Python Notes

Author : Yunqiu Xu

`Python`

- 基于python2.7.11

Chapter 0 安装Python

0.1 安装python及打开文件

- 对于windows系统, 需要添加路径`C:\Python27`到path
- ubuntu16.04自带python2.7
- 打开python: 直接在cmd输入python即可
- 使用python打开文件: python path

0.2 安装package

- 安装ez_setup + pip
- 安装包: pip install "xxxx.whl"

0.3 pycharm 安装及配置

- 激活
- 配置

Chapter 1 快速入门

1.1 基本结构和布局:

```
1.  #!/usr/bin/env python     起始行
```

```
#coding: utf-8            申明使用utf-8编码
#this is...               文档字符串
import modules            #模块导入
debug=True                #给出一系列全局变量
class(object)             #给出一系列类定义
def function()            #给出一系列函数定义
if __name__=="__main__"    #主程序
```

- 使用`__name__`指示module被加载的方式:
  - `if==True` : module被直接执行
  - 从另外一个py文件通过import导入该文件 : 不执行,`__name__`的值为模块名字
- 应用:调试代码
  - 外部模块调用时不执行测试代码,自己想排查问题时可以直接执行该测试文件

1.2 内存管理

- 变量无需事先声明
- 变量无需指定类型
- 不需关心内存管理
- 通过给全局/模块变量取新别名,能明显改善运行速度`r=random.randint()`

1.3 常用函数:

```
a,b=b,a #交换赋值
dir() #显示属性/内容
type() #显示类型
help() #帮助文档
obj.__doc__ #访问文档
obj.__file__#源代码位置
int() #转化为整形
str()/repr() #转化为字符串
len() #length
open(filename,mode) #打开文件,mode可以为"r/w/a"
range(start,stop,step) #[start,stop),间隔为step
raw_input("what you wanna input")
```

- 举个栗子

```
def fucker(x):
    '''fucker'''
fucker.__doc__ #...
```

Chapter 2 Python 对象

2.1 标准类型:

- object三要素 : ID+type+value
- 含有数据属性的对象:class/instance/module/complex/file
- 基本数据类型:
  - Integer
  - Boolean
  - Long integer
  - Floating point real number
  - Complex number
  - String
  - List
  - Tuple
  - Dictionary

2.2 操作符

- python多个比较操作可以在同一行进行,顺序从左到右
- `a is b`
  - 对象身份比较,等价于`id(a)==id(b)`
  - ==相等符与is等同并不一样,相等不一定等同,因为不一定为同一个对象
  - `id(item) #storage`
- python会缓存简单整形,但不会缓存浮点
- 注意不同电脑表现可能不同
- id是否相同主要取决于是否储存在同一块内存中
- 栗子1

```
2 is 2 #True
3 is 1+1+1 #True

a=10
b=10
a is b #True
#same memory

c=10.0
d=10.0
c is d #False
c == d #True
#different memory

e=[10]
  f=[10]
e is f #False
#different memory

id([1]),id([2]) #id一样,如果这两个不同行->连id都不一样
g,h=id([1]),id([2])
g is h #False
g == h #True
#存储位置相同,但仍旧不等同!
```

- 栗子2

```
a=20
a is 20 #True
b=257
b is 257 #alse
#different lines:<=256 True,>256 False!

c=20;c is 20 #True
d=280;d is 280 #True
#same line: True!
```

- 栗子3
  - memory s1 encodes X[0,0,0] then X[1,0,0];
  - memory s2 encodes X[0],X1,X2;
  - after changes,memory s3 encodes X[0],but X1/X2 are still in s2;

```
X = [0] * 3;
id(X)!=id(X[0])==id(X[1])==id(X[2])
X[0]=1 # X[0] changes,but X/X[1]/X[2]remain
```

- 栗子4
  - s1 encodes X[[0],[0],[0]],then X[1,1,1]
  - s2 encodes [0],then 1->all the X[i] not change

```
X=[[0]] * 3; #[[0],[0],[0]]
id(X)!=id(X[0])==id(X[1])==id(X[2])
x[0][0]=1#becomes [[1],[1],[1]]
#but all the ids remain unchanged!!
```

- 栗子5

```
X=[1,1,1]
#same to example 3
```

- 栗子6

```
X=[[0],[0],[0]]
#all the ids are different!
#s1,s2,s3,s4
X[0][0]=1-->[[1],[0],[0]]
#所有的id都不变->改变前和改变后存在一起
```

2.3 标准类型分类汇总

- number-标量-不可更改-直接访问
- string-标量-不可更改-顺序访问
- list-容器-可更改-顺序访问
- tuple-容器-不可更改-顺序访问
- dict-容器-可更改-映射访问
  - 无序不会造成影响

```
myStr='fuck'
myStr[1]='U' #Error
```

Chapter 3 Number

- 这里只标注一些原来没见过的

3.1 位操作符:只适用于integer,转化为二进制后运算

```
~num #按位取反,二进制+1后乘以-1
~5 #-(101+1)=-110=-6
num1<<num2 #num1左移num2位,等同于2**num2
10<<1 #1010->10100->20
20>>2 #10100->101->5
num1&num2 #按位与
5&3 #101&11=1
10&8 #1010&1000=1000=8
num1|num2 #按位或
5|3 #101|11=111=7
10|8 #1010|1000=1010=10
num1^num2 #按位异或,对位相加不进位
5^3 #101+11=110=6
10^8 #1010+1000=10=2
```

3.2 数字类型函数

```
int()/long()/float()/complex() #类型转换
abs()/pow() #功能函数

#商用/,余数用%
round() #四舍五入
hex()/oct() #整形转化为十六/八进制
chr()/ord() #ASCII/unicode数字转成ASCII/unicode字符；字符转成值
chr(70)="F"
ord("F")=70
```

- 注意一些python3与python2不同的地方

```
17/3 #python3:5.666667;python2:5
(50 - 5*6) / 4 # 5.0;5
```

Chapter 4 String/List/Tuple

4.1 通用序列操作

4.1.1 索引

```
x[0]
'Hello'[1] #返回"e",索引的一种
```

4.1.2 分片slice

```
numb=[0,1,2,3,4,5]
num[3:5] #返回3,4,相当于[3,5) ， 等价于num[-3:-1]
#注意,如果索引左侧在索引右侧后面出现,则返回[]
```

- 栗子:`num[-3:0]` -> `[]`

```
num[-3:] #得到后三个元素
num[:3] #得到前三个元素
num[:] #得到所有元素
num[a:b:c] #[a,b),间隔为c
    num[::3] #提取每三个元素的第一个
    num[::-2] #逆序提取每两个元素中的一个
```

4.1.3 序列的加法与乘法

```
list1+list2 #不是元素相加,而是合成一个新list,里面有list1、2的全部元素
```

- 不同数据类型的list不可相加

```
"python" *3 #输出"pythonpythonpython"
[42]*3 #输出[42,42,42]
[None]*3 #[None,None,None]
```

4.1.4 成员资格/长度/极值

```
"xxx" in object #True/False
len() #length
max()/min()
```

4.2 Strings

- 字符串为不可变tuple

4.2.1 格式化字符串

- 格式化为%s,句中若有%需要改成%%
- `%.nf` n位浮点数
- 其他转化类型:
  - `d.i` 带符号的十进制整数
  - `o/u/x/X` 不带符号的八/十/十六小写/十六大写
  - `e/E` 科学计数法浮点数小写/大写
  - `f/F` 十进制浮点数
  - `r/s` 字符串（前者为repr/后者为str)
- 先宽度后精度,几个栗子:
  - `"%10.2f"%pi` 输出"_____3.14",宽度10,精度2,用空格补位
  - `"%010.2f"%pi` 输出"0000003.14",宽度为10,用0补位
  - `"%-10.2f"%pi` -表示左对齐,输出3.14
  - `"% 5d"%10` 空格表示用空格来补位,可以用于对齐正负数
  - `"%+5d"%10` 另一种对齐方法,不管正负都带符号,输出+10

4.2.2 .format() 格式化函数,比 %()更符合代码风格

- via position

```
'{0},{1}'.format('kzc',18) #'kzc,18'
'{},{}'.format('kzc',18)  #'kzc,18'
'{1},{0},{1}'.format('kzc',18)  #'18,kzc,18'
```

- via keyword arguments

```
'{name},{age}'.format(age=18,name='kzc')  #'kzc,18'
```

- format:`^,<,>`分别是居中、左对齐、右对齐, 后面带宽度

```
'{:>8}'.format('189') #space
'{:0>8}'.format('189') #'00000189'
'{:a>8}'.format('189') #'aaaaa189'
```

```
'{:.2f}'.format(321.33345)
'{:d}'.format(17)#int
'{1:b},{0:}'.format(3.14,3)#11,3.14
```

4.2.3 字符串内建函数

```
x.find("y") #输出x中最先出现y的位置,找不到返回-1
x.title()/upper()/lower() #每个词大写,全字母大小写
x.replace("a","b") #将x中a全都换成b
x.split("分隔符") #没有分隔符默认为空格
' '.join(x) #与split互逆
```

4.2.4 Unicode

- encode编码:将默认的unicode码转换为其他格式
- 将字符串直接编码为unicode`u"string"`

```
text=u"女友" #输入text输出u'\u5973\u53cb'
text.encode("GBK") #unicode->GBK
```

- decode解码:将非unicode的字符串转化为unicode:
  - `s.decode("prevtype")`等价于`unicode(s,"prevtype")`
- 栗子:

```
#utf-8 --> unicode:
'aaa'.decode('utf-8') #等价于unicode('aaa', 'utf8')

#GBK --> unicode:
"傻逼".decode("gbk") #输出u'\u50bb\u903c'
```

- 在正则匹配时,只有pattern和content均为unicode才能匹配

4.3 List

4.3.1 内建函数

```
list("hello") #['h','e','l','l','o']
x[1]=2 #赋值,不能给一个位置不存在的元素进行赋值
name[2:]=list("fuck") #将"fuck"slice为['f','u','c','k']赋值给name[2:]
#分片赋值可以长度不同
```

```
x.append('xxx') #结尾追加一个元素(即使是list也被认为是一个元素)
    x.append(range(3))-->[xxx,[0,1,2]]
x.extend("xxx") #结尾追加序列(可以为多个元素)
    x.extend(range(3))-->[xxx,0,1,2]
x+y #也是追加序列
    #extend是修改原列表," + " 是生成新列表

x.pop() #结尾删除,括号中有位数则为定位删除
    #既能修改列表又能返回被删除的元素值
    #这个有点类似先入先出队列
del x[6] #删除
x.remove(y) #删除元素y而非位置
x.insert(y,' ') #insert
    #注意,若要先进先出,使用insert而非append
x.count(number/"character") #count
x.index(y) #在listx中找出匹配项y的位置
x.reverse() #反向存放,改变列表,不返回值
x.sort(key=None,reverse=False) #排序,改变列表,不返回值
```

- 举个栗子:

```
y=x[:] #这时y.sort(),x不变
y=x #这时y.sort(),x也被排序,如果不需要保留x的话,用这个函数
y=sorted(x) #y排序,不改变x
sorted("a B c d ".split(), #按空格分成list
        key=str.lower, #排序时按小写排
        reverse=True) #逆向,输出['d','c','B','a']
```

- 这里再说下reversed/sorted(x)和x.reverse()/x.sort()的区别:
  - 前者返回的是新对象,而后者直接改变的是原函数的值
- 可以结合后面浅复制/深复制/赋值进行理解

4.3.2 list的特殊属性

- Stack(LIFO)后进先出
- Queue(FIFO)先进先出

4.3.3 Copy:shallow/deep

- 浅复制 : 只拷贝父对象,不会拷贝对象的内部的子对象(子对象相当于赋值)
- 赋值 : 相当于引用 , 改变一个另一个也会变化
- 深复制 : 完全复制 , 无论其中一个怎样变化另一个都不变

```
from copy import copy,deepcopy
a=[1,2,3,[4,5]]
b=copy(a) #浅复制 , equal to b=a[:]
c=a # 赋值
d=deepcopy(a) #copy.deepcopy
e=list(a) #新建->相当于浅复制
b[0]=100 #a unchanged
b[3][0]=100 #a changed
d[3][3]=100 #a unchanged
e[3][4]=100 #a changed
```

4.4 元组Tuple

- 跟list相比tuple是不可变的,用()替代[]
  - 一旦定义,不能被更新或删除
  - 因为不可变,tuple 可以用作dict的key
- `(42,)`一个值的元组必须加逗号,否则会被误认为是数字运算
- `tuple(list)`将一个list 转化成tuple
- 更新元组:将原有元组和新元素相加，相当于生成了个新元组
- 删除元组:不能删除单个元素(算作更新)，只能删除整个tuple

Chapter 5 Dict and Set

5.1 dict基本结构

- `dict={"a":"123","b":"456","c":"789"}`创建时也可以为空字典{}
- 字典由多个键值对(Key,value)组成
- Key是唯一的但value不唯一
- Key不一定为整形,但一定不是list可变列表(key is hashable,不可变)

5.2 dict函数

```
d=dict(items) #通过其他映射建立dict
d=dict(name="XYQ",age=42) #关键字参数,d={"age":42,"name":"XYQ"}
{}.fromkeys(key,key,key) #使用给定的key建立新字典,默认值为None
```

```
len(d) #查询k-v对的数量
d[k] #查询key k的value
d[k]=v #赋值（可以往空白字典里赋值）
del d[k] #删除
k in d #检查是否存在
d.clear()#清除字典d里所有项,原始字典也会清空,而不是赋值{}
x=d.copy()
    #similar to 4.3.3
    a={'x':1,'y':2,'z':[3,4]}
    b=a
    c=a.copy()
    a['x']=100#b变化,c不变
    a['z'][0]='fuck'#bc都变化

d.get(key) #访问某个key
    #与直接访问不同的是,如果key不存在,get()返回none而不是出错
d.items() #以列表形式返回字典项（key+value）
d.iteritems() #返回迭代器对象
d.keys() #以列表形式返回字典key
d.iterkeys() #返回迭代器对象
d.values() #以列表形式返回字典value,可以包含重复对象
d.pop(key) #获得给定键的值然后将这个项目移除
d.update(x) #用新字典x更新字典d
    dict1.update(dict2)
    dict2.update({sb:"shabi"})
```

5.3 Set

```
set()/frozenset() #可变/不可变集合
in/not in #跟前面一样
s.add("x")/s.remove("x")/s.update("x") #添加/删除/更新,del s为删除整个集合
```

- 集合类型操作符:
  - `|`并集
  - `&`交集
  - `a-b`差补,只属于a而不属于b,相当于减法
  - `a^b`异或,只属于a/b而非共有
- set中所有的元素都是唯一的

```
num=set([1,1,2,2,3,3,4,4])
```

```
>>>num
>>>set([1,2,3,4])
```

Chapter 6 条件循环及其他语句

6.1 基本语句

- `if/elif/else`比较运算符
- `while/for`循环运算符
- `while True` 无限循环,如果不出错这个程序会一直进行,想跳出ctrl+c即可
- `break/continue/pass`
  - `break`直接结束整个循环
  - `continue`结束本轮循环, 进入下一轮循环
  - `pass`什么都不做继续下一个语句
- precedence : not>and>or

```
not x or y or not y and x #(not x) or y or ((not y) and x)
```

6.2 列表推导式List comprehension

- 核心为for循环
- 列表推导式为轻量级循环,不是语句,而是看起来类似循环的表达式

```
[x*x for x in range(10) if x%3==0] #输出[0,9,36,81]
[(x+1,y+1) for x in range(3) for y in range(5)] #三行五列矩阵
```

- print语句不能用作列表推导

6.3 iterator迭代器

- java版本

```
if(hasNext()==True) next();
```

- 迭代规则:　给定list/tuple, 使用for语句进行遍历
- 迭代器:可以被`__next__()/next()`调用并不断返回下一个值的object
- 迭代器与列表的差别 : 计算一个值取一个值,而不是一次性得到所有值,防止占用太多内存
- 一个实现了`__iter__`方法的对象是iterable的,而一个实现了`__next__()/next()`方法的对象称为迭代器
- 获得iterator:

```
it=iter([1,2,3])
it.next() #output 1
it.next() #output 2

for item in it:
    print item
```

6.4 generator一边循环一边计算

- 列表生成式的不足 : 需要生成所有数据以创建list,而生成器可以对其进行改善
- 生成器与列表生成式格式类似,但是[]变为(),每次调用next函数给出下一个value

6.4.1 简单生成器:基于for循环

```
g=(x*x for x in range(10))
next(g) #0 ,equal to g.next()
next(g) #1
next(g) #4
#循环到头继续next返回错误
```

- 使用for循环避免多次调用next()

```
for item in g:
    print item
```

$#6.4.2 对于比较复杂的generator,通过定义函数来实现

- 将print改为yield
- yield仅能在定义函数内部使用
- 栗子:使用generator实现斐波拉契

```
def fib(max):
    n=0
    a=0
    b=1
    while n<max :
        yield(b)
        a=b
        b=a+b
        n=n+1
    print "done!"
#试运行
f=fib(5)
f #输出生成器标识
f.next() #1,可以一直用然后用完了抛出error
```

Chapter 7 文件的写入和读取

7.1 文件内建函数

```
file_object=open(file_name,access_mode="r",buffering=-1)
```

- `file()`和`open()`可以任意替换,一般读写时用`open()`,而处理文件对象时用`file()`
- mode参数:

```
r #只读,需要文件已存在
rU#只读,通用换行符支持,将所有换行符转化为python格式
w #写入,清空原文件并重新创建内容
a #结尾追加,文件不存在则自动创建
r+/w+/a+ #都是以读写模式打开
#若以二进制形式操作,前面的mode后面加"b",e.g."rb/wb"
buffering=-1 #默认从头读到尾
```

7.2 文件内建方法

7.2.1 读取

```
read() #读取给定数目字节,默认size=-1读取全部内容
readline() #读取整行
```

```
readlines() #读取全部内容并返回为一行
```

- 栗子：读取文件并将元素加入list

```
path="~/home/shabi.py"
myFile=open(path,"r")
myList=[]
for line in myFile.readlines():
    myLine=line.strip().split()
    myList.append(myLine)

>>> [[],[],...[]]
```

7.2.2 写入

```
write() #刚好与read()相反
writelines() #将列表以一行的形式写入文件，注意需要自行添加换行符
```

- 不存在`writeline()`,等价于单行字符串加上换行符后调用`write()`
- 注意读取时不会删除换行符,写入时不会自带换行符,需要自己进行操作

7.2.3 文件内移动

```
seek() #文件内移动指针到某个位置
text() #显示指针在文件内的位置
```

7.2.4 文件迭代

- 逐行访问文件

```
for eachline in f:
    :
```

7.2.5 关闭文件

```
close()
```

7.3 文件内建属性

```
file.closed #True则文件已被关闭
file.encoding #给出文件编码
```

```
file.mode #给出文件打开时的mode
file.name #文件名
```

7.4 命令行参数 `sys module`

```
argv #argument vector命令行参数,包括文件名称
exit() #退出当前程序
path #查询路径
stdin/stdout/stderr #standard input/output/error flow,给出file-like 对象
```

- 举个栗子:

```
import sys
print "you entered",len(sys.argv),"arguments.."
print "they are ",str(sys.argv)

argv.py 76 tales 85 hawk #输出:
you entered 5 arguments...
they are: ['argv.py','76','tales','85','hawk']
```

7.5 文件系统 `os module`

7.5.1 文件处理

```
mkfifo() #创建命名管道
remove() #删除文件
rename() #重命名
```

7.5.2 目录/文件夹

```
chdir() #change the directory
chroot() #change root 根目录
listdir() #list the directory
getcwd() #get current work directory
mkdir() #make directory
rmdir() #remove directory
```

7.5.3 访问/权限

```
access() #检查权限
```

```
chmod() #change the mode
chown() #改变owner
umask() #set default mode
```

7.5.4 文件描述符操作

```
open() #底层的系统open
read()/write()
dup()/dup2() #duplication/功能相同,但是复制到另一个文件描述符
```

7.5.5 针对系统的操作

```
environ #对环境变量进行映射
system(command) #在子shell中执行操作系统命令
sep #路径中的分隔符
pathsep #分割路径的分隔符
linesep #行分隔符("\n","\r","\r\n")
urandom(n) #返回n个自己的加密强随机数据
```

Chapter 8 错误和异常

8.1 Type of errors

```
#10 * (1/0)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero

#4 + spam*3
Traceback (most recent call last):
File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined

#'2' + 2
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: Cannot convert 'int' object to str implicitly
```

- other errors
  - AssertionError

- AttributeError
- SyntaxError
- ImportError
- IndexError/KeyError/ValueError
- RuntimeError
- UnicodeEncodeError/UnicodeDecodeError
- IOError

8.2 检测异常

```
while True:
    try:
        f=open("sb","r")
    except IOError,e:
        print "could not open file:",e
```

- `try/except/else`如果未出现异常,则执行else

8.2 多个except

- 处理一个try中可能存在的多种异常

```
try:
    xxx
except Exception1:
    yyy
except Exception2:
    zzz
```

8.3 处理多个异常的except语句

```
try:
    except(Exception1,Exception2):
    #空except语句默认捕获所有异常
```

8.4 上下文管理`with`

- 相比`try/except`，`with`可用于简化代码

```
with open("filename","r") as f: #打开文件,如果一切正常,进行赋值
    for eachLine in f: #迭代器遍历每一行
        do ...
#如果这段代码任意一个部分发生异常,执行清理代码并自动关闭文件
```

8.5 触发异常`raise`

```
raise[someException[,args[,traceback]]]
```

8.6 `assert`语句以及异常`AssertionError`

- 断言:要求某些条件必须满足,否则直接让程序崩溃

```
age=-1
assert 0<age<10 #如果True则pass,否则返回AssertionError
```

- 断言调试 : 在可能出错的地方写入断言,如果没通过则证明此处有错

```
def foo(s):
    n = int(s)
    assert n != 0, 'n is zero!'
    return 10 / n

def main():
    foo('0') #raise error
```

- 调试完毕使用`-0`关闭断言:此时`assert`相当于`pass`

```
python -0 xxx.py
```

8.7 Test

```
def abs(n):
    '''
```

```
    Function to get absolute value of number.

    Example:

    >>> abs(1)
    1
    >>> abs(-1)
    1
    >>> abs(0)
    0
    '''
    return n if n >= 0 else (-n)

    if __name__=='__main__':
        import doctest
        doctest.testmod()
#如果没有输出内容说明通过测试
```

Chapter 9 函数和函数式编程

9.1 调用函数

9.1.1 关键字参数

- 明确每个参数的作用,如果未改变,使用该默认参数,可调换顺序

9.1.2 收集参数

- 参数前加*,给函数提供任意个参数:

```
def printaa(*params): # *指代收集其余的位置参数
    print params

printaa(3,4,5)
>>> (3,4,5)
printaa(1,2,*(3,4,5))
>>>(1,2,3,4,5)
```

9.1.3 收集关键字参数

- 参数前加**

```
def pp(x,y,z=3,*pospar,**keypar): #星号只有在定义/调用函数时才有用
```

```
    print x,y,z
    print pospar
    print keypar

print pp(1,2,3,5,6,7,foo=1,bar=2)
#x=1,y=2,z=3
#5,6,7 被归纳于非关键字参数内,使用*pospar进行收集
#foo=1,bar=2被归纳于关键字参数内,使用**keypar进行收集
>>>1 2 3 {5,6,7} {"foo":1,"bar":2}
```

9.2 前向引用

- python中声明和定义时一体的

```
def function1():
    print 'function1'
    function2()
#直接调用会失败,因为还没有声明function2()

def function1():
    print 'function1'
    function2()
def fucntion2():
    print 'function2'
#此时调用function1()会成功,因为function2()已经在被调用前被声明`了
```

9.3 传递函数

- 通过将函数A赋值给函数B,这两个函数引用了同一个函数object

```
def function1():
    print 'functions'
function2=function1
function2()
>>>'functions'
```

9.4 函数式编程

- 通过几种内建函数和lambda函数构建

9.4.1 lambda匿名函数

- 省却重新定义函数及冗杂的参数
- 栗子1:

```
f=lambda x,y,z=2:x+y+z #f(1,2,3)输出6

#等价于:
def fun(x,y,z=2):
    return x+y+z
```

- 栗子2:

```
def action(x):
    return lambda y:x+2*y
a=action(2) #这个参数是x
a(22)
>>>46
#这里x已经被提供了,算常数,y=22是参数,返回46
```

- 栗子3:

```
b=lambda x:lambda y :x+2*y #嵌套函数
a=b(3)
a(2)
>>>7
(b(2))(1)
>>>4
```

9.4.2 map

- 2.7可以直接使用,但3.4需要`list(map(…))`,否则得到的是迭代器

```
map(function, sequence[, sequence, ...])
```

- 参数序列中的每一个元素调用function函数
- 返回包含每次function函数返回值的list

```
map(lambda x: x ** 2, [1, 2, 3, 4, 5])  #[1,4,9,16,25]
```

- 在参数存在多个序列时,会依次以每个序列中相同位置的元素做参数调用function

```
#对两个序列中的函数依次求和
map(lambda x, y: x + y, [1, 3, 5, 7, 9], [2, 4, 6, 8, 10])
>>>[3,7,11,15,19]
```

- 注意function函数的参数数量,要和map中提供的集合数量相匹配．如果集合长度不相等,会以最小长度对所有集合进行截取．

```
map(None, [1, 3, 5, 7, 9], [2, 4, 6, 8, 10])
>>> [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
```

9.4.3 filter

- 2.7可以直接使用,但3.4需要`list(filter(…))`,否则得到的是迭代器

```
filter(function or None, sequence) #Output : list, tuple, or string
```

- 定义一个布尔函数作为function参数,用于过滤
- 对sequence中每个元素调用function进行过滤,然后返回结果为True的元素
- 如果函数为None,就返回完整的sequence

```
#返回所有偶数
def is_even(x):
    return x & 1 != 0
filter(is_even, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>>[1,3,5,7,9]
```

9.4.4 reduce `from functools import reduce`

- 对参数序列中的元素进行迭代

```
reduce(function, sequence, initial)
```

- function:双参数函数,不能为None；
- reduce依次从sequence中取一个元素,和上一次调用function的结果做参数再次调用function；
- 如果提供initial参数,第一次调用initial以及sequence[0]作为参数；
- 如果不提供initial参数,调用sequence[0,1]作为参数；

```
#累加
```

```
reduce(lambda x, y: x + y, [2, 3, 4, 5, 6], 1)
#(((((1+2)+3)+4)+5)+6)=21
```

- 再举个栗子:

```
reduce(lambda x,y:x*2+y,[1,2,3,4,5])
#1*2+2>>4*2+3>>11*2+4>>26*2+5=57

reduce(lambda x,y:x+y*2,[1,2,3,4,5])
#1+2*2>>5+3*2>>11+4*2>>19+5*2=29
```

9.5 偏函数partial function

- 给某个参数设定默认值,返回新函数,简化调用过程

```
#转化为二进制
from functools import partial
int2=partial(int,base=2)

#等价于
def int2(x,base=2):
    return int(x,base=2)
```

- 再举一个栗子:

```
max10=partial(max,10)
max10(1,2,3) #max(1,2,3,10),返回10
max10(8,12) #max(8,10,12),返回12
```

9.6 变量作用域

- 全局变量:在全局环境下定义的变量
- 局部变量:在定义函数的环境中定义的变量,且仅仅在函数中起作用

9.6.1 python作用域规则:

- 先在函数中寻找局部变量,找不到在全局中寻找
- 局部与全局同名,在调用函数时会优先局部,但全局的值不变

```
x=1
```

```
def func1():
    x=2
    return x
x #调用全局变量,返回1
func1() #优先局部变量,返回2
x #全局变量值不变,返回1
```

- global语句:重新定义全局变量

```
x=1
def func2():
    global x=2 #此处x为全局变量,并将x值更新为2
    return x
x #全局变量,返回2
func2() #返回2
```

9.6.2 闭包

- 如果一个函数引用了在其外部作用域（不一定是全局）的变量,那么该函数就被认为是闭包(closure)
- 一个函数和其所处的环境构成了一个闭包
- 闭包有效减少需要定义的函数数目,增加可移植性
- 栗子1:

```
def line_conf():
    b = 15
    def line(x):
        return 2*x+b #line()和line_conf()中的b构成闭包
    return line

b=5
my_line=line_conf
my_line(5) #返回5*2+15=25
```

- 栗子2:

```
def line_conf(a,b):
    def line(x):
        return a*x+b
    return line

my_line1=line_conf(1,2)
```

```
my_line2=line_conf(3,4)
my_line1(5) #7
my_line2(5) #19
```

9.6.3 比较复杂的栗子

```
j=1
k=2
def func1():
    j=3
    k=4
    print "j==%d and k==%d"%(j,k)
    k=5

def func2():
    j=6
    func1() #此处j=6 与func1() 构成闭包
    print "j==%d and k==%d"%(j,k)

k=7 #全局变量更新:k=7
func1() #Output:j==3 and k==4,引用局部变量
print "j==%d and k==%d"%(j,k) #Output:j==1 and k==7,引用全局变量

j=8 #全局变量更新,j=8
func2()
#先输出func1()的结果:j==3 and k==4
#再输出后面的print语句:j==6 and k==7,func2里未指定局部k,调用全局的k
print "j==%d and k==%d"%(j,k) #全局变量j==8 and k==7
```

Chapter 10 Modules

10.1 创建并调用module

10.1.1 创建

```
import sys
def text():
    args=sys.argv #list储存CMD的所有参数
    #python hello.py获得的sys.argv=["hello.py"]
    #python hello.py XYQ获得的sys.argv=["hello.py","XYQ"]
    if len(args)==1:#仅仅有一个xxx.py,没有名字
        print "hello,bitch!"
```

```
    elif len(args)==2:#xxx.py name
        print "hello,bitch!"%args[1]
    else:
        print "too many elements"
if __name__=="__main__":
    text()
```

10.1.2 调用模块

```
import sys
sys.path.append("c:/xyqcoding") #添加path
import module_name
module_name.text() #输出hello,bitch！
```

10.1.3 查找模块

```
sys.path.append() #返回module位置
```

- 查找代码:`sys+pprint`

```
import sys,pprint
pprint.pprint(sys.path) #给出代码存储及调用的path
```

- site-packages文件夹用于存放第三方模块,之后安装的模块在此寻找

10.2 常用自带库

10.2.1 sys/os

10.2.2 fileinput

- 遍历文本文件的所有行

```
input([files[,inplace[,backup]]) #遍历多个input flow中的row
filename() #返回当前的filename
lineno() #返回当前累计的行数
filelineno() #返回当前文件的行数
isfirstline() #检查当前行是否是文件的第一行
isstdin() #检查最后一行是否来自sys.stdin
```

10.2.3 set/heap/Double-ended queue

```
set(range(10)) #输出{1:10}
```

- Double-ended queue双端队列:按照元素增加的顺序移除元素
- heap堆为一种优先队列,任意顺序增加对象,且可以于任意时间找到并移除最小对象
- heapq module:

```
heappush(heap,x) #add x into heap
heappop(heap) #extract the smallest element in the heap
heapify(heap) #强制应用heap属性到任意一个list
heapreplace(heap,x) #弹出最小元素,x入堆,相当于push+pop
nlargest/nsmallest(n,iter) #返回iter中第n大/小的元素
```

10.2.4 time

```
asctime([tuple]) #将time tuple 转换为字符串
localtime([secs]) #将秒数转换为date tuple
mktime(tuple) #将时间元组转换为本地时间
sleep(secs) #休眠secs秒
strptime(string[,format]) #将字符串解析为时间元组
ctime() #current time
```

10.2.5 random

```
random() #[0,1)
getrandbits(n) #以长整型形式返回n个随机位
uniform(a,b) #[a,b)
randint(a,b) #[a,b]任意整数
randrange(start,stop,step) #返回range(start,stop,step)中的随机数
choice(seq) #从list seq中返回随机元素
sample(seq,n) #从list seq中选择n个随机独立元素
shuffle(x) #打乱list x的次序(原位)
```

10.2.6 re 参考正则表达式部分

Chapter 11 Object Oriented Programming

- an example from UNSW COMP9021

```
class Fraction():
    def __init__(self, *args): #初始化,病判断输入参数类型
```

```
        try:
            if len(args) != 2:
                raise TypeError
            if not self._validate(*args):
                raise ValueError
        except TypeError:
            print('Provide exactly two arguments.')
        except ValueError:
            print('Provide an integer and a nonzero integer as argument
s.')
        else:
            self._set_to_normal_form(*args)

    def _validate(self, numerator, denominator):
        if not isinstance(numerator, int):
            return False
        if not isinstance(denominator, int) or not denominator:
            return False
        return True

    def _set_to_normal_form(self, numerator, denominator):
        sign = -1 if numerator * denominator < 0 else 1
        numerator = abs(numerator)
        denominator = abs(denominator)
        gcd = self._gcd(numerator, denominator)
        self.numerator = sign * numerator // gcd
        self.denominator = denominator // gcd

    def _gcd(self, a, b): #最大公约数,用于约分病计算之后的分数加减法
        if b == 0:
            return a
        return self._gcd(b, a % b)
```

- python magic methods

```
def __repr__(self):
    #特殊函数,返回特定值
    #e.g. xxx=Fraction()-->输入对象名xxx,相当于调用这个函数
        return 'Fraction(numerator = {!r}, denominator = {!r})'.format(
self.numerator, self.denominator)

def __str__(self):
    #特殊函数,输入print(xxx)相当于调用这个函数
    return '{!s} / {!s}'.format(self.numerator, self.denominator)
```

```
9.
10.   def __add__(self, fraction):
11.       #这里的fraction参数相当于另一个属于Fraction类的对象
12.       return Fraction(self.numerator * fraction.denominator + self.denomi
      nator * fraction.numerator, self.denominator * fraction.denominator)
13.
14.   def __sub__(self, fraction):
15.           return Fraction(self.numerator * fraction.denominator - self.de
      nominator * fraction.numerator, self.denominator *
      fraction.denominator)
16.
17.   def __mul__(self, fraction):
18.       return Fraction(self.numerator * fraction.numerator,
      self.denominator * fraction.denominator)
19.
20.   def __truediv__(self, fraction):
21.       try:
22.           if fraction.numerator == 0:
23.               raise ValueError
24.       except ValueError:
25.           print('Cannot divide by 0.')
26.           return
27.       return Fraction(self.numerator * fraction.denominator,  self.denomi
      nator * fraction.numerator)
28.
29.   def __lt__(self, fraction):
30.       return (self.numerator * fraction.denominator < self.denominator *
      fraction.numerator)
31.
32.   def __le__(self, fraction):
33.       return (self.numerator * fraction.denominator <= self.denominator *
      fraction.numerator)
34.
35.   def __gt__(self, fraction):
36.       return (self.numerator * fraction.denominator > self.denominator *
      fraction.numerator)
37.
38.   def __ge__(self, fraction):
39.       return (self.numerator * fraction.denominator >= self.denominator *
      fraction.numerator)
40.
41.   def __eq__(self, fraction):
42.       return (self.numerator * fraction.denominator == self.denominator *
      fraction.numerator)
43.
```

```
    def __ne__(self, fraction):
        return (self.numerator * fraction.denominator !=
    self.denominator * fraction.numerator)
```

11.1 Basic Knowledge

- object : 基本单元,包含property(变量) + method(储存在对象内的function)
- class : object的集合,其任务为定义instances需要用到的方法
- instance : class内的一个实例,如a=Class(),a就是Class的一个instance
- property : object中的variable
  - 制造机器相当于class
  - 产品为各个class的instance
  - 颜色和尺寸相当于property
- method:object中的function
  - 调用method的过程:
    - 定义class,并在class中创建method;
    - 创建一个属于该class的instance;
    - 使用该instance调用method
  - self参数 : method必须有一个额外的第一个参数self,代表object本身
  - 可以将第一个参数绑定在所属实例/普通函数上——不需要再提供self参数

```
class Person(): #创建class(类定义,未指明则默认为object)
    def __init__(self,name,age): #定义构造器
        self.name=name #设置name
        self.age=age #设置age

    def getName(self,name): #method1
        return self.name

    def greet(self): #method2
        print "I am %s"%self.name

    def getAge(self,age): #method3
        return self.age

    def ag(self): #method4
        print 'I am %s'%self.age

a=Person("XYQ","18") #创建一个属于类Person()的instance,并传入参数
```

```
a.name #访问instance的property
a.greet() #调用方法
a.name="fuck" #外部访问函数,如果不允许外部访问,可以在定义时私有化
```

11.2 OOP

11.2.1 基本术语:

- 抽象/实现 : 对现实世界问题进行建模,建立相关子集,描绘程序结构,从而实现该模型
- Encapsulation封装/接口
  - 对数据提供接口及访问函数,从而对信息进行隐藏;
  - 即在class内部定义函数,对全局作用域其他区域隐藏信息;
- 特性存储attribute storage:将名字封装在对象内;
- 举个栗子: 在class内部定义函数setName&getName

```
o=OpenObject()
o.setName("xyq") #调用已被封装的函数
o.getName() #调用已被封装的函数
```

- Inheritance继承
  - 以普通的类为基础建立专门的类对象;
  - 此时原类变为新类的超类/父类/超类;
  - 新类为子类subclass,拥有超类的全部功能,且可以添加功能/改进;
  - 某个instance属于子类一定也属于父类,反之不成立;
  - 继承中的多态:如果instance属于父类,可直接调用父类的method而不需要考虑其subclass
- Polymorphism多态:
  - 无需了解对象类型,对不同类对象使用同样的操作;
  - 根据对象类型不同,表现出不同的行为;
  - 举个栗子:`repr(x)/str(x)`可以对任何类型使用

```
def length_msg(x):
        print "the length of",repr(x),"is",len(x)
```

11.2.2 设计思想:抽象出class,根据class创建instance

11.3 Class

11.3.1 创建类

```
class A():
    def thing(self,a,b):
        print "they are %s and %s"%(a,b)
    def food(self,c,d):
        self.thing(c,d)
    def drink(self,e):
        print "it is %s"%e
```

11.3.2 私有化inaccaccible:让方法或者特性从外部无法访问

- python不直接支持私有:在定义函数名前加两个下划线

```
class Funk():
    def __inaccessible(self):
        print u"私有函数,见者死妈"
    def accessible(self):
        self.__inaccessible()
s=Funk()
s.__inaccessible() #直接调用私有化的函数返回错误
s.accessible() #非私有化的函数可以调用
```

11.4 Inheritance继承:

- 举个栗子:

```
class Animal():
    def run(self):
        print u"动物"

class Dog(Animal): #Dog为Animal的子类,Animal为Dog的父类
    pass#子类继承了父类的全部功能

dog=Dog()
dog.run() #输出"会跑"

class Bird(Animal):
    def fly(self):
        print u"会飞" #子类可以对父类进行修改更新

bird=Bird()
```

```
bird.fly() #输出"会飞"
bird.run() #输出"会跑" ,继承中的多态
```

- 再举个栗子:多重继承

```
#原始分类
class Animal():
    pass
class Mammal(Animal):
    pass
class Bird(Animal):
    pass
class Dog(Mammal):
    pass
class Bat(Mammal):
    pass
class Parrot(Bird):
    pass
class Eagle(Bird):
    pass

#加入功能类
class Runnable():
    def run(self):
        print u"跑起来可快了"
class Flyable():
    def fly(self):
        print U"飞起来可高了"

#通过功能类对原始分类进行改造,得到新的分类
class Dog(Mammal,Runnable):
    pass
class Eagle(Bird,Flyable):
    pass
```

11.5 构造方法:初始化对象,即当对象被创建后立刻调用构造方法

```
class Method:
    def __init__(self): #初始化对象
        self.the_function_you_wanna_initiate=the_initiate_value
        self.pageIndex=1
        self.user_agent="Mozilla/5.0 (Windows NT 6.1; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/31.0.1650.63
```

```
    Safari/537.36"
6.          self.headers={"User-Agent":self.user_agent}
7.
8.  f=Method()
9.  f.user_agent #输出url
```

- `__init__`方法的第一个参数永远是self,self指向创建的实例本身
- 在`__init__`方法内部,就可以把各种属性绑定到self

11.6 常用内建函数

```
1.  C.__name__
2.  C.__doc__ #文档字符串
3.  C.__base__ #以tuple形式返回所有父类
4.  C.__dict__ #类属性
5.  C.__module__ #该类定义所处的模块
6.  I.__class__ #实例I对应的类
7.  issubclass(A,B) #判断A是否是B的子类
8.  isinstance(object,class) #判断object是否是class的实例
9.  dir()
```

- `attr()`系列函数:在class/instance中频繁使用

```
1.  hasattr(instance,"attribute") #检查某个instance是否具有某个attribute
2.  getattr(instance,"attribute") #获取某个实例的某个属性
3.  getattr(instance,"attribute","value") #给予某个不存在的属性默认值,返回该属性
4.  setattr(instance,"attribute","value") #给某个属性赋值或创建新属性,不返回
5.  delattr(instance,"attribute") #删除某个实例的某个属性
```

11.7 super().**init**()

- 传统情况的继承情况如下:

```
1.  class A():
2.      def __init__(self):
3.          print "enter A"
4.          print "leave A"
5.
6.  class B(A):
7.      def __init__(self):
```

```
        print "enter B"
        A.__init__(self)
        print "leave B"

exampleB=B()
>>> enter B
>>> enter A
>>> leave A
>>> leave B
```

- 当一个子类的父类发生变化时（如类B的父类由A变为C时），必须遍历整个类定义

```
class B(C):     # A --> C
    def __init__(self):
        print "enter B"
        C.__init__(self) # A --> C
        print "leave B"
```

- 引入super：只需要修改一处

```
class B(C):     # A --> C
    def __init__(self):
        print "enter B"
        super(B, self).__init__()
        print "leave B"
```

- 解决错误:super() argument 1 must be type, not classobj
  - PYTHON里的SUPER只能用在新式类中，不能用于以前的经典类
  - 如果经典类被作为父类，子类调用父类的构造函数时会出错!
  - ClassicClass: `class Father()`-->深度优先继承
  - NewStyleClass: `class Father(object)`-->广度优先继承
  - 解决措施,将父类`class Father()`改为`class Father(object)`

Chapter 12 执行环境

12.1 四种可调用对象

- 函数/方法/类/实例

12.2 可执行的对象声明和内建函数

- `callable(obj)` :object是否可调用
  compile(string,file,type): 创建type类型的代码对象到file(通常设为"")
- 举个栗子:

```
a=compile('100+100','','eval') #创建求值对象
eval(a) #求值,返回200
exec()/eval() #执行某段代码/求值某个数学表达式

eval("100+200")
f=open("xyq.py")
exec(f)
```

- 注意exec只能执行一次,再次调用会失败
  - exec已经读取全部代码并停留在文件末尾,再次调用无代码可读,返回错误
  - 解决方法:`seek()`回到文件开头

```
f.seek(0)
exec(f)
input() #等价于eval(raw_input())
```

12.3 导入其他python程序

12.3.1 import

```
if __name__=='__main__':
#导入时不执行代码,只有在执行py脚本时才执行代码
```

- 再举个栗子:

```
#构建一个py文件:import2.py
if __name__=='__main__':
    print '123'
#执行该脚本时会打印,但是仅仅import则没有反应
```

12.3.2 执行脚本`python import2.py`

Chapter 13 正则表达式Regular Expression

13.1 Definithon

- 一些由字符和特殊符号组成的string,可用于匹配有相似特征的字符串

13.2 匹配符号

13.2.1 基本符号

```
. #匹配除换行符外的任意字符：d.ar
^xxx #字符串开始：^dear
xxx$ #字符串结尾：dear$
[xxx] #任意一个字符：[0-9]
[^xxx] #不出现任意一个字符：[^0-9]匹配非数字
xxx* #零次或更多次
xxx+ #一次或更多次
xxx? #零次或一次
{N} #重复N次：[0-9]{3}
{M,N} #重复M到N次
{M,}/{,N} #重复至少M/至多N次
\w #[A-Za-z0-9]
\s #任意空白,[\n\t\r\v\f]
\d #[0-9]
\b #边界匹配:\bThe\b
\W\S\D\B #非w/非空白/非数字/非头尾
\xxx #转义：\\, \., \*
#注意%的转义为%%：'growth rate：%d %%'%7

r'\nsss'-->'\nsss'#prefix r 取消一切转义

A|B #匹配A或B,A优先匹配
#xxx #注释
```

- 再举几个栗子:

```
\w+-\d+ #至少一个非换行字符,至少一个数字,中间用"-"连接
\w+@\w+.com #xxx@xxx.com
.* #在遇到换行符之前匹配任意内容
[aeiou]{3|5} #匹配某个元音字母3次或5次
\bhi\b #only匹配"hi",除去"history","him"等
```

```
\b\w{6}\b #任意长度为6的字符串
\Bthe #包含the但不以其开头的单词
^\d{5,12}$ #5-12位数字,等价于\b\d{5,12}\b
```

13.2.2 特殊构造

- 贪婪与非贪婪匹配: 默认为贪婪匹配,如`a.*b` 匹配axxxb尽可能长的字符串
- `?` :非贪婪匹配,尽可能少重复
  - `*? / +? / ?? / {n,m}? / {n,}`

```
a.*?b #a开头b结尾尽可能短的字符串
abc.*?xyq #abc+中间尽可能少+xyq,这个在爬虫中很常见
(?:abc){2} #abcabc
abc(?#comment)123 #忽略注释,匹配abc123
a(?=\d) #a后面是数字才会匹配
a(?!\d) #a后面不是数字才会匹配
(?<=\d)a #a前面是数字才会匹配
(?!=\d)a #a前面不是数字才会匹配
```

13.2.3 分组

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

13.3 re module in python

13.3.1 概览

```
compile(pattern,flags=0) #预编译
match(pattern,string,flags=0) #匹配string中的pattern
search(pattern,string,flags=0) #匹配string中的pattern
findall(pattern,string,flags=0) #匹配string中的所有patterns
finditer() #迭代器形式,对于每个匹配返回一个匹配对象
split(pattern,string,max=0) #pattern分隔符分割string,未给出max则分割全部
sub(pattern,repl,string,max=0) #使用repl替换string中的pattern,未给出max则替换全部
group(num) #返回全部匹配对象,或编号为num的subgroup
groups() #返回包含全部subgroups的tuple
```

- flags参数:可选模式

```
re.I #忽略大小写
re.M #多行模式
re.S #任意点匹配,'.'可以匹配换行符'\n'
re.X #忽略空白及注释(同一行#后的内容都被忽略)
```

13.3.2 re.compile()

- 进行匹配前,正则表达式pattern需要被预编译成regex object

```
pattern=re.compile("fuck.*?you")
myList=re.findall(pattern,strings,re.I)
```

13.3.3 group()/groups()

```
m=re.match('(\w\w\w)-(\d\d\d)','abc-123')
m.group() #返回'abc-123',全部匹配对象
m.group(1) #返回'abc',subgroup
m.group(0) #返回"abc-123"
m.group(2)  #返回"123"
m.groups() #返回('abc','123'),包含全部subgroups的tuple
```

13.3.4 match() 与 search()

- match只检测字符串开始部分,如果开头没匹配到就返回None；
- search则是在整个字符串中寻找相匹配的内容
- 举个栗子:

```
re.match("foo","seafood") #返回none
re.search("foo","seafood") #返回"foo"
```

- 多个正则匹配举例:

```
re.match("3.14",string) #匹配非空非换行字符"3x14"
re.match("3\.14",string) #使用\转义,匹配3.14
re.match("bat|bet|bit",string) #匹配多个字符串,bat优先级最高
re.match("[1-5][a-z][A-Z]",string) #匹配'3aW'
re.search("^The",string) #匹配开头的The
string='the. The' #失败
```

```
string='The. the' #返回'The'
re.search("\byou",string) #边界匹配,无法匹配'loveyou'
re.search("\Byou",string) #反边界匹配,匹配'loveyou'成功 ,"you"->False
```

13.3.5 findall()

- 找到所有出现的匹配,返回list

13.3.6 sub()/subn()

- 检索并替换,`subn()`还会额外给出替换次数

```
re.sub('x','fuck',string) #将x替换为fuck
```

13.3.7 split

```
re.split(':',"1:2:3") #返回('1','2','3')
```