e', 'a', 'h', 'o', 'l'}
>>> set001.difference(set002) # 差集
{'e', 'l'}
```

clear 方法

将一个集合清空。

copy 方法

类似列表的 copy 方法，制作一个集合 copy 备份然后赋值给其他变量。

pop 方法

无序弹出集合中的一个元素，直到没有然后返回 KeyError 错误。

元组

圆括号包含几个元素就是元组 (tuple)。元组和列表的不同在于元组是不可改变。元组也是从属于序列对象的，元组的很多方法之前都讲了。而且元组在使用上和列表极其接近，有很多内容这里也略过了。

值得一提的是如果输入的时候写的是 *x,y* 这样的形式，实际上表达式就加上括号了，也就是一个元组了 *(x,y)*。

生成器表达式

类似列表解析，如果元组在这里解析也是返回的元组吗？这里并不是如此，前面谈到 python 中一般表达式的圆括号是忽略了的，所以这里的元组解析表示式有个更专门的名字叫做生成器表达式，它返回的是生成器对象，和生成器函数具体调用之后返回的对象是一样的。生成器对象具有__next__ 方法，可以调用 next 函数。

```
>>> x = [i for i in [1,2,3]]
>>> x
[1, 2, 3]
>>> y = (i for i in [1,2,3])
```

```
>>> y
<generator object <genexpr> at 0xb70dbe8c>
```

bytes 类型

基本编码知识

具体存储在计算机里面的都是二进制流，而如果要将其正确解析成为对应的字符，是需要建立一定的编码规则的，比如大家熟悉的 ASCⅡ 编码规则。ACSⅢ 编码是 Latin-1 和 utf-8 等编码的子集，也就是一连串基于 ACSⅡ 编码的字符串用 utf-8 编码也能正确解析。

python2 中目前也支持 bytes 类型了，但其只是 str 类型的一个别名[①] 。然后 python2 还有一个 unicode 类型，由于 python3 字符串默认是 unicode 编码的，所以 python3 中的 str 可以对应 python2 的 unicode。此外还有一个 bytearray 类型，目前 python2 也加入进来了，差别不大。

就实现上具体 python2 和 python3 底层还有什么区别不大清楚，而且大家都承认 python3 定义字符串 str 和字节流 bytes 这两个名字都是很好的。只是因为 python2 和 python3 在这块领域具体功能都差不多，而因为这种转变带来了困扰很多，可能也是人们迟迟不愿意接受 python3 的原因吧。

bytes 简单的理解就是没有任何字符含义的二进制字节流。然后如这样 `b'test'` ，在前面加个字符 b 或者 B，其将解析为 bytes 类型。

```
>>> x = b'test'
>>> x
b'test'
```

① 参考这个网页。

```
>>> type(x)
<class 'bytes'>
>>> x[0]
116
>>> x[1]
101
>>> list(x)
[116, 101, 115, 116]
>>>
```

python 在打印时会尽可能打印可见字符，尽管上面的 x 打印显示出了具体的 test 这个字符，但我们应该认为 x 是一连串的数字序列而不具有任何字符串含义，如果我们调用 bytes 类型的 **decode** 方法，那么 bytes 类型解码之后将变成 str 类型。

```
>>> y = x.decode('utf-8')
>>> y
'test'
>>> type(y)
<class 'str'>
```

当然具体编码方式是否正确，是否正确解析了原 bytes 字节流那又是另外一回事了。比如还可能是 big5 或者 GB 什么的编码。

此外字符串 str 类型有个 **decode** 方法可以进行编码操作从而输出对应编码的 bytes 字节流。

使用方法

我们可以如下看一下 str 类型和 bytes 类型具体有那些方法差异:

```
>>> set(dir('abc')) - set(dir(b'abc'))
{'isdecimal', 'casefold', '__rmod__', 'format_map', 'format', 'encode', '__mod__', 'isnu
>>> set(dir(b'abc')) - set(dir('abc'))
{'decode', 'fromhex'}
```

我们看到 bytes 和 str 几乎拥有相同的功能，所以大部分之前学到的用于 str 字符串类型的那些方法同样可以用于 bytes 类型中。这多少有点方法泛滥了，因为 bytes 是字节流类型，内在是没有字符含义的，可能某些方法并不推荐使用。

比如下面的 upper 方法和 replace 方法:

```
>>> b't'.upper()
b'T'
>>> b'testst'.replace(b'st',b'oo')
b'teoooo'
```

replace 方法还可以接受，但 upper 方法有点过了。

然后字节流的连接可以很方便的用加法或 join 方法来进行，如下所示:

```
>>> b't' + b'e'
b'te'
>>> b''.join([b'a',b'c'])
b'ac'
```

但是要注意，python2 里面不管是加法还是 join 方法都将丢掉那个 b 修饰符[①]:

① 参考了这个网页。

```
>>> b''.join([b'a',b'c'])
'ac'
>>> b'a' + b'b'
'ab'
```

不过这也无关紧要，因为 python2 里面我们可以理解 str 就对应的是 python3 的 bytes 类型。这一块最好 python2 和 python3 分裂得很厉害，最好不要用对接的思维了，是 python2 就用 python2 的思维，是 python3 就用 python3 的思维。

其他还有很多方法包括切片操作等就不赘述了。

bytearray 类型

bytearray 和 bytes 类型类似，而且其内部支持的方法和操作也和 bytes 类型类似，除了其更像是一个列表，可以原处修改而字符串和 bytes 是不可变的。python2 现在也有 bytearray 类型了，只是内在的文本和二进制是不分的。

文件

文件对象是可迭代对象。

写文件

对文件的操作首先需要用 **open** 函数创建一个文件对象，简单的理解就是把相应的接口搭接好。文件对象的 **write** 方法进行对某个文件的写操作，最后需要调用 **close** 方法写的内容才真的写进去了。

```
file001 = open('test.txt','w')
file001.write('hello world1\n')
file001.write('hello world2\n')
file001.close()
```

如果你们了解 C 语言的文件操作，在这里会为 python 语言的简单便捷赞叹不已。就是这样三句话：创建一个文件对象，然后调用这个文件对象的 wirte 方法写入一些内容，然后用 close 方法关闭这个文件即可。

读文件

一般的用法就是用 **open** 函数创建一个文件对象，然后用 **read** 方法调用文件的内容。最后记得用 **close** 关闭文件。

```
file001 = open('test.txt')
filetext=file001.read()
print(filetext)
file001.close()
```

此外还有 **readline** 方法是一行一行的读取某文件的内容。

open 函数的处理模式

open 函数的处理模式如下：

'r' 默认值，read，读文件。

'w' wirte，写文件，如果文件不存在会创建文件，如果文件已存在，文件原内容会清空。

'a' append，附加内容，也就是后面用 write 方法内容会附加在原文件之后。

'b' 处理模式设置的附加选项，'b' 不能单独存在，要和上面三个基本模式进行组合，比如'rb' 等，意思是二进制数据格式读。

'+' 处理模式设置的附加选项，同样'+' 不能单独存在，要和上面三个基本模式进行组合，比如'r+' 等，+ 是 updating 更新的意思，也就是既可以读也可以写，那么'r+'，'w+'，'a+' 还有什么区别呢？区别就是'r+' 不具有文件创建功能，如果文件不存在会报错，然后'r+' 不会清空文件，如果'r+' 不清空文件用 write 方法情况会有点复杂；而'w+' 具有文件创建功能，然后'w+' 的 write 方法内容都是重新开始的；而'a+' 的 write 方法内容是附加在原文件上的，然后'a+' 也有文件创建功能。

用 with 语句打开文件

类似之前的例子我们可以用 with 语句来打开文件，这样就不用 close 方法来关闭文件了。然后 with 语句来提供了类似 try 语句的功能可以自动应对打开文件时的一些异常情况。

```
with open('test.txt','w') as file01:
    file01.write('hello world1\n')
    file01.write('hello world2\n')

with open('test.txt','r') as file01:
    filetext=file01.read()
    print(filetext)
```

除字符串外其他类型的读取

文本里面存放的都是字符串类型，也就是写入文件需要用 str 函数强行将其他类型转变成字符串类型，而读取进来想要进行一些操作则需要将字符串类型转变回去。比如用 int 或者 float 等，不过列表和字典的转变则需要 **eval** 函数。

eval 这个函数严格来讲作用倒不是为了进行上面说的类型转换的，它就是一个内置函数，一个字符串类型 python 代码用 eval 函数处理了之后就能转变为可执行代码。

```
>>> eval('1+1')
2
>>> eval('[1,2,3]')
[1, 2, 3]
>>> eval("{'a':1,'b':2,'c':3}")
{'c': 3, 'b': 2, 'a': 1}
```

推荐使用 pickle 模块来处理其他类型的文件读写问题，相对来说更简单更安全。请参看 pickle 模块这一小节12。

程序中的逻辑

布尔值

boolean 类型，和大多数语言一样，就两个值：**True**，**False**。然后强制类型转换使用函数 **bool**。

其他逻辑小知识

在 python 中，有些关于逻辑真假上的小知识，需要简单了解下。

- 数 0、空对象或者其他特殊对象 None 值都认为是假
- 其他非零的数字或非空的对象都认为是真
- 前面两条用 bool 函数可以进行强制类型转换
- 比较和相等测试会递归作用在数据结构中
- 比较和相等测试会返回 True 或 False（1 和 0 的 custom version（翻译为定制版？））

比如列表都是真，但空列表是假。

None

有些函数没有 return 的值就会返回 None 值，None 值是 NoneType 对象中的一个值，和列表的空值等是不同的，它和其他任何值都不一样的。比如

re.search 如果没有找到匹配就会返回 None 值。这个时候需要知道得是 None 值在逻辑上是逻辑假，not None 是逻辑真。

```
>>> def f():
...  pass
...
>>> y = f()
>>> y
>>> type(y)
<class 'NoneType'>
```

（is 和 == 的区别对 None 的影响？）这里 y is None 和 y == None 都返回 True。没看出区别。

if 条件判断

python 中的条件语句基本格式如下:

```
if  test:
    条件判断执行区块
```

也就是 if 命令后面跟个条件判断语句，然后记住加个冒号，然后后面缩进的区块都是条件判断为真的时候要执行的语句。

```
if  test:
    do something001
else :
    do something002
```

这里的逻辑是条件判断，如果真，do something001；如果假，do something002。

```
if  test001:
    do something001
elif test002:
    do something002
```

显然你一看就明白了，elif 是 else 和 if 的结合。

逻辑与或否

and 表示逻辑与，or 表示逻辑或，not 表示逻辑否。

下面编写一个逻辑，判断一个字符串，这个字符串开头必须是 a 或者 b，结尾必须是 s，倒数第二个字符不能是单引号'。在这里就演示一下逻辑。。

code3-1

```
x='agais'
if ((x[0] == 'a' or x[0] == 'b')
    and x[-1] =='s'
    and (not x[-2] =="''")):
    print('yes it is..')
```

```
yes it is..
```

稍复杂的条件判断

现在我们了解了 if，elif 和 else 语句，然后还了解了逻辑与或非的组合判断。那么在实际编程中如何处理复杂的条件逻辑呢？

首先能够用逻辑语句“与或非”组合起来的就将其组合起来，而不要过分使用嵌套。如下面代码所示，如果一个情况分成两部分，那么就用 if...else... 语句，

```
x=-2
if x>0:
    print('x 大于 0')
else:
    print('x 小于 0')
```

而如果一个情况分成三部分，那么就用 if...elif...else 语句。同一深度的这些平行语句对应的是“或”逻辑，或者说类似其他编程语言的 switch 语句。

```
x=2
if x>0:
    print('x 大于 0')
elif x<0:
    print('x 小于 0')
else:
    print('x 等于 0')
```

我们再看一看下面的代码，这个代码是错误的，两个 if 语句彼此并不构成逻辑分析关系。①

```
x=2
if x>0:
    print('x 大于 0')
if x<0:
    print('x 小于 0')
```

① 四个甚至更多的平行或逻辑就用更多的 elif，读者请自己实验一下。

```
else:
    print('x 等于 0')
```

然后我们看到下面的代码，这个例子演示的是在加深一个深度的条件判断语句它当时处于的逻辑判断情况，这个语句的条件判断逻辑是本语句的判断逻辑再和左边（也就是前面）的判断逻辑的“与”逻辑，或者说成是“交集”。比如说 *print('0<x<2')* 这个语句就是本语句的判断逻辑 *x<2* 和上一层判断逻辑 *x>0* 的“交集”，也就是 *0<x<2*。

```
x=-2
if x>0:
    print('x 大于 0')
    if x>2:
        print('x>2')
    elif x<2:
        print('0<x<2')
    else:
        print('x=2')
elif x<0:
    print('x 小于 0')
else:
    print('x 等于 0')
```

整个过程的情况如下图所示：

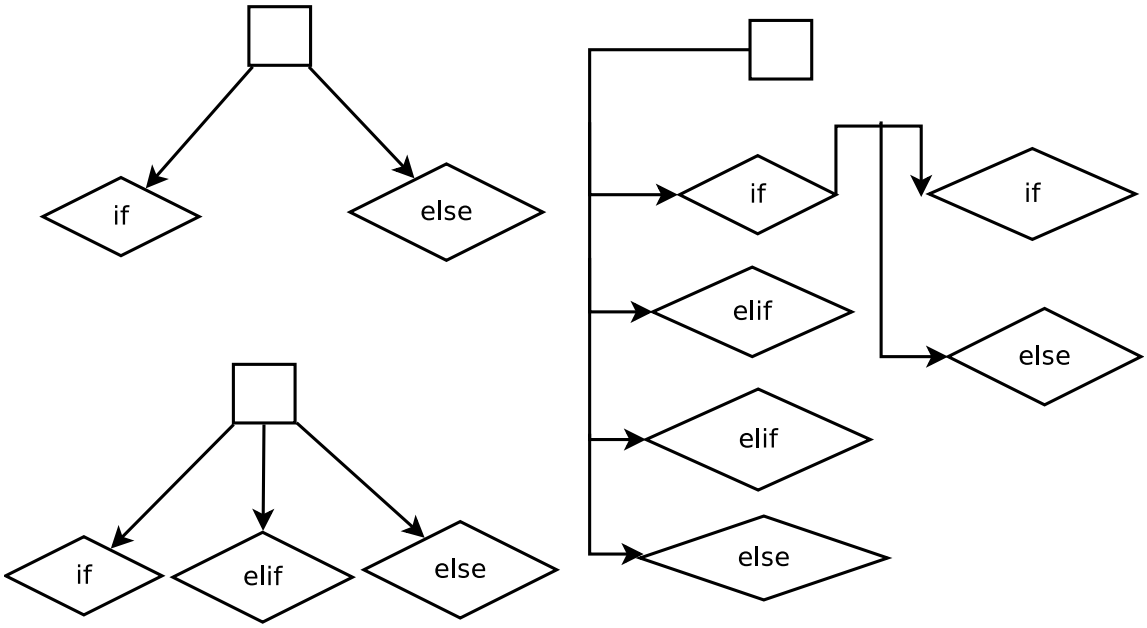


图 3-1: 复杂条件判断

为了在编程的时候对处于何种判断逻辑之下有一个清晰的认识，强烈建议读者好好思考一下。毕竟磨刀不误砍柴功。

try 语句捕捉错误

try 语句是编程中用来处理可能出现的错误或者已经出现但并不打算应付的错误最通用的方式。比如一个变量你预先想的是接受一个数值，但是用户却输入了一个字符，这个时候你就可以将这段语句包围在 try 里面；或者有时你在编程的时候就发现了这种情况，只是懒得理会他们，那么简单的把这块出错的语句包围在 try 里面，然后后面跟个 except 语句，打印出一个信息“出错了”，即可。用法如下所示：

```
while True:
    x=input(' 请输入一个数，将返回它除以 2 之后的数值\n输入"quit" 退出\n')
    if x=='quit':
        break
```

```
    try :
        num=float(x)
        print(num/2)
    except:
        print(' 出错了')
```

异常处理完整语句

```
try:
    yourCode
except yourError:
    do something
except yourError2:
    do something2
......
else:
    do somethingN
finally:
    do the funallystuff
```

这个语句的逻辑是试着执行 try 区块下的语句，如果出现异常，那么看是不是异常 yourError，如果是则执行 do something，如果是 yourError2，则执行 do something2 ..... 等等，如果没有异常，则执行 else 字句：do somethingN，如果还有异常，则这个异常将会返回（更上面的控制程序）。

那么 finally 语句的作用是什么呢，finally 语句实际上和整个语句中异常判断情况没有关系，不管有没有异常发生，最后它都将被执行。和简单地不缩进直接写在下面的语句比起来，finally 语句的特点就是就算程序发生异常了，它也会先被执行，然后将异常上传给上面的控制程序。

else 语句和 finally 语句是可选的，根据具体情况来看。

for 里面放 try 语句的情况

for 语句里面放上 try 语句还需要细讲一下。

具体 try 语句相关逻辑前面说过了，这里的问题是 for 语句的继续执行问题。首先是第一个情况，try 字句里面使用 return，这在函数里面是会跳出 for 语句的，也就是执行多次只要成功一次就会被跳出。然后错误捕捉，如果错误捕捉里面再放入一个 raise 语句，再抛出一个错误，这个时候 for 语句是会被中止的。然后抛出这个异常。然后是 else 字句，其逻辑是 try 多次没有错误，那么将会执行 else 字句，但是如果你 try 一次，然后 else 语句里面加入 break 命令，则会跳出 for 语句的。

这里面情况稍微有点复杂，目前我接触到的有如下两种应用:

这是一个 mongodb 的安全调用的函数装饰器。其在试图调用 mongodb 的时候，如果发生 AutoReconnect 异常，那么将会 sleep 一秒然后再去 try 之前的那个调用函数。如果成功了，那么进入 return，然后自然就跳出 for 语句了。

```
def safe_mongocall(call):
    '''mongodb replica set assistant'''
    def _safe_mongocall(*args, **kwargs):
        for i in xrange(100):#
            try:
                return call(*args, **kwargs)
            except pymongo.AutoReconnect:
                import time
                time.sleep(1)
                print("try to connect mongodb again...")
    return _safe_mongocall
```

第二个例子较为常用，就是在重复做某件事的时候可能会发生错误，然后捕捉这个错误，然后继续执行。然后捕捉的时候计了一下数。

```
def test():
    failcount = 0

    for i in range(src_count):
        try:
            do something
        except Exception as ex:
            failcount += 1

    sucess_count = src_count - failcount
    return sucess_count
```

其实我们还可以想到另外一种程序结构，那就是 try 和 else 在 for 语句里面构成逻辑分支。当你试着做某件事的时候，try，如果正常则执行 else 字句然后 break，如果发生某个异常则执行异常中的字句，就是 try 里面的内容不被执行。这有点反常规，但联系实际生活，我们确实也存在这样的逻辑，那就是假想如何如何，发生错误不行则执行 else 字句，就是假想 try 里面的内容不实际执行。

in 语句

in 语句对于可迭代对象都可以做出是否某个元素包含在某个对象之中的判断。

```
>>> 'a' in ['a',1,2]
True
>>> dict
{'a': 1, 'c': 2, 'b': 3, 'd': 4}
>>> 'e' in dict
```

```
False
>>> '2' in dict
False
```

从上面例子可以看到，一般的列表判断元素是否存在和我们之前预料的一致，关于字典需要说的就是 in 语句只判断键，不判断值。

for 迭代语句

一般有内部重复操作的程序可以先考虑 for 迭代结构实现，实在不行才考虑 while 循环结构，毕竟简单更美更安全。

python 的 for 迭代语句有点类似 lisp 语言的 dolist 和 dotimes 函数，具体例子如下：

code3-2

```
for x in 'abc':
    print(x)
```

```
a
b
c
```

in 后面跟的是**序列**类型，也就是字符串，列表，数组都是可以的。这个语句可以看作先执行 x='a' 或者类似的匹配赋值操作，然后执行缩进的区块，后面依次类推。（所以 for 语句也支持序列解包赋值，请参看：4.5.1）

else 分句

```
for x in 'abc':
    if x == 'b':
        print(x)
        break
else:
    print('test')
```

for 语句加上 else 分句这种形式，如果 for 迭代完了就会执行 else 分句。但如果 for 语句还在迭代过程中，break 或者 return 出来了，那么 else 分句将不会被执行。

range 函数

range 函数常和 for 迭代语句一起使用，其返回一个可迭代对象。

```
range(1,10,2)
```

range 函数的用法如上，表示从 1 开始到 10，步长为 2，如果用 list 函数将其包裹，将会输出 [1,3,5,7,9]。如果不考虑步长的话，这个 range 函数就有点类似于在序列调出多个值那一小节2.3.3谈论的区间的情况。所以 range(10) 就可以看作 [0,10)，range(1,10) 就可以看作 [1,10)。但是在这里再加上步长的概念和区间的概念又有所不同了。

code3-3

```
for x in range(-10,-20,-3):
    print(x)
```

```
-10
-13
```

```
-16
-19
```

上面例子还演示了 range 的负数概念，这里如果用区间概念来考察的话，是不能理解的，之所以行得通，是因为它的步长是负数，如果不是负数，那么情况就会和之前讨论的结果类似，将是一个空值。

迭代加上操作

迭代产生信息流并经过某些操作之后生成目标序列，更多内容请参见列表解析一节2.5.9。

```
>>> squares=[x**2 for x in [1,2,3,4,5]]
>>> squares
[1, 4, 9, 16, 25]
```

enumerate 函数

enumerate 函数返回一个 enumerate 对象，这个对象将偏移值和元素组合起来，成为一个可迭代对象了。

```
>>> enu = enumerate('abcd')
>>> [i for i in enu]
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

while 循环

while 语句用法和大多数编程语言类似，就是条件控制，循环结构。

```
while test:
    do something
else :
    do something
```

值得一提的是 else 语句和 while 语句属于一个整体，通常情况下 while 执行完了然后执行下面的语句似乎不需要加上 else 来控制[①]。不过 else 语句的一个功用就是如果 while 循环的时候遇到 break 那么 else 语句也不会执行而是直接跳过去了，见下面。

break 命令

break 跳出最近的 while 或者 for 循环结构。前面谈到了 else 和 while 语句构成一个整体的时候，break 可以跳过 else 语句。

continue 命令

continue 命令接下来的循环结构的执行区块将不执行了，跳到条件判断那里看看是不是继续循环。如果是，那么继续循环。同样在 for 语句中 continue 命令的意思也是一样的。

pass 命令

pass 命令就是什么都不做。pass 命令即可用于循环语句也可用于条件语句。

pass 命令什么都不做似乎没有什么意义，不过作为一个空占位符还是很有用的。比如你编写一个大型的 GUI 程序，信号－槽机制都构思好了，只是对应的

① 最后一下 while 是 False 所以会跳转到执行 else 的语句那里。

函数暂时还没写好，这个时候你可以将对应的函数，只是空的函数名加上 pass 语句写上，这样整个程序就可以继续边编写边调试了。

操作或者函数

函数也是一个对象，叫函数对象。函数名和变量名一样都是引用，函数名后面带个括号才真正实际执行。比如下面不带括号就只是返回了对这个函数对象的引用地址。

```
>>> print
<built-in function print>
```

要理解函数也是一个对象，比如在下面的例子中，fun 刚开始是一个函数列表，然后在 for 的迭代语句里，意思具体就是 multiply 这个函数对象，然后接下来又是 plus 这个函数对象。整个过程是对 x*a 然后再加上 b。即 $a*x+b$

```
x = 3


def multiply(x,a):
    return x*a


def plus(x,b):
    return x+b


fun = [multiply , plus]
para = [3,2]
```

```
for fun,para in zip(fun,para):
    x = fun(x,para)
print(x)
```

自定义函数

定义函数用 def 命令，语句基本结构如下：

```
def yourfunctionname(para001,para002...):
    do something001
    do something002
```

参数传递问题

函数具体参数的值是通过赋值形式[①]来传递的，这有助于理解后面的不定变量函数。而函数的参数名是没有意义的，这个可以用 lambda 函式来理解之，def 定义的为有名函数，有具体的引用地址，但内部作用原理还是跟 lambda 无名函式一样，形式参数名是 x 啊 y 啊都无所谓。为了说明这点，下面给出一个古怪的例子：

code4-1

```
y=1
def test(x,y=y):
    return x+y
print(test(4))
```

① 整个过程有点类似前面讨论的一般赋值语句，但又有点区别的。

```
5
```

输出结果是 5。我们看到似乎函数的形式参数 y 和外面的 y 不是一个东西，同时参数的传递是通过赋值形式进行的，那么具体是怎样的呢？具体的解释就是函数的形式参数 y 是这个函数自己内部的**本地变量** y，和外面的 y 不一样，更加深入的理解请看下面的变量作用域问题。

然后还有：

```
>>> x=[1,2,3]
>>> for x in x:
...  print(x)
...
1
2
3
```

我们知道 for 语句每进行一次迭代之前也进行了一次赋值操作，所以 for 语句里面刚开始定义的这个 x 和外面的 x 也不是一个东西，刚开始定义的 x 也是 for 语句内部的**本地变量**。更加深入的理解请看下面的变量作用域问题。

想到这里我又想起之前编写 removeduplicate 函数遇到的一个问题，那就是 for 语句针对列表这个可变的可迭代对象的工作原理是如何的？具体请看下面的例子：

```
>>> lst=[1,2,3,4]
>>> for x in lst:
...  print(x,lst)
...  del lst[-1]
...
```

```
1 [1, 2, 3, 4]
2 [1, 2, 3]
```

可迭代对象的惰性求值内部机制在我看来很神奇，目前还不太清楚，但从这个例子看来列表的惰性求值并没有记忆内部的数值，只是记忆了返回返回值的次数（合情合理），然后如果迭代产生了 StopIteration 异常就终止。

变量作用域问题

python 的变量作用域和大部分语言比如 c 语言或 lisp 语言的概念都类似，就是函数里面是局部变量，一层套一层，里面可以引用外面，外面不可以引用里面。

具体实现机制是每个函数都有自己的命名空间，（和模块类似）就好像一个盒子一样封装着内部的变量。所谓的本地变量和函数有关，或者其他类似的比如 for 语句；所谓的全局变量和模块有关，更确切的表述是和文件有关，比如说在现在这个文件里，你可以通过导入其他模块的变量名，但实际上模块导入之后那些变量名都引入到这个文件里面来了。

具体实现和类的继承类似也是一种搜索机制，先搜索本地作用域，然后是上一层 (def，lambda，for) 的本地作用域，然后是全局作用域，然后是内置作用域。更加的直观的说明如下图所示：

函数就是有函数作用域的情况也是盒子里面有盒子

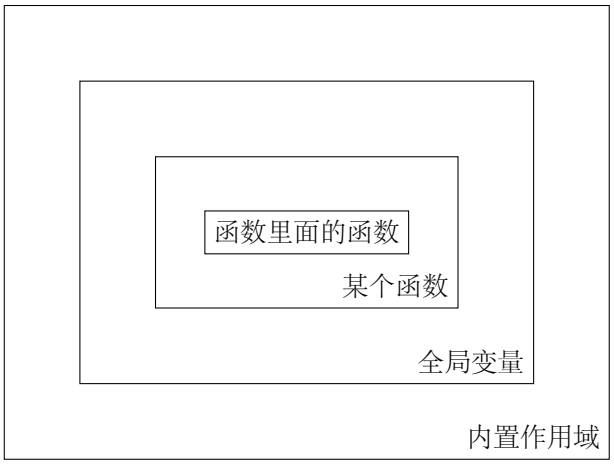


图 4-1: python 的变量作用域

简单来说 python 的变量作用域问题就是：盒子套盒子，搜索是从盒子最里面然后往外面寻找，里面可以用外面的变量，外面的不可以用里面的。

内置作用域

内置作用域就是由一个`__builtin`模块来实现的，python 的作用机制最后会自动搜索这个内置模块的变量。这个内置模块里面就是我们前面学习的那些可以直接使用的函数名，比如 print，range 等等之类的，然后还有一些内置的异常名。

所以我们想到即使对于这些 python 的内置函数我们也是可以覆盖定义的，事实确实如此：

```
>>> abs(-3)
3
>>> def abs(x):
...  print(x)
...
>>> abs(3)
```

```
3
>>> abs(-3)
-3
```

global 命令

如果希望函数里面定义的变量就是全局变量，在变量声明的时候前面加上 **global** 命令即可。

通常不建议这么做，除非你确定需要这么做，然后你需要写两行代码才能实现，意思也是不推荐你这么做。

```
def test():
    global var
    var= 'hello'
test()
print(var)
```

而且就算你这样做了，这个变量也只能在本 py 文件中被引用，其他文件用不了。推荐的做法是另外写一个专门用于配置参数的 config.py 文件，然后那些全局变量都放在里面，如果某个文件要用，就 import 进来。而对与这个 config.py 文件的修改会影响所有的 py 文件配置，这样让全局变量可见可管可控更加通用，才是正确的编程方式。

nonlocal 命令

nonlocal 命令 python3 之后才出现，这里实现的概念有点类似于 lisp 语言的闭包 (closure 技术)，就是如果你有某个需要，需要函数记忆一点自己的状态，同时又不想这个状态信息是全局变量，也不希望用类的方式来实现，那么就可以用 nonlocal 命令来简单地完成这个任务。

global 意味着命名只存在于一个嵌套的模块中，而 nonlocal 的查找只限于

嵌套的 def 中。要理解 nonlocal 首先需要理解函数里面嵌套函数的情况——也就是所谓的工厂函数，一个函数返回一个函数对象。比如说

```
def add(x):
    x=x
    def action(y):
        return x+y
    return action
```

```
>>> add1=add(1)
>>> add1(5)
6
>>> add2=add(2)
>>> add2(5)
7
```

这里的 return action 是返回一个函数对象，这样 add1 的实际接口是 def action 那里。熟悉 lisp 语言的明白，action 外面的那个函数的变量叫做自由变量，不过嵌套函数在这里可以引用自由变量[①]但不能直接修改自由变量。如果我们声明 nonlocal x，那么就可以修改嵌套函数外面声明的变量了。

```
def add(x):
    x=x
    def action(y):
        nonlocal x
        x=x+1
```

① 如果自己定义那么就是自己的本地变量了，这里的自由变量的意思是嵌套函数自己没有定义，引用母函数的变量。

```
        return x+y
    return action
```

```
>>> add2=add(2)
>>> add2(5)
8
>>> add2(5)
9
>>> add2(5)
10
```

然后我们看到这个生产出来的函数具有了运行上的状态性，实际上通过类也能构建出类似的效果，不过对于某些问题可能闭包方式处理显得更适合一些。

下面给出一个稍微合理点的例子：

```
def myrange(n):
    i=n
    def action():
        nonlocal i
        while i>0:
            i=i-1
            return i
    return action
```

```
>>> myrange5=myrange(5)
>>> myrange5()
4
>>> myrange5()
3
```

```
>>> myrange5()
2
>>> myrange5()
1
>>> myrange5()
0
>>> myrange5()
>>>
```

下面给出类似的类的实现方法：

```
class myrange:
    def __init__(self,n):
        self.i=n
    def action(self):
        while self.i > 0:
            self.i -= 1
            return self.i
```

```
>>> myrange5=myrange(5)
>>>
>>> myrange5.action()
4
>>> myrange5.action()
3
>>> myrange5.action()
2
>>> myrange5.action()
1
>>> myrange5.action()
```

```
0
>>> myrange5.action()
>>>
```

我们看到从编码思路上基本上没什么差异，可以说稍作修改就可以换成类的实现版本。推荐一般使用类的实现方法。但有的时候可能用类来实现有点不伦不类和大材小用了。这里就不做进一步讨论了，闭包思想是函数编程中很重要的一个思想，学习了解一下也好。

参数和默认参数

定义的函数圆括号那里就是接受的参数，如果参数后面跟个等号，来个赋值语句，那个这个赋的值就是这个参数的默认值。比如下面随便写个演示程序：

code4-2

```
def test(x='hello'):
    print(x)
test()
test('world')
```

```
hello
world
```

不定参量函数

我们在前面谈到 sum 函数2.2.8只接受一个列表，而不支持这样的形式：sum(1,2,3,4,5)。现在我们设计这样一个可以接受不定任意数目参量的函数。首先让我们看看一种奇怪的赋值方式。

序列解包赋值

```
>>> a,b,*c=1,2,3,4,5,6,7,8,9
>>> print(a,b,c,sep=' | ')
1 | 2 | [3, 4, 5, 6, 7, 8, 9]
>>> a,*b,c=1,2,3,4,5,6,7,8,9
>>> print(a,b,c,sep=' | ')
1 | [2, 3, 4, 5, 6, 7, 8] | 9
>>> *a,b,c=1,2,3,4,5,6,7,8,9
>>> print(a,b,c,sep=' | ')
[1, 2, 3, 4, 5, 6, 7] | 8 | 9
```

带上一个星号 * 的变量变得有点类似通配符的味道了，针对后面的序列[①]（数组，列表，字符串），它都会将遇到的元素收集在一个列表里面，然后说是它的。

for 语句也支持序列解包赋值，也是将通配到的的元素收集到了一个列表里面，如：

code4-3

```
for (a,*b,c) in [(1,2,3,4,5,6),(1,2,3,4,5),(1,2,3,4)]:
    print(b)
```

```
[2, 3, 4, 5]
[2, 3, 4]
[2, 3]
```

① 似乎序列赋值内置迭代操作

函数中的通配符

```
>>> def test(*args):
...  print(args)
...
>>> test(1,2,3,'a')
(1, 2, 3, 'a')
```

我们看到类似上面序列解包赋值中的带星号表通配的概念，在定义函数的时候写上一个带星号的参量（我们可以想象在函数传递参数的时候有一个类似的序列解包赋值过程），在函数定义里面，这个 args 就是接受到的参量组成的元组。

mysum 函数

code4-4

```
def mysum(*args):
    return sum(args[:])

print(mysum(1,2,3,4,5,6))
```

```
21
```

这样我们定义的可以接受任意参数的 mysum 函数，如上所示。具体过程就是将接受到的 args（已成一个元组了），然后用 sum 函数处理了一下即可。

任意数目的可选参数

在函数定义的写上带上两个星号的变量 **args，那么 args 在函数里面的意思就是接受到的可选参数组成的一个字典值。

```
>>> def test(**args):
...  return args
...
>>> test(a=1,b=2)
{'b': 2, 'a': 1}
```

我们看到利用这个可以构建出一个简单的词典对象生成器。

解包可迭代对象传递参数

之前 *args 是在函数定义中，然后通配一些参数放入元组中。这里是在函数调用中，针对可迭代对象，可以用一个 * 星号将其所包含的元素迭代出来，然后和参数一一对应赋值。

```
>>> map = map(lambda x:x+2,[1,2,3])
>>> print(*map)
3 4 5
>>> print(*[1,2,3])
1 2 3
```

最简单的打印文件命令

前面说到文件也是一个可迭代对象，然后如果在这里解包文件对象将是一个最简单的打印文件命令，简单得惊天地泣鬼神了...

```
print(*open('test.py'))
```

解包字典成为关键字参数

和上面的类似，通过 **args 语法可以将某个字典对象解包成为某个函数的关键字参数。还是以上面那个函数 f 为例子:

```
>>> def f(a,b,c=3):
...  print(a,b,c)
>>> f(**{'c':6,'b':4,'a':2})
2 4 6
>>> f(1,2,5)
1 2 5
```

这个例子也告诉我们不是可选参数的 a 和 b 同样也可以通过这种字典形式复制。

参数的顺序

老实说一般参数，可选参数（关键字参数），任意（通配）参数，任意（通配）关键字参数所有这些概念混在一起非常的让人困惑。就一般的顺序是:

1. 一般参数，这个如果有一定要在第一位，然后通过位置一一对应分配参数。
2. 关键字参数，关键字参数跟在一般参数后面，通过匹配变量名来分配。
3. 通配一般参数，其他额外的非关键字的参数分配到 *args 元组里面。
4. 通配关键字参数，其他额外的关键字参数分配到 **args 字典里面，这个必须在最后面。

一般的情况就是这些吧，可能你会遇到更加困难的情况，到时候再说吧。

递归函式

虽然递归函式能够在某种程度上取代前面的一些循环或者迭代程序结构，不过不推荐这么做。这里谈及递归函式是把某些问题归结为数学函数问题，而这些问题常常用递归算法更加直观（不一定高效）。比如下面的菲波那奇函数：

code4-5

```
def fib(n):
    if n==0:
        return 1
    if n==1:
        return 1
    else:
        return fib(n-1)+fib(n-2)

for x in range(5):
    print(fib(x))
```

```
1
1
2
3
5
```

我们可以看到，对于这样专门的数学问题来说，用这样的递归算法来表述是非常简洁易懂的。至于其内部细节，我们可以将上面定义的 fib 称之为函式，函式是一种操作的模式，然后具体操作就是复制出这个函式（函数或者操作都是数据），然后按照这个函式来扩展生成具体的函数或者操作。

下面看通过递归函式来写阶乘函数，非常的简洁，我以为这就是最好最美的方法了。

code4-6

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n*fact(n-1)

print(fact(0),fact(10))
```

```
1 3628800
```

什么时候用递归?

最推荐使用递归的情况是这样的情况，那就是一份工作（或函数）执行一遍之后你能够感觉到虽然所有的工作没有做完，但是已经做了一小部分了，有了一定的进展了，就好比是蚂蚁吞大象一样，那么这个时候你就可以使用递归思想了。其次有的时候有那么一种情况虽然表面上看似乎并没有什么进展，但事情在发展，你能感受到有一个条件最终将会终止程序从而得到一个输出，那么这个时候就可以用递归。

递归思想最核心的两个概念就是一做了一小部分工作，你能感觉到做着做着事情就会做完了；二有一个终止判断最终将会起作用。

其实通过递归函式也可以实现类似 for 的迭代结构，不过我觉得递归函式还是不应该滥用。比如下面通过递归函式生成一种执行某个操作 n 次的结构：

这种情况不推荐使用递归

code4-7

```
def dosomething(n):
    if n==0:
```

```
        pass
    elif n==1:
        print('do!')
    else:
        print('do!')
        return dosomething(n-1)

print(dosomething(5))
```

```
do!
do!
do!
do!
do!
None
```

可以看到，如果把上面的 print 语句换成其他的某个操作，比如机器人向前走一步，那么这里 dosomething 换个名字向前走 (5) 就成了向前走 5 步了。

lisp 的 car-cdr 递归技术

在 lisp 语言中，car-cdr 递归技术是很重要的一门技术，它的特长就是遍历随意嵌套的列表结构可以同一对列表中的每一个元素执行某种操作。

首先我们来看下面的例子，一个把任意嵌套列表所有元素放入一个列表中的函数：

code4-8

```
lst = [[1,2,[3]],[4,[5,[[[[10],11]]]],(1,2,3)],[{'a','b','c'},8,9]]
```

```
def is_list(thing):
    return isinstance(thing, list)

def flatten(iter):
    templst = []
    for x in iter:
        if not is_list(x):
            templst.append(x)
        else:
            templst += flatten(x)
    return templst

print(flatten(lst))
```

```
[1, 2, 3, 4, 5, 10, 11, (1, 2, 3), 'b', 'c', 'a', 8, 9]
```

这个函数的逻辑是如果是最小元素对象不是列表，那么收集进列表；如果不是，那么把它展开，这里就是调用的原函数继续展开函式。

上面的例子严格意义上来讲还不算 lisp 的经典 car-cdr 递归技术，下面给出一个典型的例子，就是复制任意嵌套结构的列表。当然列表的 copy 方法就可以做这个工作，这里主要通过这个例子来进一步深入 car-cdr 技术。

```
def is_list(thing):
    return isinstance(thing, list)

def copy_list(lst):
    if  not  is_list(lst):
        return lst
    elif lst == []:
```

```
        return []
    else:
        return [copy_list(lst[0])] + copy_list(lst[1:])

print(copy_list([1,[2,6],3]))
```

这种嵌套列表的复制以及后面的修改等等操作，最合适的就是 lisp 的 car-cdr 技术了，但我不得不承认，这种递归写法是递归函式里面最难懂的了。

不管怎么严格，在这个基础之上，因为第一个 if not 的语句中传递下来的 lst 实际上已经是非列表的其他元素了，然后我们可以进行一些其他修改操作，这样在保持原列表的复杂嵌套的基础上，等于遍历的对列表中的所有元素进行了某种操作。

比如所有元素都平方：

```
def square(x):
    return x**2

def square_list(lst):
    if  not  is_list(lst):
        return square(lst)
    elif lst == []:
        return []
    else:
        return [square_list(lst[0])] + square_list(lst[1:])

print(square_list([1,[2,6],3]))
```

我们可以想像更加复杂功能的函数作用于列表中所有的元素同时又不失去原列表复杂的嵌套结构，lisp 的 car-cdr 这种技术了解一下吧，但是不是一定要使用复杂的嵌套结构呢？也许没有必要吧。。

lambda 函式

lambda λ 表达式这个在刚开始介绍 lisp 语言的时候已有所说明，简单来说就是函数只是一个映射规则，变量名，函数名都无所谓的。这里就是没有名字的函数的意思。

具体的样子如下面所示：

code4-9

```
f=lambda x,y,z:x+y+z
print(f(1,2,3))
```

```
6
```

lanmbda 函式在有些情况下要用到，比如 pyqt 里面的信号－槽机制用 connect 方法的时候，槽比如是函数名或者无参函数，如果用户想加入参量的话，可以使用 lamba 函式引入。

读者如果对 lambda 函式表达不太熟悉强烈建议先简单学一学 **scheme** 语言。

print 函数

print 函数因为很常用和基础，就放在这里了。

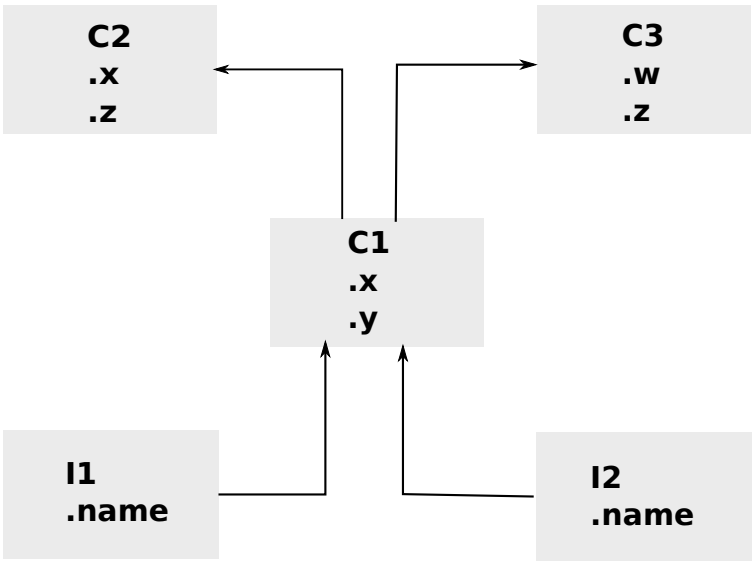
print 函数接受任意的参量，逐个打印出来。然后它还有一些关键字参数，**sep**：默认值是’ ’，也就是一个空格，如果修改为空字符串，那么逐个打印出来的字符之间就没有间隔了。**end**：默认值是’\n’，**file** 默认值是 sys.stdout，也就是在终端显示，你可以修改为某个文件变量，这样直接往某个文件里面输出内容。

类

在 python 中一切皆对象。前面学的那些操作对象都是 python 程序语言自己内部定义的对象（Object），而接下来介绍的类的语法除了更好的理解之前的那些对象之外，再就是可以创造自己的操作对象。一般面向对象 (OOP) 编程的基本概念这里不重复说明了，如有不明请读者自己随便搜索一篇网页阅读下即可。

python 中类的结构

python 中的类就好像树叶，所有的类就构成了一棵树，而 python 中超类，子类，实例的重载或继承关系等就是由一种搜索机制实现的：



python 首先搜索 self 有没有这个属性或者方法，如果没有，就向上搜索。

比如说实例 l1 没有，就向上搜索 C1，C1 没有就向上搜索 C2 或 C3 等。

实例继承了创造他的类的属性，创造他的类上面可能还有更上层的超类，类似的概念还有子类，表示这个类在树形层次中比较低。

well，简单来说类的结构和搜索机制就是这样的，很好地模拟了真实世界知识的树形层次结构。

上面那副图实际编写的代码如下：

```
class C2: ...
class C3: ...
class C1(C2,C3): ...
l1=C1()
l2=C1()
```

其中 class 语句是创造类，而 C1 继承自 C2 和 C3，这是多重继承，从左到右是内部的搜索顺序（会影响重载）。l1 和 l2 是根据类 C1 创造的两个实例。

对于初次接触类这个概念的读者并不指望他们马上就弄懂类这个概念，这个概念倒并一定要涉及很多哲学的纯思考的东西，也可以看作一种编程经验或技术的总结。多接触也许对类的学习更重要，而不是纯哲学抽象概念的讨论，毕竟类这个东西创造出来就是为了更好地描述现实世界的。

最后别人编写的很多模块就是一堆类，你就是要根据这些类来根据自己的情况情况编写自己的子类，为了更好地利用前人的成果，或者你的成果更好地让别人快速使用和上手，那么你需要好好掌握类这个工具。

类的最基础知识

类的创建

```
class MyClass:
    something
```

类的创建语法如上所示，然后你需要想一个好一点的类名。类名规范的写法是首字母大写，这样好和其他变量有所区分。

根据类创建实例

按照如下语句格式就根据 MyClass 类创建了一个实例 myclass001。

```
myclass001=MyClass()
```

类的属性

```
>>> class MyClass:
...  name='myclass'
...
>>> myclass001=MyClass()
>>> myclass001.name
'myclass'
>>> MyClass.name
'myclass'
>>> myclass001.name='myclass001'
>>> myclass001.name
'myclass001'
>>> MyClass.name
'myclass'
```

如上代码所示，我们首先创建了一个类，这个类加上了一个 name 属性，然后创建了一个实例 myclass001，然后这个实例和这个类都有了 name 属性。然后我们通过实例加上点加上 name 的这种格式引用了这个实例的 name 属性，并将其值做了修改。

这个例子简单演示了类的创建，属性添加，实例创建，多态等核心概念。后面类的继承等概念都和这些大同小异了。

类的方法

类的方法就是类似上面类的属性一样加上 def 语句来定义一个函数，只是函数在类里面我们一般称之为方法。这里演示一个例子，读者看一下就明白了。

```
>>> class MyClass:
...  name='myclass'
...  def double(self):
...   self.name=self.name*2
...   print(self.name)
...
>>> myclass001=MyClass()
>>> myclass001.name
'myclass'
>>> myclass001.double()
myclassmyclass
>>> myclass001.name
'myclassmyclass'
```

这里需要说明的是在类的定义结构里面，self 代表着类自身，self.name 代表着对自身 name 属性的引用。然后实例在调用自身的这个方法时用的是 myclass001.double() 这样的结构，这里 double 函数实际上接受的第一个参

数就是自身，也就是 myclass001，而不是无参数函数。所以类里面的方法（被外部引用的话）至少有一个参数 self。

类的继承

实例虽然说是根据类创建出来的，但实际上实例和类也是一种继承关系，实例继承自类，而类和类的继承关系也与之类似，只是语法稍有不同。下面我们来看这个例子：

code5-1

```
class Hero():
    def addlevel(self):
        self.level=self.level+1
        self.hp=self.hp+self.addhp

class Garen(Hero):
    level=1
    hp=455
    addhp=96

garen001=Garen()
for i in range(6):
    print('级别:',garen001.level,'生命值: ' ,garen001.hp)
    garen001.addlevel()
```

```
级别: 1 生命值: 455
级别: 2 生命值: 551
级别: 3 生命值: 647
级别: 4 生命值: 743
```

级别: 5 生命值：839

级别: 6 生命值：935

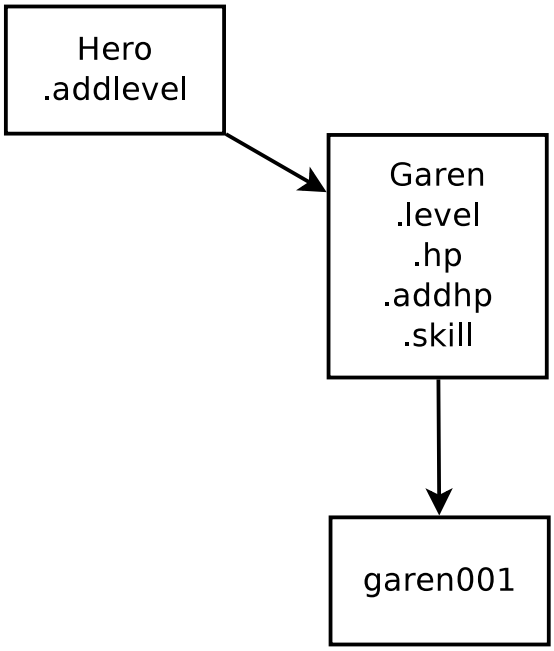


图 5-1: 类的继承示例

这里就简单的两个类，盖伦 Garen 类是继承自 Hero 类的，实例 garen001 是继承自 Garen 类的，这样 garen001 也有了 addlevel 方法，就是将自己的 level 属性加一，同时 hp 生命值也加上一定的值，整个过程还是很直观的。

类的内置方法

如果构建一个类，就使用 pass 语句，什么都不做，python 还是会为这个类自动创建一些属性或者方法。

```
>>> class TestClass:
...  pass
...
```

```
>>> dir(TestClass)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__',
  '__gt__',  '__hash__', '__init__', '__le__', '__lt__',
   '__module__', '__ne__', '__new__', '__reduce__',
   '__reduce_ex__', '__repr__', '__setattr__',
   '__sizeof__', '__str__', '__subclasshook__',
    '__weakref__']
```

这些变量名字前后都加上双下划线是给 python 这个语言的设计者用的，一般应用程序开发者还是不要这么做。

这些内置方法用户同样也是可以重定义他们从来覆盖掉原来的定义，其中特别值得一讲的就是`__init__`方法或者称之为构造函数。

__init__ 方法

`__init__`方法对应的就是该类创建实例的时候的构造函数。比如：

```
>>> class Point:
...  def __init__(self,x,y):
...   self.x=x
...   self.y=y
...
>>> point001=Point(5,4)
>>> point001.x
5
>>> point001.y
4
```

这个例子重载了__init__ 函数，然后让他接受三个参数，self 等下要创建的实例，x，还有 y 通过下面的语句给这个待创建的实例的属性 x 和 y 赋了值。

self 意味着什么

self 在类中是一个很重要的概念，当类的结构层次较简单时还容易看出来，当类的层次结构很复杂之后，你可能会弄糊涂。self 就是指现在引用的这个实例。比如你现在通过调用某个实例的某个方法，这个方法可能是一个远在天边的某个类给出的定义，就算如此，那个定义里面的 self 还是指调用这个方法的那个实例，这一点要牢记于心。

比如下面这个例子：

```
class Test():
    x = 5
    def __init__(self):
        self.x = 10


test = Test()
```

```
>>> test.x
10
>>> Test.x
5
```

其中 self.x 就是对应的创建的实例的属性 x，而前面定义的 x 则是类 Test 的属性 x。

类的操作第二版

现在我们可以写出和之前那个版本相比更加专业的类的使用版本了。

code5-2

```
class Hero():
    def addlevel(self):
        self.level=self.level+1
        self.hp=self.hp+self.addhp

class Garen(Hero):
    def __init__(self):
        self.level=1
        self.hp=455
        self.addhp=96
        self.skill=['不屈','致命打击','勇气','审判','德玛西亚正义']

garen001=Garen()
for i in range(6):
    print('级别:',garen001.level,'生命值: ' ,garen001.hp)
    garen001.addlevel()
print('盖伦的技能有: ',"".join([x + '  ' for x in garen001.skill]))
```

```
级别: 1 生命值: 455
级别: 2 生命值: 551
级别: 3 生命值: 647
级别: 4 生命值: 743
级别: 5 生命值: 839
级别: 6 生命值: 935
盖伦的技能有: 不屈  致命打击  勇气  审判  德玛西亚正义
```

似乎专业的做法类里面多放点方法，最好不要放属性，不太清楚是什么。但

确实这样写给人感觉更干净点，方法是方法，如果没有调用代码就放在那里我们不用管它，后面用了构造函数我们就去查看相关类的构造方法，这样很省精力。

类的操作第三版

code5-3

```
class Unit():
    def __init__(self,hp,atk,color):
        self.hp=hp
        self.atk=atk
        self.color=color
    def __str__(self):
        return '生命值：{0}，攻击力：{1}，颜色：\
        {2}'.format(self.hp,self.atk,self.color)

class Hero(Unit):
    def __init__(self,level,hp,atk,color):
        Unit.__init__(self,hp,atk,color)
        self.level=level
    def __str__(self):
        return '级别：{0}，生命值：{1}，攻击力：{2}，\
        颜色：{3}'.format(self.level,self.hp,self.atk,self.color)

    def addlevel(self):
        self.level=self.level+1
        self.hp=self.hp+self.addhp
        self.atk=self.atk+self.addatk

class Garen(Hero):
    def __init__(self,color='blue'):
```

```
        Hero.__init__(self,1,455,56,color)
        self.name='盖伦'
        self.addhp=96
        self.addatk=3.5
        self.skill=['不屈','致命打击','勇气','审判','德玛西亚正义']

if __name__ == '__main__':
    garen001=Garen('red')
    garen002=Garen()
    print(garen001)
    unit001=Unit(1000,1000,'gray')
    print(unit001)
    for i in range(6):
        print(garen001)
        garen001.addlevel()
    print('盖伦的技能有：','".join([x + '  ' for x in garen001.skill]))
```

级别：1，生命值：455，攻击力：56，颜色：red

生命值：1000，攻击力：1000，颜色：gray

级别：1，生命值：455，攻击力：56，颜色：red

级别：2，生命值：551，攻击力：59.5，颜色：red

级别：3，生命值：647，攻击力：63.0，颜色：red

级别：4，生命值：743，攻击力：66.5，颜色：red

级别：5，生命值：839，攻击力：70.0，颜色：red

级别：6，生命值：935，攻击力：73.5，颜色：red

盖伦的技能有：不屈 致命打击 勇气 审判 德玛西亚正义

现在就这个例子相对于第二版所作的改动，也就是核心知识点说明之。其中函数参量列表中这样表述`color='blue'` 表示 blue 是 color 变量的备选值，也就是 color 成了可选参量了。

构造函数的继承和重载

上面例子很核心的一个概念就是`__init__` 构造函数的继承和重载。比如我们看到 garen001 实例的创建，其中就引用了 Hero 的构造函数，特别强调的是只有创造实例的时候比如这样的形式 Garen() 才叫做调用了 Garen 类的构造方法，比如这里
`Hero.__init__(self,1,455,56,color)` 就是调用了 Hero 类的构造函数，这个时候需要把 self 写上，因为 self 就是最终创建的实例 garen001，而不是 Hero，而且调用 Hero 类的构造函数就必须按照它的参量列表形式来。这个概念需要弄清楚!

理解了这一点，在类的继承关系中的构造函数的继承和重载就好看了。比如这里 Hero 类的构造函数又是继承自 Unit 类的构造函数，Hero 类额外有一个参量 level 接下来也要开辟存储空间配置好。

`__str__` 函数的继承和重载

第二个修改是这里重定义了一些类的`__str__` 函数，通过重新定义它可以改变默认 print 某个类对象是的输出。默认只是一段什么什么类并无具体内容信息。具体就是 return 一段你想要的字符串样式即可。

类的高级知识

更多类的高级知识的讨论参见“类的高级知识”这一章7。

模块

现在让我们进入模块基础知识的学习吧，建立编写自己的模块，这样不断积累自己的知识，不断变得更强。

实际上之前我们已经接触过很多 python 自身的标准模块或者其他作者写的第三方模块，而 import 和 from 语句就是加载模块用的。这里我们主要讨论如何自己编写自己的模块。

from 语句和 import 语句内部作用机制很类似，只是在变量名的处理方式上有点差异（from 会把变量名复制过来）。这里重点就 import 的工作方式说明如下：

1. 首先需要找到模块文件。
2. 然后将模块文件编译成位码（需要时，根据文件的时间戳。），你会看到新多出来一个`__pycache__` 文件夹。
3. 执行编译出来的位码，创建该 py 文件定义的对象。

这三个步骤是第一次 import 的时候会执行的，第二次 import 的时候会跳过去，而直接引用内存中已加载的对象。

找到模块文件

python 模块的搜索路径会搜索几个地方，这些地方最后都会放在 sys.path 这个列表里面，所以在你的 py 文件刚开始修改这个 sys.path，append 上你想

要的地址也是可以的。我在这里选择了这种简单的方法，除此之外还有很多方法这里先不涉及。

比如主文件一般如下：

```
import os,sys
sys.path.append(os.environ['HOME']+'/pymf')
from pyconfig import *
```

这里为什么使用`from pyconfig import *`这样的语句而不是 import 语句呢？因为我决定整个项目的主 py 文件除了这一个 from 语句之外不会再 import 或者 from 其他模块了，其他所有模块的引用都放在 pyconfig.py 这个主配置文件里面。

pyconfig.py 任务就是加载最常用最通用的一些模块，如果你实际编写的另外一个项目通用 pyconfig 文件满足不了你的要求了，那么你可以把那个 pyconfig 文件复制过来，然后放在你的项目文件夹里面，然后继续衍化修改。这个经验是我从 LaTeX 文档的编写中总结出来的，既满足了共性又满足了个性。

那么为什么要用 from 语句，很简单。如果用 import 语句，那么 pyconfig.py 文件里面 import math 模块，在主 py 文件里面引用就要使用这样的格式`pyconfig.math.pi`，这既不方便而且违背大家平时惯用的那种 math.pi 格式。

现在我们让可以开始编写自己的模块吧。

编写模块

well，编写模块就是一些 py 文件，然后模块的名字和 py 文件里面的内容编写好就是了。

我现在编写了一个 pyconfig.py 文件，放在主文件夹（ubuntu 系统）的 pymf 文件里面的。里面定义了一个斐波拉契函数，如下所示：

```
# 菲波那奇数列
def fib(n):
    if n==0:
        return 1
    if n==1:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

然后我们的测试小脚本如下：

```
import os,sys
sys.path.append(os.environ['HOME']+'/pymf')
from pyconfig import *

print([fib(n) for n in range(10)])
```

import 语句

import 语句的一般使用方法之前已有接触，比如 import math，然后要使用 math 模块里面的函数或者类等需要使用这样的带点的变量名结构：math.pi。

此外 import 语句还有一个常见的缩写名使用技巧，比如 import numpy as np，那么后面就可以这样写了，np.array，而不是 numpy.array。

from 语句

from 语句的使用有以下两种情况：

```
from this import this
from what import *
```

第一种形式是点名只导入某个变量，第二种形式是都导入进来。我想读者肯定知道这点，使用第二种导入形式的时候要小心变量名覆盖问题，这个自己心里有数即可。

reload 函数

reload 函数可以重新载入某个模块，reload 函数的优点就是不需要重新启动应用程序，更加合理的动态重载一些模块。reload 只能用于 python 编写的模块，在 python3 中，reload 函数被移到 imp 模块里面去了，因此首先需要 import imp 才能使用了。比如说：

```
from imp import reload
reload(somemodule)
```

python3 进阶

类的高级知识

类内部的字典

__dict__ 值

类和模块都支持对其内部属性或者方法的点的引用方式，这种引用是根据类和模块的__dict__ 这个字典值来的，然后属性继承就是向上搜索链接的字典。python 中的类和实例实际就是带有链接的字典，其内的方法和属性都是字典值，其都可以通过__dict__ 的字典索引语法来引用。如这样的语法：X.__dict__['data']，

类的字典比如下面这个：

```
class A():
    def __init__(self,a):
        self.a=a

    def fun2(self,what):
        print('fun',what)

class B(A):
    def __init__(self):
        self.d=5
```

```
    b=2
    def fun3(self):
        print('fun3')
```

其__dict__ 值如下所示：

```
>>> B.__dict__
mappingproxy({'__doc__': None, '__init__': <function B.__init__ at
0xb7096adc>, 'fun3': <function B.fun3 at 0xb7096a4c>,
'__module__': '__main__', 'b': 2})
```

我们看到除了 class 定义的继承关系之外，类 B 内部定义的所有属性都存储在__dict__ 字典值里面了。我们可以猜测类的继承关系 python 是通过另外一种机制管理的，而且有一个通用类，这个通用类上有很多方法就是后面要谈到的那些内置的方法。

点运算索引是加上继承搜索机制了的，这里的__dict__ 的值只是本实例或者本类存储的属性。

__dict__ 这个字典值对于描述类或实例的存在状态很重要，下面有三个 python 的内置的类的方法，你可以通过重新定义它们来获得你想要的效果。

类按键取值

__getitem__ 方法支持类或实例以 Class['x'] 这样的格式引用其内字典某个（或者某一些）索引的值。

类按键赋值

__setitem__ 方法支持类或实例以 Class['x']=3 这样的格式修改其内字典的某个（或者某一些）索引的值。

字典的索引删除

__delitem__ 方法支持类或实例以 del Class['x'] 这样的格式删除字典的某个（或者某一些）索引。

关于上面三个内置方法都以下面这个例子来讲解了：

```
class GClass():
    def __getitem__(self,key):
        return self.__dict__[key]
    def __setitem__(self,key,value):
        self.__dict__[key] = value
    def __delitem__(self,key):
        del self.__dict__[key]

    def __repr__(self):
        return str(self.__dict__)
```

```
>>> test = GClass()
>>> print(test)
{}
>>> test['a'] = 1
>>> print(test)
{'a': 1}
>>> test['a'] = 2
>>> print(test)
{'a': 2}
>>> del test['a']
>>> print(test)
{}
```

实际上上面三个内置方法不一定是对类内部字典的操作，也可以是类其中其他数据的操作，而且经过修改它们也支持如同列表的切片操作。

属性管理

`__getattr__(self,name)` 返回 self.name 的值。

属性赋值

`__setattr__(self,name,value)` 和 self.name = value 动作关联。

属性删除

`__delattr__(self,name)` del self.name

属性获取

`__getattribute__(self,name)` 不推荐使用。

数学运算

一般加法

X + other , `__add__(self,other)`

右侧加法

所谓加法是 X+other，如果是右侧加法，则为 radd，然后公式是：other+X。一般不区分左右的就用上面的一般加法。

other + X，`__radd__(self,other)`

增强加法

X +=other，`__iadd__(self.other)`

一般减法

X - other，`__sub__(self,other)`

同上面情况一样类似的还有 rsub 和 isub。

其他数学运算符一览

然后其他数学运算符下面简要列表之：

* 乘法，`__mul__(self,other)`，下面的类似的都有右侧运算和增强运算，不再赘述了。

// 整除，`__floordiv__`，下面类似的参数都是 self 和 other，不再赘述了。

/ 除法，`__div__`

% 取余，`__mod__`

** 开方，`__pow__`

« 左移运算，`__lshift__`

» 右移运算，__rshift__

& 位与，__and__

| 位或，__or__

^ 位异或[1]，__xor__

类似的右侧运算名字前面加上 r，增强运算名字前面加上 i，不赘述了。

逻辑运算

bool 函数支持

bool(X) __bool__(self)

类之间的相等判断

参考网站。

这里先总结下 is 语句和 == 判断和 isinstance 和 id 还有 type 函数，然后再提及 python 类的内置方法__eq__。

python 是一个彻头彻尾的面向对象的语言，python 内部一切数据都是对象，对象就有类型 type 的区别。比如内置的那样对象类型：

```
>>> type('abc')
<class 'str'>
>>> type(123)
<class 'int'>
```

[1] 异或的逻辑是相同取 0，不同取 1。

```
>>> type([1,2,3])
<class 'list'>
```

对象除了有 type 类型之外，还有 id 属性，id 就是这个对象具体在内存中的存储位置。

当我们说 lst=[1,2,3] 的时候，程序具体在内存中创建的对象是 [1,2,3]，而 lst 这个变量名不过是一个引用。然后我们看下面的例子：

```
>>> x=[1,2,3]
>>> y=[1,2,3]
>>> type(x)
<class 'list'>
>>> type(y)
<class 'list'>
>>> id(x)
3069975884
>>> id(y)
3062209708
>>> x==y
True
>>> x is y
False
```

type 函数返回对象的类型，id 函数返回对象具体在内存中的存储位置，而 == 判断只是确保值相等，is 语句返回 True 则更加严格，需要对象在内存上（即 id 相等）完全是同一个东西。

对象之间的类型比较可以用如下语句来进行比较：

```
>>> x=10
>>> type(x) == int
True
>>> type(x) == type(0)
True
```

不过不是特别好用，比如假设 fun 是你自己定义的一个函数，用 type(fun) == function 就会出错，然后 type 比较还要小心 NoneType 和其他空列表类型不同，而且 type 比较并没有将类的继承考虑进去。

一般推荐 isinstance 函数来进行类型比较，请参考这个网站的说明。推荐使用 types 模块的特定名字来判断类型，具体如下：

types.NoneType None 这个值的类型

types.TypeType type 对象。

types.BooleanType 还可以使用 **bool**。

types.IntType 还可以使用 **int**，类似的有 **long**，**float**。

types.ComplexType 复数类型

types.StringType 字符串类型，还可以使用 **str**。

types.TupleType 元组，还可以使用 **tuple**，类似的有 **list**，**dict**。

types.FunctionType 定义的函数类型，此外还有 **types.LambdaType**。

值得一提的是 print 等内置函数不是 FunctionType 而是 BuiltinFunctionType。

```
>>> import types
>>> isinstance(print,types.FunctionType)
False
```

```
>>> isinstance(print,types.BuiltinFunctionType)
True
```

更多内容请参见types 模块的官方文档。

__eq__ 方法

__eq__ 方法定义了两个对象之间 A == B 的行为。比如下面：

```
def __eq__(self,other):
    if self.__dict__.keys() == other.__dict__.keys():
        for key in self.__dict__.keys():
            if  not self.__dict__.get(key)==other.__dict__.get(key):
                return False
        return True
    else:
        return False
```

定义了这样的__eq__ 方法之后，我们运行 == 语句，如果两个对象之间内置字典键和值都是一样的，那么就返回 True。

```
>>> test=GClass()
>>> test.a=1
>>> test2=GClass()
>>> test2.a=1
>>> test == test2
True
>>> test is test2
False
```

如果我们不重定义__eq__ 方法，似乎 test 和 test2 会从原始的 object 类继

承__eq__ 方法，然后它们比较返回的是 False，我想可能是这两个实例内部某些值的差异吧，但应该不是基于 id。

比较判断操作

类似上面的 == 比较操作，还有如下比较判断操作和对应的内置方法可以重定义。

- X != Y，行为由__ne__(self,other) 定义。
- X >= Y，行为由__ge__(self,other) 定义。
- X <= Y，行为由__le__(self,other) 定义。
- X > Y，行为由__gt__(self,other) 定义。
- X < Y，行为由__lt__(self,other) 定义。

in 语句

如下所示：

```
    def __in__(self,other):
        for key in self.__dict__.keys():
            if not self.__dict__.get(key) == other.__dict__.get(key):
                return False
        return True
```

提供了 what in X 语句的支持，上面的例子是基于类其内字典的内容而做出的判断。

强制类型变换

所包含的内置方法有:

`__int__(self)`	返回整型
`__long__(self)`	长整型
`__float__(self)`	浮点型
`__complex__(self)`	复数型
`__str__(self)`	字符型
`__oct__(self)`	八进制
`__hex__(self)`	十六进制
`__index__(self)`	切片操作

len 函数

由`__len__(self)`提供支持。

copy 方法和 deepcopy 方法

X.copy() 由`___copy__(self)`提供。

X.deepcopy() 由`___deepcopy__(self)`提供。

with 语句支持

在打开文件那里谈及的 with open(...) as f 的这类语句是由以下两个内置方法提供的: `__enter__(self)` 和`__exit__(self,...)`, exit 的还有其他一些参数这里忽略了, enter 的返回值会赋值给 with 中的 as 后面的变量。

函数调用

请看下面的例子:

```
class Position():
    def __init__(self,x=0,y=0):
        self.x = x
        self.y = y
    def __call__(self,x,y):
        self.x = x
        self.y = y
    def __repr__(self):
        return '('+str(self.x)+ ',' + str(self.y)+')'
```

```
>>> p1=Position()
>>> print(p1)
(0,0)
>>> p1(4,5)
>>> print(p1)
(4,5)
>>>
```

`__call__(self,args)` 方法支持类或者实例以 X(args) 或者 instance(args) 这样的形式调用这个函数。

和迭代操作有关

__next__ 方法

比如文件对象本身就是可迭代的，调用__next__ 方法就返回文件中下一行的内容，到达文件尾也就是迭代越界了返回：**StopIteration** 异常。

next 函数

next 函数比如 next(f) 等价于f.__next__()，其中 f 是一个文件对象。

```
>>> for line in open('removeduplicate.py'):
...  print(line,end='')
...
#!/usr/bin/env python3
#-*-coding:utf-8-*-
# 此处一些内容省略。

>>> f=open('removeduplicate.py')
>>> next(f)
'#!/usr/bin/env python3\n'
```

所以你可以通过定义类的__next__ 方法来获得这个类对于 next 函数时的反应。

iter 函数

文件对象，map 对象，zip 对象，filter 对象，生成器对象自身已经带有了__next__ 方法，所以可以直接用 next 函数。不过虽然序列（列表，元组，

字典，ranges 对象[①]）等是可迭代对象，但是没有__next__ 方法。

iter 函数可以让序列们拥有__next__ 方法，这样的它们就可以使用 next 函数了。

```
>>> list=[1,2,3]
>>> next(list)
Traceback (most recent call last):
  File ”<stdin>”, line 1, in <module>
TypeError: ’list’ object is not an iterator
>>> i=iter(list)
>>> next(i)
1
>>> i.__next__()
2
```

下面是 for 语句的 while 实现版本：

```
>>> list=[1,2,3]
>>> iter=iter(list)
>>> while True:
...     try:
...         x=next(iter)
...     except StopIteration:
...         break
...     print(x)
...
1
```

① range 函数的返回，也属于序列类型。

```
2
3
```

range 对象也可以通过 iter 函数来生成一个可迭代对象。

重构字典的 iter 函数

我们可以通过重新定义字典类的__iter__ 函数来获得一个新类，这个类用 iter 函数处理之后的迭代器返回的是经过排序的字典的键。

```
class SortedDict(dict):
    def __init__(self,dict={}):
        super().__init__(dict)

    def __iter__(self):
        self._keys = sorted(self.keys())
        for i in self._keys:
            yield i

dict02 = SortedDict()
dict02['a'] = 1
dict02['b'] = 1
dict02['c'] = 1
dict02['d'] = 1
x = iter(dict02)
```

这个__iter__ 函数就是所谓的生成器函数，需要返回一个生成器对象。然后经过 iter 函数处理之后就能调用 next 函数来逐渐获得经过排序之后的值了。

当对象内存存储回收时的操作

当对象内存存储被回收时，python 最后将执行一个内置方法__del__，这个一般不推荐使用。

静态方法

```
class Test:
#    @staticmethod
    def hello():
        print('aaa')

test=Test()
test.hello()
```

在上面的例子中，我们希望创造一个函数，这个函数和 self 实例没有关系（这里指这个函数将不接受 self 这个默认参数了）。如上所示，hello 函数只是希望简单打印一小段字符，如上面这样的代码是错误的，如果我们在这个函数上面加上**@staticmethod**，那么上面这段代码就不会报错了，

```
class Test:
    @staticmethod
    def hello():
        print('aaa')

test=Test()
test.hello()
```

这样在类里面定义出来的函数叫做这个类的静态方法，静态方法同样可以继承等等，而静态方法通常使用最大的特色就是不需要建立实例，即可以直接从类来调用，如下所示：

```
class Test:
    @staticmethod
    def hello():
        print('aaa')

Test.hello()
```

静态方法的使用比如 pyqt 中的

```
QtGui.QFileDialog.getOpenFileName(......)
```

就是一个静态方法，可以通过直接调用这个方法来弹出询问打开文件的窗口，并不需要先实例化一个对象，然后通过 self.what 等类似的形式来调用。

装饰器

python 语言的装饰器概念算是比较高级的概念了，不过并不是那种冷门的用的很少的概念，比如在前面的静态方法中就使用装饰器的概念：

```
    @staticmethod
    def what():
        pass
```

装饰器的作用机制就是对接下来的函数进行进一步的封装，比如上面的例子就是：

```
    def what():
        pass
    what = staticmethod(what)
```

可见装饰器并不是一个什么神秘的难懂的概念，同样你可以定义自己的函数，这个函数处理某个函数对象，并对其进行某种封装。

类似的装饰器还有类的方法装饰器`@classmethod`，在 pyqt 中有槽的装饰器`@pyqtSlot()` 和`@pyqtSlot(int, str)` 等，第一个例子接下来你定义的槽只接受 self 这个参数，第二个例子接下来你定义的槽除了接受 self 参数外，还接受一个 int 类型参数和一个 str 类型参数。

自定义装饰器

```
def print1(f):
    print('1',f)
    return f

@print1
def print3(c):
    print(c)

print3('c')#image print1(print3)('c')
```

比如上面的 print1 函数就做成了一个装饰器函数，后面的 print3 函数可以理解为 `print3=print1(print3)` 。

多个装饰器

```
def print1(f):
    print('1',f)
    return f

def print2(f):
    print('2',f)
    return f

@print2
@print1
def print4(c):
    print(c)

print4('c')#image print2(print1(print4))(c)
```

类似上面的多个装饰器就可以简单理解为: print4 = print2(print1(print4))

装饰器带上参数

在前面的例子中，我们就可以简单将装饰器函数理解为一个接受函数对象返回返回函数对象的函数，这很直观和简单，要有限考虑这样的模型。实际上装饰器也是可以带上自己的参数的，这需要通过什么函数的闭包结构才能完成，如下面的例子所示:

```
def print1(f):
    print('1',f)
    return f

def print2(b):
    def test(f):
```

```
        print('2',f,b)
        return f
    return test

@print2('b')
@print1
def print4(c):
    print(c)

print4('c')#image print2(print1(print4))('b')(c)
```

所谓闭包结构简单来说就是函数里面套函数的结构。前面在介绍 nolocal 关键词的时候说道，如果函数里面的嵌套函数的某个变量加上声明关键词 nolocal，那么（如果嵌套函数内没有定义该本地变量），则该变量名是对应嵌套函数外面的自由变量的（自由变量在函数生存期具有记忆能力）。

上面例子可以理解为`print2(print1(print4))('b')(c)` 这样一个过程。首先执行 print1(print4)，然后返回 print4，这很直观。然后表达式变为`print2(print4)('b')(c)`，这里紧跟着的参数应该由 python 的装饰器解释程序做的，具体我还有点困惑……，总之这里装饰器函数的额外的参数是由 print2 这个最外层的函数接收的，然后具体返回的 test 函数对象还是如同前面一样只接受目标函数对象。前面这样的`print2(print1(print4))('b')(c)` 描述可能并没很好地反映 python 的内部处理机制了，也许更好的描述是:

装饰器的参数是通过函数的闭包机制以自由变量的形式引入进去的。

类方法

还有一个装饰器有时也会用到，`@classmethod` ，叫什么类方法装饰器。其和前面的静态方法一样也可以不新建实例，而直接通过类来调用。其和静态方法的区别就是静态方法在调用的时候没有任何默认的第一参数，而类方法在调用的

时候默认第一参数就是调用的那个类[①]。

```
class Test:
    @classmethod
    def hello(cls):
        print('from class:', cls, 'saying hello')

Test.hello()
```

```
from class: <class '__main__.Test'> saying hello
```

多重继承的顺序问题

我们来看下面这个例子:

```
class B1():x='B1'
class B2():x='B2'
class B3():x='B3'
class B(B1,B2,B3):x='B'
class A1():x='A1'
class A2():x='A2'
class A(A1,A2):x='A'
class D(B,A):x='D'
test=D()
print(test.x)
```

① 参考了这个网站。

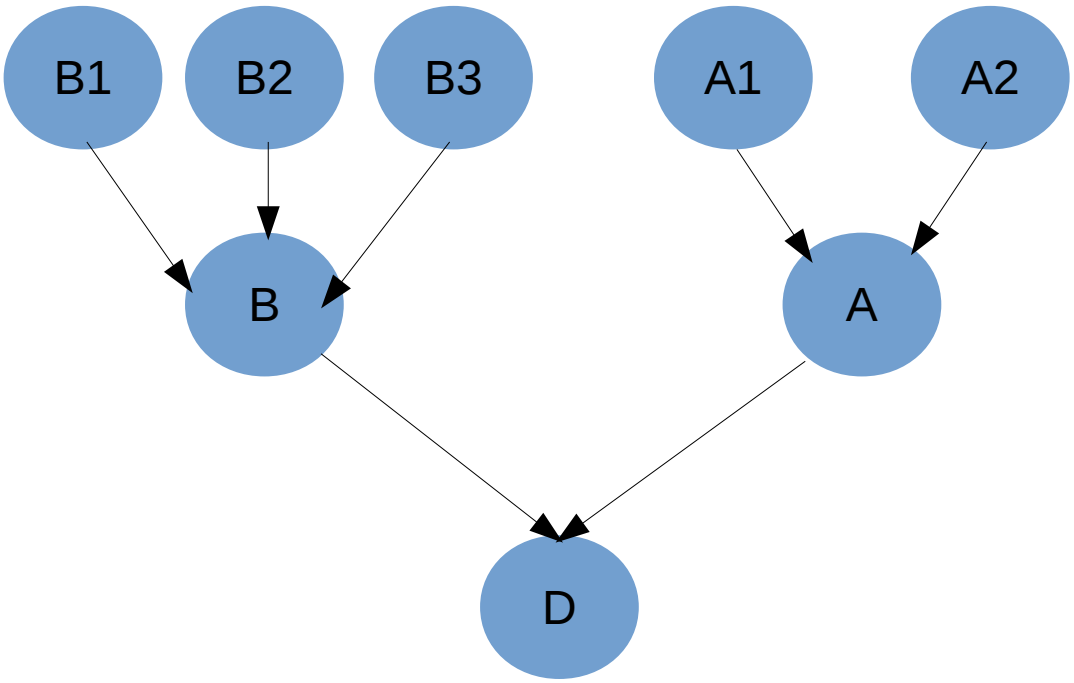


图 7-1：多重继承

你可以测试一下上面这个例子，首先当然结果是 D 自己的 x 被先查找，然后返回'D'，如果你把类 D 的 x 定义语句换成 pass，结果就是'B'。这说明这里程序的逻辑是如果 test 实例找不到 x，那么再找 D，D 找不到再接下来找 D 继承自的父类，首先是 B，到目前为止，没什么新鲜事发生。

然后我们再把 B 的 x 赋值语句换成 pass，这时的结果是'B1'，也没什么好惊讶的。然后类似的一致操作下去，我们会发现 python 的值的查找顺序在这里是：D，B，B1，B2，B3，A，A1，A2。

于是我们可以总结道：恩，类的多重继承就是深度优先法则，先把子类或者子类的子类都查找完，确认没有值之后再继续从左到右的查找。

一般情况来说这么理解是没有问题的，但是在编程界多重继承中有个有名的问题——菱形难题。

菱形难题

参考资料：维基百科菱形难题

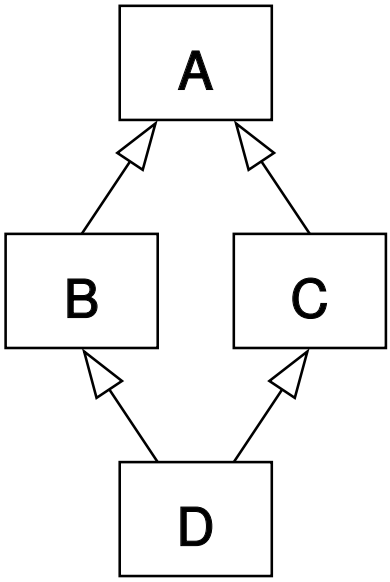


图 7-2: 菱形难题

菱形难题即在如上的类的继承中，如果 C 和 A 都有同名属性 x，那么 D 会调用谁的呢？读者测试下面的例子：

```
class E():x='E'
class F():x='F'
class G():x='G'
class A(F,G):x='A'
class B(E,F):x='B'
class D(B,A):pass
test=D()
print(test.x)
```

此时运行结果到 DBE 都没有什么出奇的，接下来要某是 DBEF[①]，要某是

① E 当然也检查过了，否则 E 有没有值是无法确认的。

DBEA，这里程序的结果是'A'。这里的情况确实比较纠结，如果没有这个 F 作为菱形难题的交叉点，似前面的层次分明，那么简单的理解为深度优先即可，这里 python3 的选择是'A'，不清楚为什么要这么选择。

我们再来看这个例子：

```
class E():x='E'
class F():x='F'
class G():x='G'
class A(F,G):x='A'
class B(F,E):pass
class D(B,A):pass
test=D()
print(test.x)
```

此时结果是'A'，连 E 都被跳过去了，变成了彻底的横向优先原则。

程序出现菱形难题之后，情况变得不可琢磨了。上面的三个情况

```
D(B(B1 B2 B3) A(A1 A2)) → D B B1 B2 B3 A A1 A2

D(B(E F) A(F G)) → D B E A F G

D(B(F E) A(F G)) → D B A F E G
```

就是这样的，总之这是很冷门的领域了。。简单的理解就是深度搜索，类似 flatten 函数处理过，然后如果遇到某个子元在下一个平行级别的子元中也含有，那么本子元会被略过，做个记号，分叉跳过去跑到 A 那里，执行完那个子元之后，又会重新调到之前的操作点上。python 怎么弄这么古怪的逻辑。。

super()

super 是 python3 新加入的特性，按照官方文档，有两种用法：

第一种是如果是单继承的类的系统，super() 这种形式就直接表示父类的意思。然后用 super(). 什么什么的来引用父类的某个变量或方法，值得一提的是原父类的 self 参量会默认加进去了，详细请看下面的调试例子。

第二种是多重继承的，搜索顺序和多重继承的搜索顺序相同，也就是从左到右。请注意调试下面的例子，如果调用 c.d 就会返回错误，说明调用的是类 A 的构造函数。

```
class A():
    def __init__(self,a):
        self.a=a

    def fun(self):
        print('fun')

    def fun2(self,what):
        print('fun',what)

class B():
    def __init__(self):
        self.d=5
    b=2
    def fun3(self):
        print('fun3')

class C(A,B):
    def __init__(self):
```

```
        super().__init__(3)
        super().fun()
        super().fun2('what')
        super().fun3()
        print(super().b)

c=C()
print(c.a,c.b)
```

```
fun
fun what
fun3
2
3 2
```

其中 A 类定义的 fun 函数在写的函数上通常有个 self 参数，而super()这种调用形式在意义上表示其的父类，同时默认第一个参数就是 self。为了理解你可以和 self 做个比较，比如 self.fun() 就是调用的实例的 fun 函数，默认的第一个参数是 self。使用 super() 在类的编写中引用本类的父类的属性和方法是很便捷的，自带支持类的多重继承功能。比如上面的例子中 fun3 能被调用是因为多重继承的机制在这里，所以它会逐个找父类。然后 c.d 会出错，因为这里初始化是用的 A 类的构造函数。

给某个对象动态加载一个方法

这里主要参考了这个网页 。

具体原理还是很简单的，那就是构建一个函数对象，然后将这个对象赋值给某个对象。但这里的函数对象如果要接受 self 参数的话，其作为类的方法还是需要一些特殊的处理的。

```
class Test():
    pass

test = Test()

def hello(self):
    print(”hello”)

import types
test.hello = types.MethodType(hello,Test)

test.hello()
```

上面的 types.MethodType 是用来构建一个类的方法的，其第一个参数是具体的函数对象，第二个参数是对应的类或实例。

然后上面的例子继续优化就是如下的形式:

```
import types

class Test():
    @classmethod
    def removeVariable(cls,name):
        return delattr(cls,name)

    @classmethod
    def addMethod(cls,func):
        return setattr(cls,func.__name__,types.MethodType(func,cls))

def hello(self):
```

```
    print(”hello”)

test = Test()

Test.addMethod(hello)

test.hello()
```

你看到了这里的 addMethod 是作用于本类的，当然你也可以选择作用于本实例:

```
import types

class Test():
    @classmethod
    def removeVariable(cls,name):
        return delattr(cls,name)

    @classmethod
    def addMethod(cls,func):
        return setattr(cls,func.__name__,types.MethodType(func,cls))

    def addMethod2(self,func):
        return setattr(self,func.__name__,types.MethodType(func,self))

def hello(self):
    print(”hello”)

test = Test()
```

```
test.addMethod2(hello)

test.hello()
```

这样这个函数就只加在本实例上面了，这用处不太大。

深入理解 python3 的迭代

在 python 中一般复杂的代码运算效率就会低一点，如果完成类似的工作但你可以更简单的语句那么运算效率就会高一点。当然这只是 python 的一个设计理念，并不尽然，但确实很有意思。

程序结构中最有用的就是多个操作的重复，其中有迭代和递归还有一般的循环语句。递归函式感觉对于某些特殊的问题很有用，然后一般基于数据结构的不是特别复杂的操作重复用迭代语句即可，最后才考虑一般循环语句。

迭代语句中 for 语句运算效率最低，然后是 map 函数（不尽然），然后是列表解析。所以我们在处理问题的时候最 pythonic 的风格，运算效率最高的就是列表解析了，如果一个问题能够用列表解析解决那么就用列表解析解决，因为 python 的设计者的很多优化工作都是针对迭代操作进行的，然后 python3 进一步深化了迭代思想，最后 python 中的迭代是用 c 语言来实现的（你懂的）。

可是让我们反思一下为什么列表解析在问题处理的时候如此通用？比如说 range 函数或者文件对象或者列表字符串等等，他们都可以称之为可迭代对象。可迭代对象有内置方法`__next__` 这个我们之前有所谈及，可迭代对象最大的特色就是有一系列的元素，然后这一系列的元素可以通过上面的内置方法逐个调出来，而列表解析就是对这些调出来的元素进行了某个表达式操作，然后将其收集起来。这是什么？我们看下面这张图片：

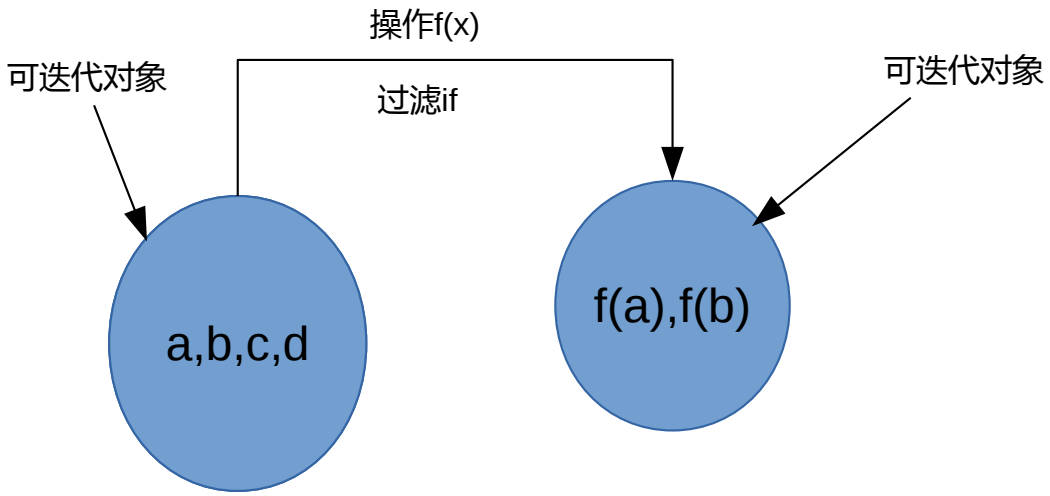


图 8-1：列表解析

这张图片告诉我们列表解析和数学上所谓的集合还有函数的定义非常的类似，可迭代对象就好像是一个集合（有顺序或者没顺序都行），然后这些集合中的所有元素经过了某个操作，这个操作似乎就是我们数学中定义的函数，然后加上过滤条件，某些元素不参加运算，这样就生成了第二个可迭代对象（一般是列表也可以是字典什么的。）

有一个哲学上的假定，那就是我们的世界一切问题都可以用数学来描述，而一些数学问题都可以用函数即如上的信息操作过滤流来描述之。当然这不尽然，但我们可以看到列表解析在一般问题处理上是很通用的思想。

不过我们看到有限的元素的集合问题适合用迭代，但无限元素的集合问题也许用递归或者循环更适合一些。然后我们又想到集合的描述分为列举描述（有限个元素的列举）和定义描述。比如说 1<x<10，x 属于整数，这就定义了一个集合。那么我们就想到 python 存在这样的通过描述而不是列举（如列表一样）的集合吗？range 函数似乎就是为了这样的目的而生的，比如说 range(10) 就定义了 [0,10) 这一系列的整数集合，range 函数生成一个 range 对象，range 对象是一个可迭代对象，我们可以把它看作可迭代对象中的描述集合类型吧。这时我们就问了，既然 0<=x<10 这样的整数集合可以通过描述来实现，那么更加复杂的函数描述可不可以实现呢？我们可不可以建立更加复杂的类似 range 对象的描述性可迭代对象呢？

生成器函数

一般函数的定义使用 return 语句，如果使用 yield 语句，我们可以构建出一个生成器函数，

```
>>> def test(x):
...     for i in range(x):
...         yield 2*i+ 1
...
>>> test(3)
<generator object test at 0xb704348c>
>>> [x for x in test(3)]
[1, 3, 5]
>>> [x for x in test(5)]
[1, 3, 5, 7, 9]
```

这个 test 函数叫做什么生成器函数，返回的是什么 generator object，生成器对象？anyway，通过 yield 这样的形式定义出来的生成器函数返回了一个生成器对象和 range 对象类似，都是描述性可迭代对象，里面的元素并不立即展开，而是请求一次运算一次，所以这种编程风格对内存压力很小，主要适合那些迭代元素特别多的时候的情况吧。

上面的 test 函数我们就可以简单理解为 2x+1，其中 0<=x<n（赋的值）。

下面给出一个问题作为练习：描述素数的生成器函数。这是网上流行的素数检验函数，效率还是比较高的了。

```
def isprime(n):
    if n ==2:
        return True
    # 按位与 1，前面一定都是 0 个位数如果是 1 则
    # 是奇数则返回 1 则真则假，如果是偶数则返回
    #0 则假则真
    elif n<2 or not n & 1:
        return False
    # 埃拉托斯特尼筛法...
# 查一个正整数 N 是否为素数，最简单的方法就是试除法，
# 将该数 N 用小于等于 N**0.5 的所有素数去试除，
# 若均无法整除，则 N 为素数
    for x in range(3,int(n**0.5)+1,2):
        if n % x == 0:
            return False
    return True
```

然后我们给出两种形式的素数生成器函数，其中 prime2 的意思是范围到（to）那里。而 prime(n) 的意思是到第几个素数。我们知道生成器函数是一种惰性求值运算，然后 yield 每迭代一次函数运算一次（即产生一次 yield），但这种机制还是让我觉得好神奇。

```
def prime2(n):
    for x in range(n):
        if isprime(x):
            yield x


def prime(n):
    i=0
    x=1
```

```
    while i<n:
        if isprime(x):
            i +=1
            yield x
        x +=1
```

在加载这些函数之后我们可以做一些检验：

```
>>> isprime(479)
True
>>> [x for x in prime2(100)]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, ........]
>>> [x for x in prime2(1000) if 100< x < 200]
[101, 103, 107, 109, 113, 127, 131, 137, 139, 149, .......]
>>> len([x for x in prime2(10000) if -1 < x < 3572])
500
>>> [x for x in prime(1)]
[2]
>>> [x for x in prime(2)]
[2, 3]
```

map 和 filter 函数

按照之前的迭代模式的描述，虽然使用常见的列表解析格式 (for 语句) 就可以完成对某个集合中各个元素的操作或者过滤，不过 python 中还有另外两个函数来实现类似的功能，map 对应对集合中各个元素进行某个函数操作（可以接受 lambda 函式），而 filter 则实现如上所述的过滤功能。然后值得一提的是 python3 之后 map 函数和 filter 函数返回都是一个可迭代对象而不是列表，和 range 函数等其他可迭代对象一样可用于列表解析结构。

map 函数

这里列出一些例子，具体编程还是先考虑列表解析模式，可能会在某些情况下需要用到 map 函数?

```
>>> map(abs, [-2,-1,0,1,2])
<map object at 0xb707dccc>
>>> [x for x in map(abs, [-2,-1,0,1,2])]
[2, 1, 0, 1, 2]
>>> [x for x in map(lambda x : x+2, [-2,-1,0,1,2])]
[0, 1, 2, 3, 4]
```

map 函数还可以接受两个可迭代对象的协作参数模式，这个学过 lisp 语言的会觉得很眼熟，不过这里按照我们的理解也是很便捷的。具体就是第一个可迭代对象取出一个元素作为 map 的函数的第一个参数，然后第二个可迭代对象取出第二个参数，然后经过函数运算，得到一个结果，这个结果如果不列表解析的话就是一个 map 对象（可迭代对象），然后展开以此类推。值得一提的是两个可迭代对象的深度由最短的那个决定，请看下面的例子:

```
>>> [x for x in map(lambda x,y : x+y, [-2,-1,0,1,2],[-2,-1,0,1,2])]
[-4, -2, 0, 2, 4]
>>> [x for x in map(lambda x,y : x+y, [-2,-1,0,1,2],[-2,-1,0,1])]
[-4, -2, 0, 2]
```

filter 函数

同样和上面的谈及的类似，filter 函数过滤一个可迭代对象然后产生一个可迭代对象。类似的功能可以用列表解析的后的 if 语句来实现。前面谈到 map 函数的时候提及一般还是优先使用列表解析模式，但 filter 函数这里有点不同，因

为列表解析后面跟个 if 可能有时会让人困惑，这时推荐还是用 filter 函数来进行可迭代对象的过滤操作。

filter 函数的基本逻辑是只有 return True（用 lambda 表达式就是这个表达式的值为真，具体请参看 python 的逻辑小知识和布尔值的一些规则3.1）的时候元素才被收集起来，或者说是过滤出来。这里强调 True 是因为如果你的函数没有 return 值那么默认的是 return None，这个时候元素也是不会过滤出来的。

请参看下面的例子来理解：

这里位运算与就是控制个位数是 *1* 那么就是奇数，这种方式更加的节省计算。

```
>>> [x for x in filter(lambda x:x&1,[1,2,3,5,9,10,155,-20,-25])]
[1, 3, 5, 9, 155, -25]
>>> [x for x in filter(lambda x:not x&1,[1,2,3,5,9,10,155,-20,-25])]
[2, 10, -20]
```

当然你也可以传统的编写函数：

```
>>> def even(n):
...     if n % 2 ==0:
...         return True

>>> [x for x in filter(even,[1,2,3,5,9,10,155,-25])]
[2, 10]
```

zip 函数

这里就顺便把 zip 函数也一起提了，zip 函数同样返回一个可迭代对象，它接受任意数目的可迭代对象，然后逐个取出可迭代对象元素构成一个元组成为自己的一个元素（待迭代出来）。和 map 函数类似迭代深度由最短的那个可迭代对象决定。

```
>>> zip(['a','b','c'],[1,2,3,4])
<zip object at 0xb7055e6c>
>>> [x for x in zip(['a','b','c'],[1,2,3,4])]
[('a', 1), ('b', 2), ('c', 3)]
>>> list(zip(['a','b','c'],[1,2,3,4]))
[('a', 1), ('b', 2), ('c', 3)]
>>> dict(zip(['a','b','c'],[1,2,3,4]))
{'c': 3, 'b': 2, 'a': 1}
```

字典到列表

这个例子似乎使用价值不大，只是说明 zip 函数接受任意数目参数的情况。y.items() 解包之后是 4 个参数传递给 zip 函数，而 zip 函数的封装逻辑就是如果有人问我，我就把你们这些迭代对象每个取出一个元素，然后用元组包装之后返回。

```
x1 = ['a','b','c','e']
x2 = [1,2,3,4]
y = dict(zip(x1,x2))
print(' 列表到字典: ',y)
new_x1,new_x2 = zip(*y.items())
print(new_x1,new_x2)
```

```
列表到字典:  {'b': 2, 'c': 3, 'a': 1, 'e': 4}
('b', 'c', 'a', 'e') (2, 3, 1, 4)
```

这个例子如果到更加复杂的情况，我们可以跳过字典形式，来个数据映射对：

```
>>> x1 = ['a','b','c','e']
>>> x2 = ['red','yellow','red','blue']
>>> x3 = [1,2,3,4]
>>> list(zip(x1,x2,x3))
[('a', 'red', 1), ('b', 'yellow', 2), ('c', 'red', 3), ('e', 'blue', 4)]
>>> new_x1,new_x2,new_x3 = zip(*list(zip(x1,x2,x3)))
>>> new_x1
('a', 'b', 'c', 'e')
>>> new_x2
('red', 'yellow', 'red', 'blue')
>>> new_x3
(1, 2, 3, 4)
```

当然对于多属性数据问题一般还是推荐使用类来处理，不过某些情况下可能不需要使用类，就这样简单处理之。

值得一提的是这种数据存储形式和 sql 存储是一致的，而且不知道你们注意到没有，这似乎实现了矩阵的转置功能。

模块包

———————————————— 随着 python 学习的深入，对这章和初学的时候见解很不相同了，获得了很多有价值的经验。下面内容较陈旧了，有时间还需要大量的修改一下。

多个模块 py 文件组成一个多文件夹目录的整体就是一个模块包。

模块包这部分知识是我们理解前人编写的各个有用的模块包的基础，同时以后我们自己要编写大型的项目也是一个人编写一个模块，一个模块对应一个任务或者一个功能的形式展开的，然后多个模块合并成一个大型的模块包。以前我们都是编写的不超过一百行的小 python 代码，不过就是对于大型的项目也不意味着我们要找一个大型的编辑器，然后一写就是上万行。在模块包的合理布局下，我们完全还可以轻松的一次编写也就那么一两百行的小代码，最后各个模块组合起来，就是一个宏大的系统了。

__init__ 文件

well，模块包和简单的模块在管理上多出来的唯一的一个内容就是你需要在每个文件夹里面加一个`__init__.py` 文件，文件的内容就是空白都没关系，但必须要有。

现在我们新建一个文件夹，`mymodule`。然后进入 mymodule 文件夹，新建一个空白文件`__init__.py`，然后新建一个文件`mymod.py`，然后里面定义了一个简单的 myfun 函数，没什么意义，就是打印了一段信息。然后我们在当前目录下进入 python3 的 eval 模式就可以开始测试你的这个新模块包了，因为

sys.path 是默认自带搜索当前工作目录的。

你的模块包必须要有一个__init__.py 文件，这个文件实际上就是暗指的你的这个模块，比如说这里是 mymodule。不管你是 import 还是使用 from 语句，只要调用 mymodule，你的__init__.py 都将执行一次。

你的模块包如果还有其他 py 文件[①]，那么这个文件在你的__init__.py 文件没有任何配置的情况下只能通过：

```
import mymodule.mymod
```

这样的语句来引入进来。

在一般情况下，__init__.py 文件是空白的或者里面填上一些类和函数来表示 mymodule 自身的配置，这满足了一般的需求了。然后模块包里面的 py 文件在使用时都用 mymodule.mymod 来引用进来。

__init__.py 文件如果是空白我们不用关心它到底起了什么作用，但如果它不是空白，里面有一些代码，比如定义了一些函数和类，因为每个模块包下的__init__.py 在 python 首次导入时都会执行一次这个文件内部的代码。所以当你 import mymodule 之后，mymodule 里面的函数和类就可以通过 mymodule.what 来使用了。

如果里面有 import 语句

如果 import 的是外部的模块方面本文件某些函数或类的调用这自不必说。这里讨论的是这种情况，有的时候你的模块包的某些子模块，你不希望 import mymodule 之后还需要 import mymodule.mymod 才能使用 mymod.py 文件里面的内容，你希望马上就能够使用，那么你需要在__init__.py 文件里面使用 import 语句了。

比如在这里__init__.py 文件加上一句话：

① 这个 py 文件就是子模块的概念。

```
import mymodule.mymod
```

这样在简单的 import mymodule 之后，你就可以直接使用 mymodule.mymod.myfun 来引用 mymod.py 文件里面的 myfun 函数了，而之前还需要额外的执行 import mymodule.mymod 命令一次。

如果里面有 from 语句

前面谈到 from 语句的前面和 import 语句是没有区别的，除了额外的引入变量名操作。在这里就是你厌倦了 mymodule.mymod.myfun 这类常常的引用的方式，你就想简单点就是 mymodule.myfun，那么你可以在__init__.py 文件中加上如下语句：

```
from mymodule.mymod import myfun
```

这样就把 myfun 这个变量名引入到 mymodule（也就是__init__.py 文件）里面去了。此时 mymodule 点的下面引用就多了一些函数和类的定义了，在你简单 import mymodule 之后。

读者应该猜到了，如果现在你在外面 eval 模式下使用 from mymodule import *，那么就可以直接调用 myfun 命令了。

__all__ 变量

__init__.py 文件还有一个高级功能，这个高级功能多少取代了前面谈论的一切，是一种新的管理模式。

如果外围 eval 调用使用的 import mymodule 语句，那么 __all__ 变量将不会被读取。只有在外围 eval 调用使用 from mymodule import * 这样的语句情况下，__all__ 变量才会被读取。同时要特别提醒的是，如果__init__.py 文件里面没有 __all__ 变量，那么情况就是我们上面讨论的，如果有 __all__ 变

量，并且使用了 from mymodule import * 语句，那么__init__.py 文件里面的其他 import 和 from 语句都将失效（其他语句还是有效的）。

表面上看 __all__ 变量的引入似乎使得事情变得混乱不堪了，不过我们外围可以使用这样的语法：

```
import mymodule
from mymodule import *
```

这样 from* 语句就负责 __all__ 变量部分，import mymodule 则负责其他 import 和 from 语句的管理，两者并无冲突。此时 mymodule，mymodule.mymod 和 mymod2 都是可以引用的。

但是在这里最大的问题是当你使用 from mymodule import * 之后，mymod2 被提到和 mymodule 同等级的状态，这多少不够美观，我们宁愿用 mymodule.myfun 或者 mymodule.mymod2 这样的形式，单独使用 mymod2 多少有点主次不分了。总的说来不推荐使用 __all__ 变量，当然也不推荐使用 from mymodule import * 这样的语法。

例外：在保证 mymodule 这个主入口不被侵犯的情况下，某些子模块的子模块（这里我们在 mymodule 文件夹里面新建一个 mymod3 文件夹，然后新建文件__init__.py，这个__init__.py 里面通过__all__ 代入了 mymod4，然后新建 mymod4.py，里面定义一个简单地打印函数。）的情况可以使用。

如上面说明的，在我们对项目模块包内部文档管理更加美观的要求下，现在在 import mymodule 之后，模块的模块的模块..... 都可以通过 __all__ 变量来优化，从而外围可以直接通过 mymodule.mymod4 这样的二级引用格式来引用，如果我们不使用这种技术，按照常规手段，那么我们需要 mymodule.mymod3.mymod4 这样的格式来引用，这太过于复杂了。

模块中的帮助信息

well，大家都知道，我们编写的模块主要是给别人看得，给别人用的，所以多写点帮助信息吧，这个没人嫌你写得多的。其他 # 下的注释就不用说了，这里主要讲一下其他的帮助信息。

还是跟着上面的例子来：

```
"""mymod.py
这是一个测试模块"""

def myfun():
    """myfun 函数
    用于打印测试"""

    print('myfun is found')
```

```
"""mymodule
我在 mymodule 文件夹的 __init__.py 文件里面"""

print('mymodule already import')

from mymodule import mymod
__all__ = ['mymod']
```

然后是测试代码：

```
import os,sys
sys.path.append(os.environ['HOME']+'/pymf')
from pyconfig import *
```

```
import  mymodule
```

具体测试查看情况如下所示，其中 help 命令显示的内容就不粘贴在这里了，请读者自己查看之，还是很有意思的。

```
mymodule already import
>>> dir(mymodule)
['__all__', '__builtins__', '__cached__', '__doc__', ......]
>>> dir(mymodule.mymod)
['__builtins__', '__cached__', '__doc__', '__file__', ......., 'myfun']
>>> print(mymodule.__doc__)
mymodule
我在 mymodule 文件夹的 __init__.py 文件里面
>>> print(mymodule.mymod.__doc__)
mymod.py
这是一个测试模块
>>> print(mymodule.mymod.myfun.__doc__)
myfun 函数
    用于打印测试
>>> help(mymodule)

>>> help(mymodule.mymod)

>>> help(mymodule.mymod.myfun)
```

类和模块的这些文档信息都存放在__doc__ 变量里面的。

文件处理高级知识

接下来的例子如果涉及到文件的请自己随便创建一个对应文件名的文件，内容随意了。

一行行的操作

因为文件对象本身是可迭代的，我们简单迭代文件对象就可以对文件的一行行内容进行一些操作。比如：

```
f = open('removeduplicate.py')

for line in f:
    print(line,end='')
```

这个代码就将打印这个文件，其中 end=" 的意思是取消\n，因为原来的行里面已经有\n 了。

然后代码稍作修改就可以在每一行之前加上 »> 这个符号了。

```
f = open('removeduplicate.py')

for line in f:
    print('>>>',line,end='')
```

什么？这个输出只是在终端，没有到某个文件里面去，行，加上 file 参数。

然后代码变成如下：

```
import sys

f = open('removeduplicate.py')
pyout=open(sys.argv[1] ,"w")

for line in f:
    print('>>>',line,end='',file=pyout)

pyout.close()
f.close()
```

这样我们就制作了一个小 python 脚本，接受一个文件名然后输出这个文件，这个文件的内容就是之前我们在终端中看到的。

整个文件的列表解析

python 的列表解析（迭代）效率是很高的，我们应该多用列表解析模式。

readlines 方法

文件对象有一个 readlines 方法，能够一次性把整个文件的所有行字符串装入到一个列表中。然后我们再对这个列表进行解析操作就可以直接对整个文件的内容做出一些修改了。不过不推荐使用 readlines 方法了，这样将整个文件装入内存的方法具有内存爆炸风险，而迭代版本更好一点。

文本所有某个单词的替换

这里举一个例子，将 removeduplicate.py 文件接受进来，然后进行列表解析，将文本中的 newlist 全部都替换为 list2。

```
import sys


pyout=open(sys.argv[1] ,"w")


print(''.join([line.replace('newlist','list2')
for line in open('removeduplicate.py')]),file=pyout)


pyout.close()
```

我们可以看到这种列表解析风格代码更加具有 python 风格和更加的简洁同时功能是异常的强大的。

从这里起我们看到如果需要更加复杂的文本处理技巧就需要学习正则表达式和 re 模块了，请参见 re 模块这一小节24。

与 c 语言或 c++ 语言编写的模块集成

安装和配置

通过 apt 安装

```
sudo apt-get install swig
```

在 Ubuntu14.04 这将安装 swig2.0 版本。

从 github 下载最新版安装

从 github 上下载最新版本：

```
git  clone  https://github.com/swig/swig
```

安装需要的前提软件：

```
sudo apt-get install autotools-dev
sudo apt-get install automake
sudo apt-get install byacc
sudo apt-get install yodl
```

安装：

```
./configure
make
sudo make install
```

安装后的配置

需要确认安装了 python3-dev，好支持`#include python.h`:

```
sudo apt-get install python3-dev
```

beginning

手工编译

找到源码的 [Examples]→[python]→[simple] 文件夹。简单的编译过程如下:

```
swig -python example.i

gcc -fpic -c example.c example_wrap.c -I/usr/include/python3.4m

gcc -shared example.o example_wrap.o -o _example.so
```

可以通过如下命令查看具体 **-I** 的引用地址:

```
python3-config --includes
```

简单的使用如下:

```
>>> import example
>>> example.gcd(100,25)
25
>>> example.cvar.Foo
3.0
>>>
```

通过 setuptools 安装

更加简便的处理方式是用 setuptools 模块来自动处理这一些，包括 build 到安装 egg 文件。

```
from setuptools import setup ,Extension


setup(
    name = 'example',
    version = '0.01',
    ext_modules = [Extension('_example', ['example.i','example.c'],
    swig_opts=['-modern', '-I../include'])],
    py_modules = ['example']
)
```

上面的 swig_opts 推荐加上。

编写自己的一个函数:

```
#include <stdio.h>
void hello(){
    printf("hello, world\n");
}
```

修改 example.i 文件:

```
%module example


%inline %{
extern int    gcd(int x, int y);
extern double Foo;
extern void hello();
%}
```

重新用 setuptools 安装一下。

```
>>> import example
>>> example.hello()
hello, world
```

常用的模块

这些常用的模块都是 python3 语言自带的模块。

pickle 模块

pickle 模块可以将某一个复杂的对象永久存入一个文件中，以后再导入这个文件，这样自动将这个复杂的对象导入进来了。

将对象存入文件

```
import pickle

class Test:
    def __init__(self):
        self.a=0
        self.b=0
        self.c=1
        self.d=1

    def __str__(self):
        return str(self.__dict__)

if __name__ == '__main__':
    test001=Test()
    print(test001)
    testfile=open('data.pkl','wb')
```

```
    pickle.dump(test001,testfile)
    testfile.close()
```

从文件中取出对象

值得一提的是从文件中取出对象，原来的类的定义还是必须存在，也就是声明一次在内存中的，否则会出错。

```
import pickle

class Test:
    def __init__(self):
        self.a=0
        self.b=0
        self.c=1
        self.d=1

    def __str__(self):
        return str(self.__dict__)

if __name__ == '__main__':
    testfile=open('data.pkl','rb')
    test001=pickle.load(testfile)
    print(test001)
    testfile.close()
```

pickle 模块的基本使用就是用 dump 函数将某个对象存入某个文件中，然后这个文件以后可以用 load 函数来加载，然后之前的那个对象会自动返回出来。

更多内容请参见官方文档。

shelve 模块

shelve 模块是基于 pickle 模块的，也就是只有 pickle 模块支持的对象它才支持。之前提及 pickle 模块只能针对一个对象，如果你有多个对象要处理，可以考虑使用 shelve 模块，而 shelve 模块就好像是自动将这些对象用字典的形式包装起来了。除此之外 shelve 模块的使用更加简便了。

存入多个对象

我们根据类的操作第三版中定义的类（5.5）建立了一个 Hero.py 文件，就是将那些类的定义复制进去。然后我们新建了几个实例来存入 test.db 文件中。

```
import shelve
from Hero import Garen

if __name__ == '__main__':
    garen1=Garen()
    garen2=Garen('red')
    garen3=Garen('yellow')
    db=shelve.open('test.db')
    for (key,item) in [('garen1',garen1),('garen2',garen2),('garen3',garen3)]:
        db[key]=item
    db.close()
```

我们看到整个过程的代码变得非常的简洁了，然后一个个对象是以字典的形

式存入进去的。

读取这些对象

读取这些对象的代码也很简洁，就是用 shelve 模块的 open 函数打开数据库文件，open 函数会自动返回一个字典对象，这个字典对象里面的数据就对应着之前存入的键值和对象。

同时通过这个例子我发现，如果自己定义的类，将他们提取出来放入另外一个文件，那么 shelve 模块读取文件时候是不需要再引入之前的定义。这一点值得我们注意，因为 shelve 模块内部也采用的是 pickle 的机制，所以可以猜测之前 pickle 的那个例子类的定义写在写入文件代码的里面，所以不能载入数据库；而如果将这些类的定义放入一个文件，然后这些类以模块或说模块载入的形式引入，那么读取这些对象就可以以一种更优雅的形式实现。如下所示：

```
import shelve


if __name__ == '__main__':
    db=shelve.open('test.db')
    for key in sorted(db):
        print(db[key])
    db.close()
```

我们看到就作为简单的程序或者原型程序的数据库，shelve 模块已经很好用而且够用了。

更多内容请参见官方文档。

time 模块

time 模块提供了一些和时间相关的函数，更加的底层，不过有些函数可能在某些平台并不适用。类似的模块还有 datetime 模块，datetime 是以类的框架来解决一些时间问题的。所以如果只是需要简单的调用一下时间，那么用 time 模块，如果是大量和时间相关的问题，推荐使用 datetime 模块。

time 函数

```
>>> import time
>>> time.time()
1404348227.07554
```

time 函数返回一个数值，这个数值表示从 1970 年 1 月 1 号 0 时 0 分 0 秒到现在的时间过了多少秒。

gmtime 函数

这个函数可以接受一个参数，这个参数是多少秒，然后返回一个特定格式的时间数组`struct_time`。如果不接受参数，那么默认接受的秒数由 time 函数返回，也就是从那个特定时间到现在过了多少秒，这样这个特定格式的时间数组对应的就是当前时间。

```
>>> time.gmtime()
time.struct_time(tm_year=2014, tm_mon=7, tm_mday=3, tm_hour=0,
tm_min=53, tm_sec=0, tm_wday=3, tm_yday=184, tm_isdst=0)
>>> time.gmtime(0)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=3, tm_yday=1, tm_isdst=0)
```

localtime 函数

此外类似的还有 **localtime** 函数，和 gmtime 用法和返回完全一模一样，唯一的区别就是返回的是当地的时间。

```
>>> time.strftime('%Y-%m-%d %H:%M:%S',time.localtime())
'2014-07-03 09:19:40'
>>> time.strftime('%Y-%m-%d %H:%M:%S',time.gmtime())
'2014-07-03 01:19:49'
```

ctime 函数

```
>>> time.ctime()
'Thu Jul  3 08:54:54 2014'
>>> time.ctime(0)
'Thu Jan  1 07:00:00 1970'
```

和 gmtime 类似，不过返回的是字符串格式的时间。我们看到 ctime 默认设置的时间是根据 localtime 函数来的。

strftime 函数

接受那个特定格式的时间数组struct_time 作为参数，然后返回一定字符串格式的时间。具体例子请参看前面的例子。

其中最常用的格式符有：

%Y，多少年；%m，多少月；%d，多少日；
%H，多少小时；%M，多少分；%S，多少秒。

%X 直接输出 09:27:19 这样的格式，也就是前面的多少小时多少分多少秒可以用一个%X 表示即可。

还有一些，比如：%I 表示多少小时，不过是 [0-12] 的形式；%y 表示多少年，不过是 [00-99] 的格式，比如 2014 年就输出 14；%p，本地的 AM 或 PM 文字。等等。

sleep 函数

sleep 函数有时需要用到，将程序休眠个几秒的意思。需要接受一个数值参数，单位是秒，可以是零点几秒。但 sleep 函数只是大概休眠几秒的意思，最好不去用来计时，因为它不大精确。

更多内容请参见官方文档。

sys 模块

sys 模块有一些功能很常用，其实在前面我们就看到过一些了。

sys.argv

在刚开始说明 python 执行脚本参数传递的问题时就已经讲了 sys.argv 这个变量。这是一个由字符串组成的列表。

```
import sys


print(sys.argv)
for i in range(len(sys.argv)):
    print(sys.argv[i])
```

比如新建上面的一个 test.py 文件，然后执行：

```
python3 test.py test1 test2
['test.py', 'test1', 'test2']
test.py
test1
test2
```

我们可以看到 sys.argv[0] 就是这个脚本的文件名，然后后面依次是各个参数。

exit 函数

这个我们在编写 GUI 程序的时候经常看到，在其他脚本程序中也很常用。如果不带参数的话那么直接退出程序，还可以带一个字符串参数，返回错误提示信息，或者带一个数字，这里的详细讨论略过。

```
>>> import sys
>>> sys.exit(' 出错了')
出错了
wanze@wanze-ubuntu:~$
```

sys.platform

返回当前脚本执行的操作系统环境。

Linux 返回字符串值：linux；Windows 返回 win32；Mac OS X 返回 darwin。

sys.path

一连串字符串列表，是 python 脚本模块的搜索路径，所以我们自定义的 python 模块，只需要在 sys.path 这个列表上新加一个字符串路径即可。

标准输入输出错误输出文件

sys.stdin，sys.stdout，sys.stderr 这三个文件对象对应的就是 linux 系统所谓的标准输入标准输出和错误输出文件流对象。

sys.version

sys.version 输出当前 python 的版本信息和编译环境的详细信息。

```
sys.version_info[0]
```

返回当前 python 主版本的标识，比如 python3 就返回数字 3。

sys.maxsize

返回当前计算环境下整数 (int) 类型的最大值，32 位系统是 $2**31-1$。

```
>>> 2**31-1
2147483647
>>> import sys
>>> sys.maxsize
2147483647
```

sys.stdin.isatty()

测试输入流是不是终端。如果是终端，则返回 True。

更多内容请参见官方文档。

fileinput 模块

fileinput 模块提供了便捷的多文件处理方案。

input 函数

```
fileinput.input(files=None, inplace=False)
```

input 函数最常用的两个参数如上所示，如果不带参数，那么就是默认的命令行的 sys.argv[1:] 接受到的一系列文件。如果命令行也没有输入文件参数，那么就是默认的 sys.stdin 标准输入。特定的文件输入用 **files** 来指定，参数为文件名字符值的列表。

inplace 默认是 False，也就是原文件没有被修改，如果设置为 True，那么对于 line 的一些操作是会原地修改原文件的。具体请看下面的例子：

```
allfile = [os.path.join(path,f) for path,dirs,files in os.walk('.')
        if files for f in files]
def process(line):
    return line.replace('skeleton',project_name)
with fileinput.input(files=allfile,inplace=1) as f:
    for line in f:
```

```
        print(process(line),end='')
```

上面代码第一行使用用 os.walk 来生成本文件夹下所有的文件名路径。然后作为参数输入给 files，这里一般定义一个 process 函数来处理每一行，然后返回处理后的值。用 print 函数打印具体的处理的效果。如果这里 inpalce 没有设置为 True 那么结果只是打印在终端上，设置为 True 之后，原文件对应行将被修改。inplace 参数很有用[①]，在你确认操作结果的情况下慎重使用。

① 参考网站

os.path 模块

前面提到 sys.argv 只能返回当前 python 脚本的文件名，而我们常常需要这个 python 脚本在系统中的具体位置。前面如 os.getcwd 等也能获得当前 python 脚本的所在目录，不过 os.path 模块的一个优点就是跨平台特性支持很好，也就是一般我们通过其他方式获得的 path 路径都会用这个模块的函数辅助处理一下。

我们来看下面的例子：

```
import os


print(os.path.abspath(__file__))
print(os.path.dirname(os.path.abspath(__file__)))


print(os.path.basename(__file__))
print(os.path.basename(os.environ['HOME']))
```

```
/home/wanze/桌面/test.py
/home/wanze/桌面
test.py
wanze
```

其中__file__ 表示当前脚本文件所在的路径。

abspath 函数

abspath 函数接受一个 path 路径值然后返回一个正规的普适的路径地址。具体效果类似于执行了: `normpath(join(os.getcwd(), path))` 。

再看下面的例子演示了空字符串默认当前工作目录，然后也接受绝对路径等。

```
>>> import os
>>> os.path.abspath('')
'/home/wanze'
>>> os.path.abspath('test')
'/home/wanze/test'
>>> os.path.abspath('/test')
'/test'
>>> os.path.abspath('test/')
'/home/wanze/test'
```

我们看到如果 abspath 接收的是空字符串，其定位是当前脚本的工作目录，那么是引用的模块里面的`os.path.abspath('')`，具体对应的也是当前脚本的工作目录。然后 os.path.abspath(".") 返回的是当前脚本工作目录。

dirname 函数

dirname 函数接受一个路径值然后返回这个路径除开最后一个元素的前面的路径值。比如上面的例子，路径指向文件，那么 dirname 函数返回的是除开这个文件名的前面的路径；而如果接受的路径指向目录，那么返回的是除开最后一个文件夹名的前面的路径值。

basename 函数

如上面例子所示，basename 函数接受一个路径值然后返回路径的最后一个元素，如果路径指向文件，那么返回的是文件名；如果路径指向目录，那么返回的是最后那个目录的文件夹名。比如下面实现了从绝对路径提取出文件名的功能。

```
>>> import os.path
>>> string = '/home/wanze/test.txt'
>>> fileName,fileExtension = os.path.splitext(os.path.basename(string))
>>> fileName
'test'
```

split 函数

将路径 path 字符串分割，可以视作 dirname 和 basename 的组合。

```
>>> os.path.split('/usr/local/bin/test.txt')
('/usr/local/bin', 'test.txt')
>>> os.path.dirname('/usr/local/bin/test.txt')
'/usr/local/bin'
>>> os.path.basename('/usr/local/bin/test.txt')
'test.txt'
```

splitext 函数

将某个路径 path 的后缀分开，这里主要是针对文件名为输入的时候，那么第一个为该文件的名字，输出数组的第二个值是该文件的后缀。这个函数在提取

文件名后缀和前面的名字的时候很有用，方便组合出新的文件名。

```
>>> import os
>>> fileName, fileExtension = os.path.splitext('/path/to/somefile.ext')
>>> fileName
'/path/to/somefile'
>>> fileExtension
'.ext'
```

join 函数

用于连接多个路径值合并成一个新的路径值，同样相对于简单的字符串拼接，用这个函数处理路径组合具有操作系统普适性和灵活性。

```
>>> os.path.join(os.path.expanduser('~'),'test','lib')
'/home/wanze/test/lib'
```

上面 join 函数多个参数生成的新 path 在 windows 下又是不同的输出的。

expanduser 函数

```
>>> import os
>>> os.path.expanduser('~')
'/home/wanze'
>>> os.path.expanduser('~/pymf')
'/home/wanze/pymf'
>>> os.path.join(os.path.expanduser('~'),'pymf','mymodule')
'/home/wanze/pymf/mymodule'
```

~ 这个符号可以在这里使用，从而展开为以/home/wanze 为基础的绝对路径，兼容大部分系统（在 windows 下也可以使用。）

同时我们看到 join 函数可以接受很多不定量的参数，然后将他们组合成为一个新的路径，而且不用你费心是/还是\，你不需要写这些了，用 join 函数自然料理好一切。

exists 函数

os.path.exists(path)：测试路径或文件等是否存在。如果存在返回 True，否则返回 False。

isfile 和 isdir 还有 islink

os.path.isfile(path)：接受一个字符串路径变量，如果是文件那么返回 True，否则返回 False（也就是文件不存在或者不是文件是文件夹等情况都会返回 False）。

类似的有 isdir 和 islink 函数。

samefile 函数

os.path.samefile(path1,path2)：如果两个文件或路径相同则返回 True，否则返回 False。

getmtime 函数

os.path.getmtime(path)

返回文件的最后修改时间，返回值是多少多少秒，可用 time 模块的 ctime 或 localtime 函数将其转换成 time.struct_time 对象，然后使用 strftime 来进行更好的格式输出。

getctime 函数

类似 getmtime，返回文件的最后创建时间。在 unix 系统中是指最后文件的元信息更改的时间。

更多内容请参见官方文档。

glob 模块

glob 模块用法很简单，初步学习就是一个 glob 函数，接受一个 pathname 路径值，然后返回这个路径下某些文件名组成的列表。支持* ？ ，意思是任意数量的字符或者任意的一个字符，然后[?] 明确表示问号。

```
>>> import glob
>>> glob.glob('*.py')
['re_subst.py', 're_sub.py', 'test2.py']
```

subprocess 模块

我想大家都注意到了现在的计算机都是多任务的，这种多任务的实现机制就是所谓的多个进程同时运行，因为计算机只有一个 CPU（现在多核的越来越普及了，它们内部的工作原理我没了解过。）所有计算机一次只能处理一个进程，而这种多进程的实现有点类似你人脑（当然不排除某些极个别现象），你不能一边看电影一边写作业，但是可以写一会作业然后再看一会电影（当然不推荐这么做、），计算机的多进程实现机制也和这个类似，就是一会干这个进程，一会儿做那个进程。

计算机的一个进程里面还可以分为很多个线程，这个较为复杂，就不谈了。比如你编写的一个脚本程序，系统就会给它分配一个进程号之类的，然后 cpu 有时就会转过头来执行它一下（计算机各个进程之间的切换很快的，所以才会给我们一种多任务的错觉。）而你的脚本程序里面还可以再开出其他的子进程出来，python 的 subprocess 模块主要负责这方面的工作。

call 函数

```
import subprocess


# Command with shell expansion
subprocess.call(["echo", "hello world"])
subprocess.call(["echo", "$HOME"])
```

```
subprocess.call('echo $HOME',shell=True)
```

```
hello world
$HOME
/home/wanze
```

其中使用 shell=True 选项后用法较简单较直观，但网上提及安全性和兼容性可能有问题，他们推荐一般不适用 shell=True 这个选项。 参考网站

如果不使用 shell=True 这个选项的，比如这里$HOME 这个系统变量就无法正确翻译过来，如果实在需要 home 路径，需要使用 os.path 的 expanduser 函数。

getoutput 函数

取出某个进程命令的输出，返回的是字符串形式。

```
import subprocess

name=subprocess.getoutput('whoami')
print(name)
```

getstatusoutput 函数

某个进程执行的状态。

Popen 类

根据 Popen 类创建一个进程管理实例，可以进行进程的沟通，暂停，关闭等等操作。前面的函数的实现是基于 Popen 类的，这是较高级的课题，这里暂时略过。

更多内容请参见官方文档。

os 模块

getcwd 函数

不管你在终端运行 python 还是运行某个 python 脚本，总有一个变量存储着当前工作目录的位置。你可以通过 getcwd 命令来查看当前工作目录。

```
import os
print(os.getcwd())
```

上面是通过 LaTeX 文件运行的 python 小脚本，当你以 python 命令来运行某个脚本的时候，你调用 python 命令的地方就是当前的工作目录[①]。然后加载的其他模块的各个 py 文件运行时的当前工作目录和主 py 文件脚本的当前目录是一样的，都是你运行 python 命令的地方。

如果是终端调用 python 就是你终端的当前工作目录所在，你可以用 pwd 命令来查看。如下所示：

```
=>pwd
/home/wanze
=>python3
>>> import os
```

① 这里在 LaTeX 文档下的情况有点小复杂，通过我编写的 xverbatim.sty 我们可以看到当时运行 python3 命令的当前工作目录就在这个 tex 文档所在的目录下。

```
>>> print(os.getcwd())
/home/wanze
```

mkdir 函数

新建一个文件夹。

```
os.mkdir(str)
```

chdir 函数

os 模块里有一个 chdir 函数来更改当前工作目录所在地。

可以使用.和..语法，也可以使用简单的”test” 调转到 test 文件夹。

```
>>> os.chdir('/home/wanze/pymf')
>>> print(os.getcwd())
/home/wanze/pymf
```

删除文件

os.remove(path)

支持相对路径表达。如果路径是目录将会抛出一个 OSError 异常。

os.rename

```
os.rename(src, dst)
```

第一个参数是目标文件或目录，第二个参数是要替换成为的名字。这个命令一方面可以重命名文件，此外可以移动文件。

支持相对路径语法表达，rename 在 windows 下不一定替换原文件，repalce 一定替换文件。

os.repalce

```
os.replace(src, dst)
```

rename 在 windows 下不一定替换原文件，repalce 一定替换文件。

支持相对路径语法表达。

删除空目录

os.rmdir(path)

支持相对路径语法表达，只能删除空目录。如果要删除整个目录，请使用 shutil.rmtree(path)。

listdir 命令

```
os.listdir(path='.')
```

相当于简单的 ls 命令，将返回一个字符串列表，其内包含本 path 下所有的文件和文件夹名（包括链接文件）。

可以结合前面介绍的 os.path 模块的 isfile 等函数新建一个函数 listdir_file，listdir_dir 和 listdir_link，将普通文件，目录和链接文件区分开来。

```
import os

def listdir_dir(path='.'):
    '''os 的 listdir 函数加强，只返回文件夹。'''
    return [dir for dir in os.listdir(path) if os.path.isdir(dir) ]
def listdir_file(path='.'):
    '''os 的 listdir 函数加强，只返回普通文件'''
    return [file for file in os.listdir(path) if os.path.isfile(file)
     and  not os.path.islink(file)]
def listdir_link(path='.'):
    '''os 的 listdir 函数加强，只返回链接文件'''
    return [link for link in os.listdir(path) if os.path.islink(link) ]
```

遍历目录树

```
os.walk('.')
```

产生一个生成器对象，具体数值含义如下：（dirpath, dirnames, filenames），其中 dirpath 和 filenames 可以合并出本目录下所有文件的具体文件名路径，而 dirpath 和 dirnames 可以合并出本目录下所有目录的具体路径名。

根据这个 os.walk 函数我写了一个`gen_file` 函数，其是一个生成器函数，会遍历目录树，并返回本目录下的文件信息。具体代码如下所示：

```
def gen_file(startpath='.',filetype=""):
    ''' 利用 os.walk 遍历某个目录，收集其内的文件，返回
    (文件路径列表，本路径下的文件列表)
    比如：
    (['shortly'], ['shortly.py'])
(['shortly', 'templates'], ['shortly.py'])
(['shortly', 'static'], ['shortly.py'])

    第一个可选参数 startpath  默认值 '.'
    第二个参数  filetype  正则表达式模板 默认值是"" 其作用是只选择某些文件
    如果是空值，则所有的文件都将被选中。比如 "html$|pdf$" 将只选中 html 和 pdf 文件。
    '''
    for root, dirs, files in os.walk(startpath):
        filelist = []
        for f in files:
            fileName,fileExt = os.path.splitext(f)
            if filetype:
                if re.search(filetype,fileExt):
                    filelist.append(f)
            else:
                filelist = files
        if filelist:# 空文件夹不加入
            dirlist = root.split(os.path.sep)
            dirlist = dirlist[1:]
            if dirlist:
                yield (dirlist, filelist)
            else:
                yield (['.'], filelist)
```

这个函数可以帮助你管理本目录下（可以通过正则表达式过滤）你感兴趣的文件，都刷一边。然后继续必要的操作，比如查找等等之类的。

environ 函数

os.environ，返回一个字典值，这个字典值里面存储着当前 shell 的一些变量和值。比如系统中“HOME”所具体的路径名是：

```
import os
print(os.environ['HOME'])
```

```
/home/wanze
>>>
```

---

getpid 函数

os.getpid 函数，返回当前运行的进程的 pid。

stat 函数

返回文件的一些信息。比如 st_size 是文件的大小，单位是字节。

st_size 属性

```
import os
import glob


print([os.path.abspath(f) for f in glob.glob('*.py')])
```

```
print([f for f in glob.glob('*.py') if os.stat(f).st_size > 400])
```

```
['/home/wanze/桌面/test.py', '/home/wanze/桌面/flatten.py']
['flatten.py']
```

下面这个例子进行了文件大小输出单位的优化：

```
import os
import sys

filename = sys.argv[1]
filesize = os.stat(filename).st_size

for unit in [' 字节','KB','MB','GB','TB']:
    if filesize > 1024:
        filesize = filesize/1024
    else:
        break

print(filename + ' 大小是' +str(int(filesize)) + unit)
```

这个 python 小脚本自动输出合适的单位，具体程序逻辑还是很简单的。

st_mtime 属性

最后文件修改的时间。

st_ctime 属性

最后文件创建的时间，在 windows 下是严格的最初文件创建时间，在 unix 下是最后文件 metadata 的改变时间。

给进程发送信号

可以通过 os 模块的 kill 函数来给某个进程发送某个信号。

```
os.kill(pid, sig)
```

函数第一个参数是进程的 pid，第二个参数是具体发送的信号。比如:

```
os.kill(pid, signal.SIGSTOP)
```

就是暂停某个进程，然后

```
os.kill(pid, signal.SIGCONT)
```

是继续某个进程。然后 **killpg** 函数能够对某个进程包括其子进程发送某个信号，参考了这个网页。

除此之外还有 **SIGINT** （正常终止进程信号）和 **SIGKILL** （强制终止进程信号）等等，更多信号请参看关于 unix 信号那块，比如这个 wiki 页面 。

更多 os 模块内容请参见官方文档。

shutil 模块

相当于 os 模块的补充，shutil 模块进一步提供了一些系统级别的文件或文件夹的复制，删除，移动等等操作。

复制文件

```
shutil.copyfile(src, dst)
shutil.copy(src, dst)
shutil.copy2(src, dst)
```

其中 **copyfile** 的 src 和 dst 两个参量都是完整文件路径名，第一个参量是待复制的文件，第二个参量是复制后的文件名；而 **copy** 函数的第一个参量是待复制的文件，但是第二个参量是目标文件夹路径；**copy2** 函数和 copy 函数类似，不同的是它能尝试保留文件的所有元信息 metadata（模块开头有说明是理论上但不尽然）。

复制文件夹

```
shutil.copytree(src, dst)
```

copytree 函数第一个参量是待复制的文件夹路径名，第二个参量是目标文件夹路径名，其将被创建不应该存在。

删除整个目录

```
shutil.rmtree(path)
```

rmtree 函数用于删除整个文件夹，path 就是目标文件夹的路径名。

移动文件夹

```
shutil.move(src,dst)
```

move 函数把一个文件或者一个文件夹移动到一个文件夹内。

chown 函数

```
shutil.chown(path, user=None, group=None)
```

chown 函数类似的 linux 系统下的 chown 函数，这个函数基于 os.chown 函数，不过接口更友好。

which 函数

```
shutil.which(cmd)
```

which 函数类似的 linux 系统下的 which 函数。

更多 shutil 模块内容请参见官方文档。

tarfile 和 zipfile 模块

tarfile 是 gzip，bz2 和 lzma 压缩文件读写的解决方案，zipfile 模块是 zip 压缩文件的解决方案，值得一提的是 pip 管理的 egg 文件也可以通过 zipfile 模块来管理。

制作 gz 压缩文件

请看下面的例子：

```
import tarfile
with tarfile.open("skeleton.tar.gz", "w:gz") as tar:
    for name in ["setup.py","LICENSE","README.md","skeleton", "docs"]:
        tar.add(name)
```

这里首先用 tarfile 模块的 **open** 函数来返回一个 TarFile 对象，其中第一个参数是你的压缩文件的名字，第二个参数是处理模式。

模式可接受的参数如下：

r 默认值是 r，就是只读某个压缩文件。类似有 **r:gz**，**r:bz2** 和 **r:xz**，这里的意思就是具体设置好要读的压缩文件的格式（gzip，bzip2 和 lzma）。

w 类似的还有 **w:gz**，**w:bz2**，**w:xz**。这里 **w** 或者 **w:** 官方文档的说明是（Open for uncompressed writing），我对这个无压缩方式写不是很理解。

a 还有 **a:**，Open for appending with no compression. 文件如果不存在将被创建。

TarFile 的 add 方法

然后接下来就是往压缩文件里面添加内容（文件或者整个目录），具体就是用创建的 TarFile 对象的 add 方法，如上例子所示。

解压缩 gz 压缩文件

最简单的例子如下所示：

```
with tarfile.open("skeleton.tar.gz") as tar:
    tar.extractall()
```

TarFile 的 extractall 方法

用 tarfile 模块的 open 函数打开那个压缩文件，用返回的 TarFile 对象的 extractall 方法解压缩这个文件，注意用 os.chdir 来控制当前工作目录。

更多 tarfile 模块内容请参见官方文档。

提取 egg 文件中的内容

简单的例子如下所示：

```
zip=zipfile.ZipFile("test.egg")
zip.extract('test.txt')
```

这里用 zipfile 模块的 ZipFile 构造函数创建了一个 ZipFile 对象，然后用 ZipFile 的 **extract** 方法提取出了 test.txt 文件在当前工作目录。

相关的 **extractall** 方法将会提取出压缩文件中所有的内容。

制作 zip 压缩文件

简单的示例如下：

```
with zipfile.ZipFile('test.zip','w') as zip:
    zip.write('test2.png')
```

首先用 zipfile 模块的 ZipFile 构造函数创建一个 ZipFile 对象，这里 mode 需要使用**'w'**，然后使用 ZipFile 对象的 write 方法来添加内容。你可以猜到如果模式是**'a'** 的话 write 方法是给这个压缩文件添加内容（a 模式同文件操作含义如果原压缩文件不存在也是可以创建的）。

更多 zipfile 模块内容请参见官方文档。

collections 模块

namedtuple 函数

collections 模块里面的 namedtuple 函数将会产生一个有名字的数组的类（有名数组），通过这个类可以新建类似的实例。比如：

```
from collections import namedtuple


Point3d=namedtuple('Point3d',['x','y','z'])
p1=Point3d(0,1,2)
print(p1)
print(p1[0],p1.z)
```

```
Point3d(x=0, y=1, z=2)
0 2
```

Counter 计数类

更多内容请参见官方文档。

re 模块

re 模块提供了 python 对于正则表达式的支持，对于字符串操作，如果之前在介绍字符串类型的一些方法（比如 split，replace 等等），能够用它们解决问题就用它们，因为更快更简单。实在需要动用正则表达式理念才考虑使用 re 模块，而且你要克制写很多或者很复杂的（除非某些特殊情况）正则表达式的冲动，因为正则表达式的引入将会使得整个程序都更加难懂和不可捉摸。

更多内容请参见官方文档。

re 模块中的元字符集

. 表示一行内的任意字符，如果如果通过 re.compile 指定 **re.DOTALL**，则表示多行内的任意字符，即包括了换行符。此外还可以通过字符串模板在它的前面加上 **(?s)** 来获得同样的效果。

***** 对之前的字符匹配零次或者多次。

+ 对之前的字符匹配一次或者多次。

? 对之前的字符匹配零次或者一次。

{m} 对之前的字符匹配 (exactly)m 次。

{m,n} 对之前的字符匹配 m 次到 n 次，其中 n 次可能省略，视作默认值是无穷大。

^ 表示字符串的开始，如果加上 **re.MULTILINE** 选项，则表示行首。此外字符串模板加上 **(?m)** 可以获得同样的效果。

$ 表示字符串的结束，同 ^ 类似，如果加上 **re.MULTILINE** 选项，则表示行尾，可以简单理解为\n 换行符。此外字符串模板加上 **(?m)** 可以获得同样的效果。

$ 符号在 re.sub 函数中可以被替换为另外一个字符串，其具体效果就是原字符串尾加上了这个字符串，类似的 ^ 被替换成某个字符串，其具体效果就是原字符串头加上了这个字符串。这里显然 ^ 和 $ 在字符串中都不是真实存在的字符，而没有这个所谓的标记，所以这种替换总给人怪怪的感觉。

[] [abc] 字符组匹配一个字符，这个字符是 a 或者 b 或者 c。类似的 [a-z] 匹配所有的小写字母，[\w] 匹配任意的字母或数字，具体请看下面的特殊字符类。

| 相当于正则表达式内的匹配或逻辑。

() 圆括号包围的部分将会记忆起来，方便后面调用。这个后面在谈及。

re 模块中的特殊字符类

\w　任意的字母或数字　[a-zA-Z0-9_]　(meaning word)

\W　匹配任何非字母非数字 [^a-zA-Z0-9_]

\d　[0-9]　(digit) 数字

\D　[^0-9] 非数字

\s　匹配任何空白字符　[\t\n\r\f\v] 。

\S　匹配任何非空白字符

匹配中文:[\u4e00-\u9fa5]

\b　文档说严格的定义是 \w 和 \W 之间的边界，反之亦然。粗略的理解可以看作是英文单词头或者尾。

其中 ^ 在方括号 [] 里面，只有在最前面，才表示排除型字符组的意思。

转义问题

正则表达式的转义问题有时会比较纠结。一个简单的原则是以上谈及的有特殊作用的字符有转义问题，如果 python 中的字符都写成`r''`这种形式，也就是所谓的 raw string 形式，这样`\n`在里面就可以直接写成`\n`，而`\section`可以简单写为`\\section`即可，也就是`\`字符需要转义一次。

然后字符组的方括号内 [] 有些字符有时是不需要转义的，这个实在不确定就转义吧，要不就用 Kiki 测试一下。

re 模块的使用

compile 方法生成 regular expression object 这一条线这里略过了，接下来的讨论全部基于（原始的）字符串模板。

字符串模板前面提及 (?m) 和 (?s) 的用法了，然后 **(?i)** 表示忽略大小写。

匹配和查找

search，match 方法简单地用法就是：

```
re.search(字符串模板, 待匹配字符串)
re.match(pattern, string)
```

它们将会返回一个 match object 或者 none，其中 match object 在逻辑上就是真值的意思。match 对字符串的匹配是必须从一开始就精确匹配，这对于正则表达式多少 0 有点突兀。推荐使用 search 方法，如果一定要限定行首，或者字符串开始可以用前面讨论的正则表达式各个符号来表达。请看下面的例子。

```
import re
string = '''this is test line.
this is the second line.
today is sunday.'''

match = re.search('(?m)^today',string)

if match:
    print(' 所使用的正则表达式是: ',match.re)
    print(' 所输入的字符串是: ',match.string)
    print(' 匹配的结果是: ',match.group(0))
    print(' 匹配的字符串 index',match.span())
else:
    print('return the none value')
```

前面说道圆括号的部分将会记忆起来，作为匹配的结果，默认整个正则表达式所匹配的全部是 group 中的第 0 个元素，然后从左到右，子 group 编号依次是 1，2，3.....。

```
所使用的正则表达式是:  re.compile('(?m)^today', re.MULTILINE)
所输入的字符串是:  this is test line.
this is the second line.
today is sunday.
匹配的结果是:  today
匹配的字符串 index (44, 49)
```

具体这些信息是为了说明情况，实际最简单的情况可能就需要判断一下是不是真值，字符串模板是不是匹配到了即可。

分割操作

re 模块的 split 函数可以看作字符串的 split 方法的升级版本，对于所描述的任何正则表达式，匹配成功之后都将成为一个分隔符，从而将原输入字符串分割开来。

下面是我写的 zwc 小脚本的最核心的部分，用途是统计中英文文档的具体英文单词和中文字符的个数。其中最核心的部分就是用的 re 的 split 函数进行正则表达式分割，如果不用那个圆括号的话，那么分隔符是不会包含进去的，这里就是具体匹配的中文字和各个标点符号等等。用了圆括号，那么圆括号匹配的内容也会进去列表。这里就是具体的各个分隔符。

```
import re

def zwc(string):
    # 中英文常用标点符号
    lst = re.split('([\u4e00-\u9fa5\s，。；])',string)
    # 去除 空白
    # 去除 \s 中英文常用标点符号
    lst = [i for i in lst if not  i in
    ["","　","\n","\t","\r","\f","\v","；","，","。"]]
    print(lst)

if __name__ == '__main__':
    string='''道可道，非常道。名可名，非常名。無名天地之始，有名萬物之母。
    故常無欲，以觀其妙；常有欲，以觀其徼。此兩者同出而異名，
    同謂之玄，玄之又玄，眾妙之門。　'''
    zwc(string)
```

字符分割之后后面做了一个小修正，将匹配到的空白字符和中英文标点符号等都删除了，这些是不应该统计入字数的。

具体这个 github 项目链接在这里：zwc 项目。

替换操作

基于正则表达式的替换操作非常的有用，其实前面的 search 方法，再加上具体匹配字符串的索引值，然后修改原字符串，然后再 search 这样循环操作下去，就是一个替换操作了。re 模块有 sub 方法来专门解决这个问题。

让我们为 Linux 系统写一个 resub 命令，这个命令的用途就是将某一个标准输入流或者 utf-8 文本文件按照你定义的正则表达式规则，依次完成一个[①]正则表达式文本替换工作。这个命令在我们需要对某个 utf-8 文本文件进行某个你想要的——非简单的精确相同匹配然后替换操作时——特别有用。

为了作为程序的检验，这里提出两个任务：第一个任务是我们在 ocr PDF 文档之后的输出，经常发现很多标点符号问题，这些需要人手工修改会非常的耗费精力。其中第一个问题如下，”这是一段文字”需要替换成为“这是一段文字”。这个例子之所以特别是因为中文的双引号是分左和右的，这里必须要用正则表达式匹配和替换；第二个任务更加的复杂，那就是从排版角度上讲，如果括号里面的文字都是英文或者数字，那么就使用英文的括号 ()，如果括号里面有中文或者全是中文[②]，那么就使用中文的括号（）。ocr 出来或者甚至人编写的文档都常常难以做到没有瑕疵，第二个任务就是通过 resub 命令来确保之后的输出文档的括号满足这一要求。

然后程序还需要建立两个选项，一个是自动替换所有，一个是对于每一个替换操作都请求确认——需要打印相关信息。

程序需要经过如下几个阶段：1. 明确匹配模板 1.1 写出字符串模板匹配操作给出匹配的所有情况，最好是行模板匹配模式。最后明确匹配情况 2. 明确匹配的文字的后给出情况

① 这里简单起见就是一个，多个情况可以考虑编写另外一个程序来控制之。

② 这里程序的逻辑是都换成中文的全角括号（毕竟中文 unicode 码具体范围的判断是不太精确的），只有那些纯英文纯数字或者基本英文标点和其他简单符号的再换成英文括号

itertools 模块

repeat 函数

其定义函数如下：

```
def repeat(object, times=None):
    # repeat(10, 3) --> 10 10 10
    if times is None:
        while True:
            yield object
    else:
        for i in range(times):
            yield object
```

也就是返回一个可迭代对象，这么封装最大的一个用处是用于填充 map 函数或者 zip 函数的某个常数值。因为你填写 repeat(5) 之后将一个返回一个可迭代对象，不停的返回数字 5 而不需要你考虑长度问题。

starmap 函数

starmap 函数具体定义如下所示：

```
def starmap(function, iterable):
    # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
    for args in iterable:
        yield function(*args)
```

其接受一个可迭代对象，然后逐个将可迭代对象中的元素解包之后送入函数当参数（最后当然函数也执行了）。

multiprocessing 模块

multiprocessing 为 python 提供了多进程（或者多线程）的解决方案。

Pool 类

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

这里 Pool 的第一个参数是可选项 **processes**，如果不设置，则默认的是`os.cpu_count()`，即当前系统的 cpu 数。

一般 cpu 密集型的任务推荐使用多进程处理，当然前提是你有多个 cpu 的情况下，如果你并没有相关硬件条件，追求时髦动用多进程还不如用单进程，集中精力用好的算法办好一件事。不过不同进程（不同程序）之间的通信还是很有用的。

这里的 map 是 Pool 对象的 **map** 方法，其除了接受第一个列表参数，然后用多进程分别处理这些列表中的元素之外，并不能在额外接受其他参数了。如果

你想要接受一些参数，推荐使用 **starmap** 方法。

starmap 方法

starmap 类似 map 方法，不过其接受的是一系列的函数参数。值得注意的是 python 的 itertools 模块25里面提供的 starmap，repeat 还有 python 语言的 zip 和 map 函数等在此处有时会很有用，可能是你感兴趣的。

```
from multiprocessing import Pool


def f(x,y):
    return (x*x+y*y)

if __name__ == '__main__':
    with Pool(5) as p:
        print(p.starmap(f, [(1,1),(2,2),(3,3)]))
```

在看一个有可选参数的情况：

```
from multiprocessing.dummy import Pool as ThreadPool
from itertools import repeat


urls = get_all_urls(base_url,model_url)


with ThreadPool(6) as tp:
    imgs = tp.starmap(get_url_imgs,zip(urls,repeat(inclass))
```

这里的`get_url_imgs` 除了接受第一个 url 参数之外，还接受一个可选参数 inclass，这里通过 repeat 封装之后将这两个参数用 zip 函数封装即能达到这样的执行效果：

```
get_url_imgs(url,inclass)
get_url_imgs(url,inclass)
get_url_imgs(url,inclass)
.....
```

然后这些都是多线程运行的。

ThreadPool 类

一般和网络相关的 I/O 密集型任务推荐使用多线程处理，但也不是线程设置得越多越好，一般设置为当前 CPU 数 *2+2 左右，再多并没有起到提速效果，有时反而会降低速度。

multiprocessing 还有一个 dummy 子模块，其提供了和 multiprocessing 一样的 API[①]，不同的是 multiprocessing 作用于进程，而 dummy 子模块是作用于线程的。

```
from multiprocessing.dummy import Pool as ThreadPool

with ThreadPool(6) as tp:
    imgs = tp.map(partial_get_url_imgs,urls)
```

① 请参看这个网页。

python3 高级篇

python 的多任务处理

进程的定义是：一个正在执行的程序实例。每个进程都有一个唯一的进程 ID，也就是所谓的 **PID** 。使用`ps` 命令的第一个列就是每个进程的 PID 属性。在 python 中你可以使用`os.getpid()` 来查看当前进程的 PID。

以前只有一个 CPU 的机器上，多任务操作系统实际上一次也只能运行一个进程，操作系统是通过不断切换各个进程给你一种多任务似乎同时在运行多个程序的感觉的。多 CPU 机器上是真的可以同时运行多个进程。

进程 fork

进程 fork 简单来说就类似于 git 某个项目的 fork，进行了一些基本代码信息和其他配置以及其他相关信息的复制或注册。这就相当于在当前代码环境下，你有两个分别单独运行的程序实例了。

下面是一个非常简单的小例子，你可以把 os.fork() 语句移到 print('before fork') 之前来看看变化。

```
import os, time


print('before fork ')
os.fork()
```

```
print('say hello from', os.getpid())


time.sleep(1)


print('after fork')
```

对于这个程序简单的理解就是，本 py 文件编译成字节码进入内存经过某些成为一个程序实例了（其中还包含其他一些信息），然后程序具体运行的时候会通过 os.fork 来调用系统的 fork 函数，然后复制本程序实例（以本程序实例目前已经所处的状态），因为 print('before fork') 已经执行了，所以子进程就不会执行这一行代码了，而是继续 os.fork() 下面的代码继续执行。此时就相当于有两个程序在运行了，至于后面的打印顺序那说不准的。

关于操作系统具体如何 fork 的我们可以暂时不考虑，这两个程序实例里面的变量和运行环境基本上是一模一样的，除了运行的状态有所不同之外。fork 可以做出一种程序多任务处理方案吧，不过 os 模块的 fork 方法目前只支持 unix 环境。

子进程和父进程分开

请看下面的代码:

```
import os, time


print('before fork ')
pid = os.fork()
if pid:
    print(pid)
    print('say hello from parent', os.getpid())
else:
```

```
    print(pid)
    print('say hello from child', os.getpid())

time.sleep(1)

print('after fork')
```

其运行结果大致如下:

```
before fork
13762
say hello from parent 13761
0
say hello from child 13762
after fork
after fork
```

我们看到在父进程那一边，pid 是本父进程的子进程 PID，而在子进程那一边，os.fork() 返回的是 0。可以利用这点将父进程的操作和子进程的操作分开。具体上面的代码 if pid 那一块是父进程的，else 那一块是子进程的。

线程入门

线程的内部实施细节其实比进程要更加复杂，一般通俗的说法就是线程是轻量级进程，这里不深入讨论具体线程的细节。

python 操作线程的主要模块是 **threading** 模块，简单的使用就是新建一个线程对象 (Thread)，然后调用 **start** 方法来启动它，具体线程要做些什么由本线程对象的 **run** 确定，你可以重定义它，如果是默认的就是调用本线程

Thread 类新建是输入的 **target** 参数，这个 target 参数具体指向某个函数。下面是一个简单的例子:

```
import random, threading

result = []

def randchar_number(i):
    number_list = list(range(48,58))
    coden = random.choice(number_list)
    result.append(chr(coden))
    print('thread:', i)

for i in range(8):
    t = threading.Thread(target = randchar_number, args=(i,))
    t.start()

print(''.join(result))
```

```
thread: 0
thread: 1
thread: 2
thread: 3
thread: 4
thread: 5
thread: 6
thread: 7
22972371
```

注意: 控制参数后面那个逗号必须加上。

我不太喜欢这种风格，因为线程对接的那个函数实际上并不能 return 什么值，而且其保存的值也依赖于前面的定义，并不能称之为真正意义上的函数（一个定义很好的函数必须复用特性很强）。所以线程还是如下类的风格编写。下面代码参考了 这个网页。

```
import random, threading

threads = []

class MyThread(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.result = ''
    def run(self):
        number_list = list(range(48,58))
        coden = random.choice(number_list)
        self.result = chr(coden)
    def getvalue(self):
        return self.result


for i in range(8):
    t = MyThread()
    t.start()
    t.join()
    threads.append(t)

result = ''
for t in threads:
    result += t.getvalue()
```

```
print(result)
```

```
05649040
>>>
```

上面调用线程对象的 **join** 方法是确保该线程执行完了，其也可能返回异常。上面的做法不太标准，更标准的做法是单独写一行 t.join 代码:

```
for t in threads:
    t.join()
```

来确保各个线程都执行完了，如之前的形式并不能达到多任务并行处理的效果。

上面的例子对线程的执行顺序没有特殊要求，如果有的话推荐使用 python 的 queue 模块，这里就略过了。

多线程：一个定时器

这个例子主要参考了这个网页。

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import time
import threading
```

```
class Timer(threading.Thread):
    def __init__(self,interval, action=lambda:print('\a')):
        threading.Thread.__init__(self)
        self.interval = interval
        self.action = action

    def run(self):
        time.sleep(self.interval)
        self.action()

    def set_interval(self,interval):
        self.interval = interval

#timer = Timer(5)
#timer.start()

class CountDownTimer(Timer):
    def run(self):
        counter = self.interval
        for sec in range(self.interval):
            print(counter)
            time.sleep(1.0)
            counter -= 1
        ####
        self.action()

#timer = CountDownTimer(5)
#timer.start()

def hello():
```

```
    print('hello\a')


timer = CountDownTimer(5, action = hello)
timer.start()
```

具体还是很简单的，这里之所以使用线程就是为了 timer.sleep 函数不冻结主程序。

多线程下载大文件

本小节参考了 这个网页 和 这个网页 。

下面的 `get_content_tofile` 函数在目标内容大小大于 1M 的时候将启动多线程下载方法。其中 `guess_url_filename` 函数是根据 url 来猜测可能的目标下载文件名字，还只是一个尝试版本。

注意下面使用 requests.get 函数的时候加上了 `stream=True` 参数，这样连接目标 url 的时候只是获得头文件信息而不会进一步下载 content 内容。这方便我们早期根据 headers 里面的信息做出一些判断。

接下来根据 HTTP 头文件的 `content-length` 来判断要下载内容的大小，如果没有这个属性，那么目标 url 是没有 content 内容的，本函数将不会对这一情况做出反应，这通常是单网页 url，使用 requests 的 get 方法获取网页文本内容即可。

然后如果目标长度小于 1M，那么就直接打开文件，使用 requests 模块里 response 对象的`iter_content` 方法来不断迭代完 content 内容。

如果目标长度大于 1M，则采用一种多线程下载方法。首先是`get_content_partly` 这个函数，接受 url 和 index，这个 index 是一个简单的索引，具体多少 bytes 后面还需要计算。关于多线程操作和具体多少 bytes 的计算细节这里略过讨论了。唯一值得一提的就是 HTTP 协议的 Range 属性，begin-end，对应具体的范围 0-1024，还包括 1024 位，所以实际上有 1025 个 bytes，为了获得和我

们 python 中一致的体验，我们让其 end 为 begin+1024-1。这样就有 1024 个 bytes 位，然后定位是 (0, 1024)，即和 python 中的一样，不包括 1024 位。

然后还有一个小信息是，HTTP 协议返回的头文件中的 **content-range** 属性，如果你请求 Range 越界了，那么将不会有这个属性。那么 begin 没有越界，end 越界的请求如何呢？HTTP 协议处理得很好，这种跨界情况都只返回最后那点 content 内容。

最后写文件那里降低内存消耗，使用了下面的语句来强制文件流写入文件中，好释放内存，否则你的下载程序内存使用率是剧增的。

```
f.flush()
os.fsync(f.fileno())
```

```
import re
def guess_url_filename(url):
    ''' 根据 url 来猜测可能的目标文件名, '''
    response = requests.get(url, stream=True)### 还有一个 content-type 信息可以利用
    s = urlsplit(url)
    guess_element = s.path.split('/')[-1]
    guess_pattern = re.compile(r'''
    (.png|.flv)
    $           # end of string
    ''', re.VERBOSE | re.IGNORECASE)

    if re.search(guess_pattern,guess_element):
        filename = guess_element
    else:
```

```
        filename = guess_element + '.html'
    return filename

import threading
import os
class DownloadThread(threading.Thread):
    def __init__(self, url,begin,chunk_size = 1024*300):
        threading.Thread.__init__(self)
        self.url = url
        self.begin = begin
        self.chunk_size = chunk_size
        self.result = b''
    def run(self):
        headers = {'Range':'bytes={begin}-{end}'.format(begin = str(self.begin),
            end = str(self.begin + self.chunk_size-1))}

        response = requests.get(url, stream=True, headers = headers)

        if response.headers.get('content-range') is None:
            self.result = 0### 表示已经越界了
        else:
            self.result = response.content
            print('start download...', self.begin/1024, 'KB')

    def getvalue(self):
        return self.result

def get_content_partly(url, index):
    threads = []
    content = b''
```

```
    chunk_size = 1024*300# 这个不能设置太大也不能设置太小
    block_size = 10*chunk_size# 具体线程数

    for i in range(10):
        t = DownloadThread(url, index * block_size + i*chunk_size )
        t.start()
        threads.append(t)

    for i,t in enumerate(threads):
        t.join()

    for t in threads:
        if  t.getvalue():
            content += t.getvalue()

    return content

import os
def get_content_tofile(url,filename = ''):
    ''' 简单的根据 url 获取 content，并将其存入内容存入某个文件中。
    如果某个内容 size 小于 1M 1000000 byte ，则采用多线程下载法'''

    if not filename:
        filename = guess_url_filename(url)

    # NOTE the stream=True parameter
    response = requests.get(url, stream=True)
    if not response.headers.get('content-length'):
        print('this url does not have a content .')
        return 0
```

```
    elif response.headers.get('content-length') < '1000000':
        with open(filename, 'wb') as f:
            for chunk in response.iter_content(chunk_size=1024):
                if chunk: # filter out keep-alive new chunks
                    f.write(chunk)
                    f.flush()
                    os.fsync(f.fileno())
    else:
        with open(filename, 'wb') as f:
            for i in range(1000000):###very huge
                content = get_content_partly(url, i)
                if content:
                    f.write(content)
                    f.flush()
                    os.fsync(f.fileno())
                else:
                    print('end...')
                    break
```

python 的元类编程

python 中的类和 class 自己定义的类和具体实例化的对象我们都了解了，而以上这些都是基于元类 (metaclass) 这个概念。

元类可以看作 python 中类的最底层的原子结构了，默认的元类叫做 **type**。元类创造类，类创造实例。所谓的元类编程就是..... 读者在使用元类编程之前需要评估好是不是需要使用这么底层的概念。

```
class C():
    pass
```

```
C = type('C',(),{})
```

附录

更多的 python 信息

1. 首先当然是官方各个模块的文档以及各第三方模块的官方参考 tutorial 或 reference。
2. github 上的各个 python 相关源码，这是我的 github 地址，其中有些项目很适合初学者学习。
3. 为了控制本文档的内容，很多我整理的一些额外的和 python 相关信息都放在这个网站的 **python** 部分。

linux 及时了解

1.

2. 我整理的有关 linux 系统上的一些知识，在这个网站的 **linux** 部分。

古怪的 python

本附录部分写了一些 python 中我认为很古怪的部分，因为古怪，所以常常被人忽视。

字符串比较大小

读者可以实验一下 python 中字符串之间是可以比较大小的：

```
>>> 'abc' > 'ab'
True
>>> 'fabc' > 'abc'
True
>>> '3.04' > '3'
True
```

这个特性有的时候很有用的，具体是如何比较大小的呢？按照 python 官方文档的描述，采用的是词典编纂顺序。具体描述信息如下：

> 序列之间比较大小是，首先两个序列各自的第一个元素开始比较，如果它们相同，则进行下一个比较，直到任何一个序列被穷尽。如果两个序列各自比较的类型都是相同的，那么整个过程将一直进行下去。如果两个序列是相等的则认为它们是相等的，如果某一个序列

是另外一个序列的子序列，则那个短的序列认为比长的序列要小。具体到每一个元素的大小比较，是按照 ASCII 顺序对其进行比较的。

中文比较大小?

读者这时会想到，既然 python 中字符串都默认是 unicode 编码（utf-8），那么中文应该也是能够比较大小的吧，事实确实如此:

```
>>> ’ 章’ > ’ 张’
True
>>> ’ 章’ < ’ 张’
False
>>> ord(’ 章’)
31456
>>> ord(’ 张’)
24352
```

感兴趣的读者可以打开字符映射表看一下，’ 张’ 对应的 unicode 编号是 U+5F20，你输入 0x5f20，返回的正是 24352。如果你输入 hex(24352)，返回的就是’0x5f20’。

ord 和 chr 函数

ord 函数接受一个字符，然后返回其 unicode 编码，十进制的。chr 函数是 ord 函数的反向，比如你输入 24352 这个十进制 uniocde，就返回了对应的字符。

```
>>> chr(24352)
’ 张’
```

所以我们可以总结到，python3 的字符串比较大小，是基于 utf-8 编码的。

其他

exec 和 eval

exec 和 eval 都可以用来执行 python 代码的字符串形式，exec 没有返回值，eval 有返回值。不过这两个函数使用都要慎重，按照 diveintopython3 第 8 章第九节的讲解，这些代码如果混入网络服务器中确实会很危险，如果一定要用，必须对输入字符串进行严格正则限定。

不过话虽然这样说，但这两个函数的使用有时能够给程序的架构带来意想不到的好处。比如说我编写的 youget 模块，其最核心的一个代码就是：

```
    def get_info(self):
        if self.netloc in netloc_id:
            target = netloc_id.get(self.netloc)
            print(' 在调用模块', target)
            exec('from youget.{0} import youget'.format(target), globals())
            self.info = youget(self.url)
        else:
            print(' 还不支持站点',self.netloc)


        return self.info
```

其对输入网站的 netloc 进行判断，然后选择某个 target 模块来 import youget 函数，exec 的第二个参数是一个变量设置，这里用 globals 函数继承自本模块的全局变量，这样就给全局变量 import 对应的 youget 函数了（具体细

节还需要进一步摸清。）这给这个模块带来了非常灵活的结构协同性。

在比如说我写过一个根据小型 python 代码生成 svg 文件的小模块，具体绘图的 python 代码类似下面的样子：

```
from pysvg.basicshapes import *
from pysvg.core import *

svg = Svg(width=XMAX * 2,height=YMAX * 2)
p0 = Point(0,0)
circle = Circle(p=p0, r=Quantity(2))
circle.set('fill',"red")
svg.add(circle)

p1 = Point(0,0)
p2 = Point(2,2)
line = Line(p1,p2)
svg.add(line)

rect = Rect(Point(-2,2),Point(2,-2))
svg.add(rect)

g1 = Group('g1',circle,rect)
g1.set('transform','translate(100)')
svg.add(g1)

print(svg)
```

这里不讨论那些类的具体细节，实际上很简单，就是编好 `__str__` 字符串输出控制函数。这里我们看到最后的那个 print 函数。然后字符串的输出流是用

下面这个核心代码控制的[①]：

```
    codeOut = StringIO()
    codeErr = StringIO()
    sys.stdout = codeOut
    sys.stderr = codeErr
    exec(code)
    sys.stdout = sys.__stdout__
    sys.stderr = sys.__stderr__
    content = codeOut.getvalue()
```

首先从 io 模块里面引入 StringIO 这个类，然后构建 codeOut，codeErr 这两个文本流对象，然后将 sys 默认的标准输出和错误输出对接成它们，然后使用 **exec** 函数执行代码即可，代码里面 print 的内容会进入文本流对象中，使用 getvalue 方法即可取出。最后重置了 sys 的默认的标准输出和错误输出流。

这里采用这样的 exec 方法很是暴力，但对于实现某些小的基于 python 的脚本方言还是很方便的。

如果执行 import 语句

参考了这个网页，如果在 exec 语句里面使用 import 语句，具体引入的变量名希望被外围程序使用，则需要如下所示。这里 globals() 返回当前全局变量值字典。

```
exec('from youget.{0} import youget'.format(target), globals())
```

① 参考了这个网页。

assert 语句

assert 语句简单的理解就是 `assert True` ，正常刷过去，而 `assert False` 将抛出 **AssertionError** 。

属性管理的函数

hasattr，setattr，getattr，delattr，这些函数都属于关于 python 中各个对象的属性管理函数，其都是内置函数。

其中 hasattr(object, name) 检测某个对象有没有某个属性。

setattr(object, name, value) 用于设置某个对象的某个属性为某个值，`setattr(x,a,3)` 对应 `x.a = 3` 这样的语法。

getattr(object, name[, default]) 用于取某个对象的某个属性的值，对应 `object.name` 这样的语法。

delattr(object,name) 用于删除某个对象的某个属性，对应`del object.name` 这样的语法。

参 考 文 献

[1] python 入门教程，python 官网上的 tutorial。原作者：Guido van Rossum Fred L. Drake ；中文翻译：刘鑫等；版本：2013-10-28；pdf 下载链接：python 入门教程 。

[2] learning python，主要 python 语言参考，我主要参看了 python 学习手册（第四版）。原作者：Mark Lutz，中文翻译：李军，刘红伟等。

[3] programming python，作者 Mark Lutz 对 python 编程的进阶讨论；版本：第四版。

[4] python官网上的资料。

[5] 第三方模块参考手册，如 numpy, scipy, matplotlib 等等第三方模块官网上发布的官方参考手册。

[6] zetcode 网站 pyqt4 部分，这是英文网站链接，这是 jimmykuu 的中文翻译网站。

[7] dive into python3，the english origin site , 这是中文网站链接。

[8] numpy beginner's guide book，the pdf download link is here .

[9] Introduction to Python for Econometrics,Statistics and Data Analysis，the pdf download link ishere.

[10] A Guide to Python's Magic Methods，作者：Rafe Kettler , 版本：2014-01-04，github 地址：here.

[11] metaprogramming，the website link is here .

[12] matplotlib tutorial，author: Nicolas P. Rougier ，the website is here.

[13] Foundations of Python Network Programming ，python 网络编程基础，[美] John Goerzen 著，莫迟等译。