# Python17个常用内置模块总结

转载[weixin\_33686714](https://me.csdn.net/weixin_33686714" \t "https://blog.csdn.net/weixin_33686714/article/details/_blank) 最后发布于2016-09-02 20:40:21 阅读数 572  收藏

展开

<--目录-->

1、getpass

2、os

3、sys

4、subprocess

5、hashlib

6、json

7、pickle

8、shutil

9、time

10、datetime

11、re

12、random

13、configparser

14、traceback

15、yaml

16、itertools

17、logging

1、getpass模块详解

pwd = getpass.getpass("请输入密码：")  #输入密码不可见

yh = getpass.getuser()                #显示当前登录系统用户名;

2、os模块

os.getcwd()                           #获取当前工作目录，即当前python脚本工作的目录路径

os.chdir("dirname")                   #改变当前脚本工作目录；相当于shell下cd

os.curdir                             #返回当前目录: ('.')

os.pardir                             #获取当前目录的父目录字符串名：('..')

os.makedirs('dirname1/dirname2')      #可生成多层递归目录

os.removedirs('dirname1')             #若目录为空，则删除，并递归到上一级目录，如若也为空，则删除，依此类推

os.mkdir('dirname')                   #生成单级目录；相当于shell中mkdir dirname

os.rmdir('dirname')                   #删除单级空目录，若目录不为空则无法删除，报错；相当于shell中rmdir dirname

os.listdir('dirname')                 #列出指定目录下的所有文件和子目录，包括隐藏文件，并以列表方式打印

os.remove()                           #删除一个文件

os.rename("oldname","newname")        #重命名文件/目录

os.stat('path/filename')              #获取文件/目录信息

os.sep                                #输出操作系统特定的路径分隔符，win下为"\\",Linux下为"/"

os.linesep                            #输出当前平台使用的行终止符，win下为"\t\n",Linux下为"\n"

os.pathsep                            #输出用于分割文件路径的字符串

os.name                               #输出字符串指示当前使用平台。win->'nt'; Linux->'posix'

os.system("bash command")             #运行shell命令，直接显示

os.environ                            #获取系统环境变量

os.path.abspath(path)                 #返回path规范化的绝对路径

os.path.split(path)                   #将path分割成目录和文件名二元组返回

os.path.dirname(path)                 #返回path的目录。其实就是os.path.split(path)的第一个元素

os.path.basename(path)                #返回path最后的文件名。如何path以／或\结尾，那么就会返回空值。即os.path.split(path)的第二个元素

os.path.exists(path)                  #如果path存在，返回True；如果path不存在，返回False

os.path.isabs(path)                   #如果path是绝对路径，返回True

os.path.isfile(path)                  #如果path是一个存在的文件，返回True。否则返回False

os.path.isdir(path)                   #如果path是一个存在的目录，则返回True。否则返回False

os.path.join(path1[, path2[, ...]])   #将多个路径组合后返回，第一个绝对路径之前的参数将被忽略

os.path.getatime(path)                #返回path所指向的文件或者目录的最后存取时间

os.path.getmtime(path)                #返回path所指向的文件或者目录的最后修改时间

3、sys模块

sys.argv                              #命令行参数List，第一个元素是程序本身路径

sys.exit(n)                           #退出程序，正常退出时exit(0)

sys.version                           #获取Python解释程序的版本信息

sys.maxint                            #最大的Int值

sys.path                              #返回模块的搜索路径，初始化时使用PYTHONPATH环境变量的值

sys.platform                          #返回操作系统平台名称

sys.stdout.write('please:')

val = sys.stdin.readline()[:-1]

4、subprocess模块

执行系统命令

os.system

commands.\*      --废弃，3.x中被移除

result = commands.getoutput('cmd')

以上执行shell命令的相关的模块和函数的功能均在 subprocess 模块中实现，并提供了更丰富的功能。

call

执行命令，返回状态码

ret = subprocess.call(["ls", "-l"], shell=False)

ret = subprocess.call("ls -l", shell=True)

shell = True ，允许 shell 命令是字符串形式

check\_call

执行命令，如果执行状态码是 0 ，则返回0，否则抛异常

subprocess.check\_call(["ls", "-l"])

subprocess.check\_call("exit 1", shell=True)

check\_output（此下两条命令在2.6执行失败，要是2.7应该才可以）

执行命令，如果状态码是 0 ，则返回执行结果，否则抛异常

subprocess.check\_output(["echo", "Hello World!"])

subprocess.check\_output("exit 1", shell=True)

subprocess.Popen(...)

用于执行复杂的系统命令

参数：

args：                      #shell命令，可以是字符串或者序列类型（如：list，元组）

bufsize：                   #指定缓冲。0 无缓冲,1 行缓冲,其他 缓冲区大小,负值 系统缓冲

stdin, stdout, stderr：     #分别表示程序的标准输入、输出、错误句柄

preexec\_fn：                #只在Unix平台下有效，用于指定一个可执行对象（callable object），它将在子进程运行之前被调用

close\_sfs：                 #在windows平台下，如果close\_fds被设置为True，则新创建的子进程将不会继承父进程的输入、输出、错误管道。所以不能将close\_fds设置为True同时重定向子进程的标准输入、输出与错误(stdin, stdout, stderr)。

shell：                     #同上

cwd：                       #用于设置子进程的当前目录

env：                       #用于指定子进程的环境变量。如果env = None，子进程的环境变量将从父进程中继承。

universal\_newlines：        #不同系统的换行符不同，True -> 同意使用 \n

startupinfo与createionflags #只在windows下有效

将被传递给底层的CreateProcess()函数，用于设置子进程的一些属性，如：主窗口的外观，进程的优先级等等

import subprocess

ret1 = subprocess.Popen(["mkdir","t1"])

ret2 = subprocess.Popen("mkdir t2", shell=True)

5、hashlib模块

用于加密相关的操作，代替了md5模块和sha模块，主要提供 SHA1, SHA224, SHA256, SHA384, SHA512 ，MD5 算法

import hashlib

# ######## md5 ########

hash = hashlib.md5()

hash.update('admin')

print hash.hexdigest()

# ######## sha1 ########

hash = hashlib.sha1()

hash.update('admin')

print hash.hexdigest()

# ######## sha256 ########

hash = hashlib.sha256()

hash.update('admin')

print hash.hexdigest()

# ######## sha384 ########

hash = hashlib.sha384()

hash.update('admin')

print hash.hexdigest()

# ######## sha512 ########

hash = hashlib.sha512()

hash.update('admin')

print hash.hexdigest()

以上加密算法虽然依然非常厉害，但时候存在缺陷，即：通过撞库可以反解。所以，有必要对加密算法中添加自定义key再来做加密。

import hashlib

# ######## md5 ########

hash = hashlib.md5('898oaFs09f')

hash.update('admin')

print hash.hexdigest()

还不够吊？python 还有一个 hmac 模块，它内部对我们创建 key 和 内容 再进行处理然后再加密

import hmac

h = hmac.new('wueiqi')

h.update('hellowo')

print h.hexdigest()

6,7、json 和 pickle

用于序列化的两个模块

json，用于字符串 和 python数据类型间进行转换

pickle，用于python特有的类型 和 python的数据类型间进行转换

Json模块提供了四个功能：dumps、dump、loads、load

pickle模块提供了四个功能：dumps、dump、loads、load

pickle

>>> import pickle

>>> data = {'k1' : 123, 'k2' : 'hello'}

>>> p\_str = pickle.dumps(data)   #序列化

>>> print p\_str

>>> loadsed = pickle.loads(p\_str)   #反序列化

>>> print loadsed

序列化到文件

>>> li = ['wsyht',11,22,'ok','yes']

>>> pickle.dump(li,open('test.txt','w'))   #序列化到文件

>>> pickle.load(open('test.txt'))          #从文件反序列化出来

json

>>> import json

>>> data = {'k1':123,'k2':'abc'}

>>> str = json.dumps(data)

>>> stt= json.loads(str)

序列化到文件

>>> li = ['wsyht',11,22,'ok','yes']

>>> json.dump(li,open('test.txt','w'))  #序列化到文件

>>> json.load(open('test.txt'))         #从文件反序化出来

8、shutil模块

shutil.make\_archive(base\_name, format,...)

创建压缩包并返回文件路径，例如：zip、tar

base\_name： 压缩包的文件名，也可以是压缩包的路径。只是文件名时，则保存至当前目录，否则保存至指定路径，

如：www                        =>保存至当前路径

如：/Users/wupeiqi/www =>保存至/Users/wupeiqi/

format： 压缩包种类，“zip”, “tar”, “bztar”，“gztar”

root\_dir： 要压缩的文件夹路径（默认当前目录）

owner： 用户，默认当前用户

group： 组，默认当前组

logger： 用于记录日志，通常是logging.Logger对象

#将 /Users/wupeiqi/Downloads/test 下的文件打包放置当前程序目录

import shutil

ret = shutil.make\_archive("wwwwwwwwww", 'gztar', root\_dir='/Users/wupeiqi/Downloads/test')

#将 /mnt下的文件打包放置 /tmp目录

import shutil

ret = shutil.make\_archive("/tmp/www", 'gztar', root\_dir='/mnt')  #2.6用不了，2.7或许可以

类似于高级API，而且主要强大之处在于其对文件的复制与删除操作更是比较支持好。

相关API介绍

copyfile(src, dst)

从源src复制到dst中去。当然前提是目标地址是具备可写权限。抛出的异常信息为

IOException. 如果当前的dst已存在的话就会被覆盖掉。

 copyfile( src, dst) 从源src复制到dst中去。当然前提是目标地址是具备可写权限。抛出的异常信息为IOException. 如果当前的dst已存在的话就会被覆盖掉

 copymode( src, dst) 只是会复制其权限其他的东西是不会被复制的

 copystat( src, dst) 复制权限、最后访问时间、最后修改时间

 copy( src, dst)   复制一个文件到一个文件或一个目录

 copy2( src, dst) 在copy上的基础上再复制文件最后访问时间与修改时间也复制过来了，类似于cp –p的东西

 copy2( src, dst) 如果两个位置的文件系统是一样的话相当于是rename操作，只是改名；如果是不在相同的文件系统的话就是做move操作

 copytree(olddir,newdir,True/Flase) 把olddir拷贝一份newdir，如果第3个参数是True，则复制目录时将保持文件夹下的符号连接，如果第3个参数是False，则将在复制的目录下生成物理副本来替代符号连接

 shutil.rmtree("te")      删除一个目录

import shutil

shutil.copyfile('f:/temp.txt', 'f:/os.txt') #复制文件

shutil.copytree('f:/temp', 'f:/os')  #复制目录

# ######## zip的用法 ########

shutil 对压缩包的处理是调用 ZipFile 和 TarFile 两个模块来进行的，详细：

import zipfile

# 压缩

z = zipfile.ZipFile('laxi.zip', 'w')

z.write('a.log')        #压缩包写入a.log

z.write('data.data')    #写入data文件

z.close()

# 解压

z = zipfile.ZipFile('laxi.zip', 'r')

z.extractall()

z.close()

9、time模块

三种表示主式：

1、时间戳 1970年1月1日后的秒

2、元组包含了：年、日、星期等...time.struct\_time

3、格式化的字符串 2014-11-11 11:11 print time.time()

#时间戳形式存在

print time.time()

print time.mktime(time.localtime())  #print (time.localtime())此为元组形式，这一整句意思是把元组形式转化成时间戳形式

#元组形式存在

print time.gmtime() #可加时间戳参数

print time.localtime() #可加时间戳参数

print time.strptime('2014-11-11','%Y-%m-%d') #字符串形式转换成元组形式

#字符串形式存在

print time.strftime('%Y-%m-%d')  #默认当前时间，必须记住，工作中用得最多

print time.strftime('%Y-%m-%d',time.localtime())  #默认当前时间

print time.asctime()

print time.asctime(time.localtime())

print time.ctime(time.time())

时间的三种表示方式演示

>>> import time

>>> print time.time()

1469014348.5   #秒，时间戳的方式

>>> print time.gmtime()

time.struct\_time(tm\_year=2016, tm\_mon=7, tm\_mday=20, tm\_hour=11, tm\_min=25, tm\_sec=53, tm\_wday=2, tm\_yday=202, tm\_isdst=0)

>>> print time.strftime('%Y-%m-%d %H:%M:%S')

2016-07-20 19:36:16

10、datetime模块

import datetime

'''

datetime.date：表示日期的类。常用的属性有year, month, day

datetime.time：表示时间的类。常用的属性有hour, minute, second, microsecond

datetime.datetime：表示日期时间

datetime.timedelta：表示时间间隔，即两个时间点之间的长度

timedelta([days[, seconds[, microseconds[, milliseconds[, minutes[, hours[, weeks]]]]]]])

strftime("%Y-%m-%d")

'''

import datetime

print datetime.datetime.now()

print datetime.datetime.now() - datetime.timedelta(days=5)

11、re模块

compile

match search findall

group groups

正则表达式常用格式：

　　字符：\d \w \t  .

　　次数：\* + ? {m} {m,n}

示例：

#!#/usr/bin/env python

#coding:utf-8

import re

result1 = re.match('\d+', '1afsfweasfcxvsfd123')  #在你给的字符串起始位置去匹配,\d从数字开始找，+表示一个到多个

if result1:   #当result1等于True的时候，就是匹配，如果匹配就输出里面的内容

    print result1.group()  #用group方法把他匹配的内容输出出来

else:

    print 'nothing'

result2 = re.search('\d+', 'alsfj3af') #在整个内容里面去匹配，\d从数字开始找，+表示一个到多个

if result2:

    print result2.group()    #用group方法把他匹配的内容输出出来

result3 = re.findall('\d+', 'asfaf11sf22lj33') #只要匹配全都拿出来

print result3

com = re.compile('\d+')

print com.findall('asfaf11sf22lj33')

result5 = re.search('(\d+)\w\*(\d+)','aasflsjfa12aaljsf22lj13bb')

print result5.group()   #所有匹配内容输出

print result5.groups()  #只把括号\d，也就是组里面的内容输出

result6 = re.search('a{3,5}','aaaaaa') #匹配3到5次的aaaaa输出出来

print result6.group()

总结：

match：只在第一个字符串开始找，如果没有匹配，则不再继续找，如果第一个字符串中有，则只输出第一个

searh: 在所有内容里找，直到找到为止，但只输出找到的第一个

findall：把所有找到的匹配的内容，都通过列表的形式打印出来

compile: 编译之后再去匹配，这样可以加快匹配的速度

group: 把他匹配的内容输出出来

groups：分组

匹配的字符：

\d：表示数字的意思

\w: 代表下划线，字母，数字

\t：制表符，除了回车以外的所有字符

匹配的次数：

\* 大于等于0，0到多个

+ 大于等于1，1个到多个

?  0或1

{m} 次数，如a{6}，出现6次a的进行匹配

{m,n} 如a{3,7} 出现3到7次的就进行匹配

例子1：

法1

>>> ip = '12.23.84.dsfa.23s.3234~lsjfw+23sfaf192.168.32.43\_w342d~@#9436'

>>> import re

>>> re.findall('[0-9]{1,3}',ip)

['12', '23', '84', '23', '323', '4', '23', '192', '168', '32', '43', '342', '943', '6']

>>> re.findall('[0-9]{1,3}\.[0-9]{1,3}',ip)

['12.23', '192.168', '32.43']

>>> re.findall('[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}',ip)

['12.23.84', '192.168.32']

>>> re.findall('[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}',ip)

['192.168.32.43']

法2:

>>> re.findall('(\d+)',ip)

['12', '23', '84', '23', '3234', '23', '192', '168', '32', '43', '342', '9436']

>>> re.findall('(\.+\d+){1,3}',ip)

['.84', '.23', '.3234', '.43']

>>> re.findall('(?:\.+\d+){1,3}',ip)    #?:表示匹配括号的那一组数据，必须连着

['.23.84', '.23', '.3234', '.168.32.43']

>>> re.findall('[0-9]{1,3}(?:\.+\d+){3}',ip)

['192.168.32.43']

法3：

>>> re.findall('(?:\d+\.+){3}\d{1,3}',ip)

['192.168.32.43']

法4：

>>> re.findall('(?:\d{1,3}\.){3}\d{1,3}',ip)

['192.168.32.43']

12、random模块

import random

print random.random()

print random.randint(1,2)

print random.randrange(1,10)

随机验证码实例：

import random

checkcode = ''

for i in range(4):

    current = random.randrange(0,4)

    if current != i:

        temp = chr(random.randint(65,90))

    else:

        temp = random.randint(0,9)

    checkcode += str(temp)

print checkcode

13、ConfigParser模块

用于对特定的配置进行操作，当前模块的名称在 python 3.x 版本中变更为 configparser。

1.读取配置文件

-read(filename) 直接读取ini文件内容

-sections() 得到所有的section，并以列表的形式返回

-options(section) 得到该section的所有option

-items(section) 得到该section的所有键值对

-get(section,option) 得到section中option的值，返回为string类型

-getint(section,option) 得到section中option的值，返回为int类型

2.写入配置文件

-add\_section(section) 添加一个新的section

-set( section, option, value) 对section中的option进行设置

         需要调用write将内容写入配置文件。

[root@test1 mnt]# cat data.txt

[sec\_a]

a\_key1 = 20

a\_key2 = 10

[sec\_b]

b\_key1 = 121

b\_key2 = b\_value2

b\_key3 = $r

b\_key4 = 127.0.0.1

>>> import ConfigParser

>>> cf = ConfigParser.ConfigParser()

>>> cf.read("data.txt")

['data.txt']

>>> secs = cf.sections()  #获得所有区域

>>> print 'sections:', secs

sections: ['sec\_b', 'sec\_a']

>>> opts = cf.options("sec\_a")

>>> print 'options:', opts

options: ['a\_key1', 'a\_key2']

>>>

>>> for sn in secs:

...  print cf.options(sn)   #打印出每个区域的所有属性

...

['b\_key4', 'b\_key1', 'b\_key2', 'b\_key3']

['a\_key1', 'a\_key2']

>>> str\_val = cf.get("sec\_a", "a\_key1")

>>> int\_val = cf.getint("sec\_a", "a\_key2")

>>> print "value for sec\_a's a\_key1:", str\_val

value for sec\_a's a\_key1: 20

>>> print "value for sec\_a's a\_key2:", int\_val

value for sec\_a's a\_key2: 10

>>> cf.set("sec\_b", "b\_key3", "new-$r")

>>> cf.set("sec\_b", "b\_newkey", "new-value")

>>> cf.add\_section('a\_new\_section')

>>> cf.set('a\_new\_section', 'new\_key', 'new\_value')

>>> cf.write(open("data.txt", "w"))

>>> cf.has\_section('a\_new\_section')  #判断存不存在[sec\_a]

True

>>> cf.remove\_section('sec\_a')   #删除[sec\_a]

True

>>> cf.has\_section('a\_section')    #判断存不存在[sec\_a]

False

>>> cf.write(open("data.txt", "w"))

14、traceback模块

[root@test1 mnt]# cat test.py

#!/usr/bin/env python

#coding:utf-8

import traceback

try:

    1/0

except Exception,e:

    #print e

    traceback.print\_exc(file=open('tb.txt','w+'))

else:

    print 'success'

15、yaml模块的使用

yaml在python上的具体实现：PyYaml

将yaml写成配置脚本test.yaml ,以下介绍如何读写yaml配置。

使用python的yaml库PyYAML。http://pyyaml.org/

安装到python lib下后就可以正常使用了。

#加载yaml

import yaml

f = open('test.yaml')   #读取文件

x = yaml.load(f)   #导入

print x

f.close()

import yaml

f = open('d:/newtree.yaml', "w")

yaml.dump(dataMap, f)

f.close()

16、itertools模块的使用

[root@test1 mnt]# cat test.py

# ######## count(1) ########

import itertools

natuals = itertools.count(1)   #count创建无限个迭代器

for n in natuals:

        print n

# ######## cycle() ########

cycle()会把传入的一个序列无限重复下去：

>>> import itertools

>>> cs = itertools.cycle('ABC') # 注意字符串也是序列的一种

>>> for c in cs:

...     print c

# ######## repeat() ########

repeat()负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复次数：

>>> ns = itertools.repeat('A', 10)

>>> for n in ns:

...     print n

...

打印10次'A'

# ######## takewhile() ########

>>> natuals = itertools.count(1)

>>> ns = itertools.takewhile(lambda x: x <= 10, natuals)

>>> for n in ns:

...     print n

...

打印出1到10

# ######## chain() ########

for c in itertools.chain('ABC', 'XYZ'):

    print c

# 迭代效果：'A' 'B' 'C' 'X' 'Y' 'Z'

# ######## groupby() ########

groupby()把迭代器中相邻的重复元素挑出来放在一起：

>>> for key, group in itertools.groupby('AAABBBCCAAA'):

...     print key, list(group) # 为什么这里要用list()函数呢？

...

A ['A', 'A', 'A']

B ['B', 'B', 'B']

C ['C', 'C']

A ['A', 'A', 'A']

# ######## groupby() ########

实际上挑选规则是通过函数完成的，只要作用于函数的两个元素返回的值相等，这两个元素就被认为是在一组的，而函数返回值作为组的key。如果我们要忽略大小写分组，就可以让元素'A'和'a'都返回相同的key：

>>> for key, group in itertools.groupby('AaaBBbcCAAa', lambda c: c.upper()):

...     print key, list(group)

...

A ['A', 'a', 'a']

B ['B', 'B', 'b']

C ['c', 'C']

A ['A', 'A', 'a']

# ######## imap() ########

>>> for x in itertools.imap(lambda x, y: x \* y, [10, 20, 30], itertools.count(1)):

...     print x

itertools

阅读: 5044

Python的内建模块itertools提供了非常有用的用于操作迭代对象的函数。

首先，我们看看itertools提供的几个“无限”迭代器：

>>> import itertools

>>> natuals = itertools.count(1)

>>> for n in natuals:

...     print n

...

1

2

3

...

因为count()会创建一个无限的迭代器，所以上述代码会打印出自然数序列，根本停不下来，只能按Ctrl+C退出。

cycle()会把传入的一个序列无限重复下去：

>>> import itertools

>>> cs = itertools.cycle('ABC') # 注意字符串也是序列的一种

>>> for c in cs:

...     print c

...

'A'

'B'

'C'

'A'

'B'

'C'

...

同样停不下来。

repeat()负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复次数：

>>> ns = itertools.repeat('A', 10)

>>> for n in ns:

...     print n

...

打印10次'A'

无限序列只有在for迭代时才会无限地迭代下去，如果只是创建了一个迭代对象，它不会事先把无限个元素生成出来，事实上也不可能在内存中创建无限多个元素。

无限序列虽然可以无限迭代下去，但是通常我们会通过takewhile()等函数根据条件判断来截取出一个有限的序列：

>>> natuals = itertools.count(1)

>>> ns = itertools.takewhile(lambda x: x <= 10, natuals)

>>> for n in ns:

...     print n

...

打印出1到10

itertools提供的几个迭代器操作函数更加有用：

chain()

chain()可以把一组迭代对象串联起来，形成一个更大的迭代器：

for c in itertools.chain('ABC', 'XYZ'):

    print c

# 迭代效果：'A' 'B' 'C' 'X' 'Y' 'Z'

groupby()

groupby()把迭代器中相邻的重复元素挑出来放在一起：

>>> for key, group in itertools.groupby('AAABBBCCAAA'):

...     print key, list(group) # 为什么这里要用list()函数呢？

...

A ['A', 'A', 'A']

B ['B', 'B', 'B']

C ['C', 'C']

A ['A', 'A', 'A']

实际上挑选规则是通过函数完成的，只要作用于函数的两个元素返回的值相等，这两个元素就被认为是在一组的，而函数返回值作为组的key。如果我们要忽略大小写分组，就可以让元素'A'和'a'都返回相同的key：

>>> for key, group in itertools.groupby('AaaBBbcCAAa', lambda c: c.upper()):

...     print key, list(group)

...

A ['A', 'a', 'a']

B ['B', 'B', 'b']

C ['c', 'C']

A ['A', 'A', 'a']

imap()

imap()和map()的区别在于，imap()可以作用于无穷序列，并且，如果两个序列的长度不一致，以短的那个为准。

>>> for x in itertools.imap(lambda x, y: x \* y, [10, 20, 30], itertools.count(1)):

...     print x

注意imap()返回一个迭代对象，而map()返回list。当你调用map()时，已经计算完毕：

>>> r = map(lambda x: x\*x, [1, 2, 3])

>>> r # r已经计算出来了

[1, 4, 9]

当你调用imap()时，并没有进行任何计算：

>>> r = itertools.imap(lambda x: x\*x, [1, 2, 3])

>>> r

<itertools.imap object at 0x103d3ff90>

# r只是一个迭代对象

itertools

阅读: 5044

Python的内建模块itertools提供了非常有用的用于操作迭代对象的函数。

首先，我们看看itertools提供的几个“无限”迭代器：

>>> import itertools

>>> natuals = itertools.count(1)

>>> for n in natuals:

...     print n

...

1

2

3

...

因为count()会创建一个无限的迭代器，所以上述代码会打印出自然数序列，根本停不下来，只能按Ctrl+C退出。

cycle()会把传入的一个序列无限重复下去：

>>> import itertools

>>> cs = itertools.cycle('ABC') # 注意字符串也是序列的一种

>>> for c in cs:

...     print c

...

'A'

'B'

'C'

'A'

'B'

'C'

...

同样停不下来。

repeat()负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复次数：

>>> ns = itertools.repeat('A', 10)

>>> for n in ns:

...     print n

...

打印10次'A'

无限序列只有在for迭代时才会无限地迭代下去，如果只是创建了一个迭代对象，它不会事先把无限个元素生成出来，事实上也不可能在内存中创建无限多个元素。

无限序列虽然可以无限迭代下去，但是通常我们会通过takewhile()等函数根据条件判断来截取出一个有限的序列：

>>> natuals = itertools.count(1)

>>> ns = itertools.takewhile(lambda x: x <= 10, natuals)

>>> for n in ns:

...     print n

...

打印出1到10

itertools提供的几个迭代器操作函数更加有用：

chain()

chain()可以把一组迭代对象串联起来，形成一个更大的迭代器：

for c in itertools.chain('ABC', 'XYZ'):

    print c

# 迭代效果：'A' 'B' 'C' 'X' 'Y' 'Z'

groupby()

groupby()把迭代器中相邻的重复元素挑出来放在一起：

>>> for key, group in itertools.groupby('AAABBBCCAAA'):

...     print key, list(group) # 为什么这里要用list()函数呢？

...

A ['A', 'A', 'A']

B ['B', 'B', 'B']

C ['C', 'C']

A ['A', 'A', 'A']

实际上挑选规则是通过函数完成的，只要作用于函数的两个元素返回的值相等，这两个元素就被认为是在一组的，而函数返回值作为组的key。如果我们要忽略大小写分组，就可以让元素'A'和'a'都返回相同的key：

>>> for key, group in itertools.groupby('AaaBBbcCAAa', lambda c: c.upper()):

...     print key, list(group)

...

A ['A', 'a', 'a']

B ['B', 'B', 'b']

C ['c', 'C']

A ['A', 'A', 'a']

imap()

imap()和map()的区别在于，imap()可以作用于无穷序列，并且，如果两个序列的长度不一致，以短的那个为准。

>>> for x in itertools.imap(lambda x, y: x \* y, [10, 20, 30], itertools.count(1)):

...     print x

...

10

40

90

注意imap()返回一个迭代对象，而map()返回list。当你调用map()时，已经计算完毕：

>>> r = map(lambda x: x\*x, [1, 2, 3])

>>> r # r已经计算出来了

[1, 4, 9]

当你调用imap()时，并没有进行任何计算：

>>> r = itertools.imap(lambda x: x\*x, [1, 2, 3])

>>> r

<itertools.imap object at 0x103d3ff90>

# r只是一个迭代对象

必须用for循环对r进行迭代，才会在每次循环过程中计算出下一个元素：

>>> for x in r:

...     print x

...

1

4

9

这说明imap()实现了“惰性计算”，也就是在需要获得结果的时候才计算。类似imap()这样能够实现惰性计算的函数就可以处理无限序列：

>>> r = itertools.imap(lambda x: x\*x, itertools.count(1))

>>> for n in itertools.takewhile(lambda x: x<100, r):

...     print n

17、logging模块

用于便捷记录日志且线程安全的模块

import logging

logging.basicConfig(filename='log.log',

                    format='%(asctime)s - %(name)s - %(levelname)s -%(module)s:  %(message)s',

                    datefmt='%Y-%m-%d %H:%M:%S %p',

                    level=5)

logging.debug('debug')

logging.info('info')

logging.warning('warning')

logging.error('error')

logging.critical('critical')

logging.log(10,'log')

对于等级：

CRITICAL = 50

FATAL = CRITICAL

ERROR = 40

WARNING = 30

WARN = WARNING

INFO = 20

DEBUG = 10

NOTSET = 0