PyMOTW doc

Release 1.0

vbarter and liz

December 31, 2008

CONTENTS

0.1 介绍

0.2 所有文档

0.2.1 PyMOTW: anydbm

- 模块：anydbm
- 目的：anydbm提供了针对DBM-style、String-keyed数据库的字典接口。
- python版本：1.4++

描述

anydbm面向DBM数据库，利用简单的字符串值作为key来访问包含字符串的记录。它利用whichdb模块来识别dbhash、gdbm和dbm数据库，并使用appropriate模块来打开它们，常常在shelve 中作为后端使用，比如我们知道如何利用pickle来存储对象。她常被用作为shelve的后端，就像我们知道如何利用pickle来存储对象。

创建一个新的数据库

新数据库的存储格式是可以通过查询如下模块来选择：

- dbhash
- gdbm
- dbm
- dumbdbm

open函数通过flags标志来控制如何处理数据库文件，当在必要时创建一个新的数据库时候，使用"c",当经常要创建一个新数据库时，使用"n"。

c和n的区别： 也就是说用c，如果不存在则创建，如果存在就不创建新的了，用n的话，不管存不存在都是创建新的空数据库。

开始，我们加载一些有用的模块：

```
import anydbm

db = anydbm.open('/tmp/example.db', 'n')
db['key'] = 'value'
db['today'] = 'Sunday'
db['author'] = 'Doug'
db.close()
```

在这个例子中，这个文件会总是被重新初始化，如果想知道被创建的数据库类型，可以使用whichdb。

```
import whichdb

print whichdb.whichdb('/tmp/example.db')
```

你得到的结果可能会不同，它取决于你安装在系统中的模块。

```
$ python anydbm_whichdb.py
dbhash
```

打开一个存在的数据库

要打开一个存在的数据库，使用标记"r"（只读）或者"w"（读写）。你不需要担心格式问题，因为数据库格式会自动由whichdb来识别，如果一个文件可以被识别，那么对应的模块会打开它。

```
import anydbm

db = anydbm.open('/tmp/example.db', 'r')
try:
    print 'keys():', db.keys()
    for k, v in db.iteritems():
        print 'iterating:', k, v
    print 'db["author"] =', db['author']
 finally:
    db.close()
```

一旦打开，db就是一个字典对象，支持一些常规方法:

```
$ python anydbm_existing.py
keys(): ['key', 'today', 'author']
iterating: key value
iterating: today Sunday
iterating: author Doug
db["author"] = Doug
```

错误案例

数据库关键词必须是字符串。

```
import anydbm

db = anydbm.open('/tmp/example.db', 'w')
try:
    db[1] = 'one'
finally:
    db.close()
```

传递其它类型结果将导致TypeError异常。

```
$ python anydbm_intkeys.py
Traceback (most recent call last):
File "/Users/dhellmann/Documents/PyMOTW/in_progress/anydbm/PyMOTW/anydbm/anydbm_intkeys.py", line 16,
db[1] = 'one'
File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/bsddb/__init__.py", line 230, i
_DeadlockWrap(wrapF) # self.db[key] = value
File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/bsddb/dbutils.py", line 62, in
return function(*_args, **_kwargs)
File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/bsddb/__init__.py", line 229, i
self.db[key] = value
TypeError: Integer keys only allowed for Recno and Queue DB's
```

值必须是字符串或者是空。

```
import anydbm

db = anydbm.open('/tmp/example.db', 'w')
try:
    db['one'] = 1
finally:
    db.close()
```

如果值是非字符串，那么同样会抛出TypeError异常。

```
$ python anydbm_intvalue.py
Traceback (most recent call last):
File "/Users/dhellmann/Documents/PyMOTW/in_progress/anydbm/PyMOTW/anydbm/anydbm_intvalue.py", line 1
db['one'] = 1
File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/bsddb/__init__.py", line 230,
_DeadlockWrap(wrapF) # self.db[key] = value
File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/bsddb/dbutils.py", line 62, in
return function(*_args, **_kwargs)
File "/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/bsddb/__init__.py", line 229,
self.db[key] = value
TypeError: Data values must be of type string or None.
```

参考

- 标准库文件：anydbm

0.2.2 PyMOTW: base64

base64模块提供一些函数l来把二进制数据转换为ASCII集，通常在明文协议的传输中使用。

- 模块：base64
- 目的：编码二进制数据转化为ASCII码
- python版本：1.4+

描述

base64、base32、base16可以分别编码转化8位字节为6位、5位、4位，允许非ASCII字节以编码为ASCII码的协议中传输，例如SMTP，“base”值对应是在每一个编码中字母表的长度。有一些原始编码的url类型会使用略有不同的结果。

Base64 编码

简单的文本编码示例如下：

```
import base64

initial_data = open(__file__, 'rt').read()
```

```
encoded_data = base64.b64encode(initial_data)

num_initial = len(initial_data)
padding = { 0:0, 1:2, 2:1 }[num_initial % 3]

print '%d bytes before encoding' % num_initial
print 'Expect %d padding bytes' % padding
print '%d bytes after encoding' % len(encoded_data)
print
print encoded_data
```

输出显示原来529字节的文本在编码之后被扩展到了708个字节，从编码过程来看，每一个24位序列（3个字节）作为输入，输出时候则增加了4个字节，最后2个字符“==”，则是简单的追加，因为原始字符串的位数不能被24整除。

在标准输出中时没有很多回车府，但是为了在文档中有好的可读性，在如下显示中稍作了变化。

```
$ python base64_b64encode.py
529 bytes before encoding
    Expect 2 padding bytes
    708 bytes after encoding

IyEvdXNyL2Jpbi9lbnYgcHl0aG9uCiMgZW5jb2Rpbmc
6IHV0Zi04CiMKIyBDb3B5cmlnaHQgKGMpIDIwMDggRG
91ZyBIZWxsbWFubiBBbGwgcmlnaHRzIHJlc2VydmVkL
gojCiIiIgoiIiIKCl9fdmVyc2lvbl9fID0gIiRJZDog
cHltb3R3LnB5IDEyMzkgMjAwOC0wMS0xNiAxMDo1NTo
xOVogZGhlbGxtYW5uICQiCgppbXBvcnQgYmFzZTY0Cg
ppbml0aWFsX2RhdGEgPSBvcGVuKF9fZmlsZV9ffLCAnc
nQnKS5yZWFkKCkCmVuY29kZWRfZGF0YSA9IGJhc2U2
NC5iNjRlbmNvZGUoaW5pdGlhbF9kYXRhKQoKbnVtX2l
uaXRpYWwgPSBsZW4oaW5pdGlhbF9kYXRhKQpwYWRkaW
5nID0geyAwOjAsIDE6MiwgMjoxIH1bbnVtX2luaXRpY
WwgJSAzXQoKcHJpbnQgJyVkIGJ5dGVzIGJlZm9yZSBl
bmNvZGluZycgJSBudW1faW5pdGlhbApwcmludCAnRXh
wZWN0ICVkIHBhZGRpbmcgYnl0ZXMnICUgcGFkZGluZw
pwcmludCAnJWQgYnl0ZXMgYWZ0ZXIgZW5jb2RpbmcnI
CUgbGVuKGVuY29kZWRfZGF0YSkKcHJpbnQKcHJpbnQg
ZW5jb2RlZF9kYXRhCg==
```

Base64 解码

编码的字符串可以转换为原来的格式，利用反向查询，把4个字节转换为3个字节。b64decode()函数可以帮助你。

```
import base64

original_string = 'This is the data, in the clear.'
print 'Original:', original_string

encoded_string = base64.b64encode(original_string)
print 'Encoded :', encoded_string

decoded_string = base64.b64decode(encoded_string)
print 'Decoded :', decoded_string
```

```
$ python base64_b64decode.py
Original: This is the data, in the clear.
Encoded : VGhpcyBpcyB0aGUgZGF0YSwgaW4gdGhlIGNsZWFyLg==
Decoded : This is the data, in the clear.
```

URL-Safe变化

默认的base64字母表可能会使用+和/，而这些字符可能出现在url中，因此必须为这些字符指定可选择的编码情况，+由a-来代替，(_)来代替/，其他字母表还是相同。

```
import base64

for original in [ '\xfb\xef', '\xff\xff' ]:
    print 'Original         :', repr(original)
    print 'Standard encoding:', base64.standard_b64encode(original)
    print 'URL-safe encoding:', base64.urlsafe_b64encode(original)
    print
```

```
$ python base64_urlsafe.py
Original         : '\xfb\xef'
Standard encoding: ++8=
URL-safe encoding: --8=

Original         : '\xff\xff'
Standard encoding: //8=
URL-safe encoding: __8=
```

其他编码

除了base 64以外，还有base 32和base 16（16进制）提供函数用于编码数据。

```
import base64

original_string = 'This is the data, in the clear.'
print 'Original:', original_string

encoded_string = base64.b32encode(original_string)
print 'Encoded :', encoded_string

decoded_string = base64.b32decode(encoded_string)
print 'Decoded :', decoded_string
```

```
$ python base64_base32.py
Original: This is the data, in the clear.
Encoded : KRUGS4ZANFZSA5DIMUQGIYLUMEWCA2LOEB2GQZJAMNWGKYLSFY======
Decoded : This is the data, in the clear.
```

base 16中的函数是以16进制方式工作。

```
import base64

original_string = 'This is the data, in the clear.'
print 'Original:', original_string

encoded_string = base64.b16encode(original_string)
print 'Encoded :', encoded_string

decoded_string = base64.b16decode(encoded_string)
print 'Decoded :', decoded_string


$ python base64_base16.py
Original: This is the data, in the clear.
Encoded : 546869732069732074686520646174612C20696E2074686520636C6561722E
Decoded : This is the data, in the clear.
```

参考

- RFC 3548 - The Base16, Base32, and Base64 Data Encodings

0.2.3 PyMOTW: datetime

datetime 模块包含了一些用于时间解析、格式化、计算的函数。

- 模块：datetime
- 目的：日期/时间处理
- python版本：2.3+

时间

时间值由time类来表示，Times有小时，分，秒和微秒属性，以及包含时区信息。初始化time实例的参数是可选的，但这样的话，你将获得初始值0（也许不是你所想要的）。

```
import datetime

t = datetime.time(1, 2, 3)
print t
print 'hour :', t.hour
print 'minute:', t.minute
print 'second:', t.second
print 'microsecond:', t.microsecond
print 'tzinfo:', t.tzinfo


$ python datetime_time.py
01:02:03
hour  : 1
minute: 2
second: 3
microsecond: 0
tzinfo: None
```

一个time实例只包含时间值，不包含日期值。

```
import datetime

print 'Earliest  :', datetime.time.min
print 'Latest    :', datetime.time.max
print 'Resolution:', datetime.time.resolution
```

类属性中的最大最小值反应了一天中的时间范围。

```
$ python datetime_time_minmax.py
Earliest  : 00:00:00
Latest    : 23:59:59.999999
Resolution: 0:00:00.000001
```

时间的最小取值是微秒，更精确的位数就被截断了。

```
import datetime

for m in [ 1, 0, 0.1, 0.6 ]:
    print '%02.1f :' % m, datetime.time(0, 0, 0, microsecond=m)
```

实际中，如果使用浮点型作为微秒参数，那么将产生一些警告信息。

```
$ python datetime_time_resolution.py
/home/cjj/python/pymotw/datetime_time_resolution.py:14: DeprecationWarning: integer argument expected


print '%02.1f :' % m, datetime.time(0, 0, 0, microsecond=m)


1.0 : 00:00:00.000001
0.0 : 00:00:00
0.1 : 00:00:00
0.6 : 00:00:00
```

日期

日期值可以由date类来表示，实例有年、月、日属性，使用data类的 `today()` 方法可以方便的表示出今天的日期。

```
import datetime

today = datetime.date.today()
print today
print 'ctime:', today.ctime()
print 'tuple:', today.timetuple()
print 'ordinal:', today.toordinal()
print 'Year:', today.year
print 'Mon :', today.month
print 'Day :', today.day
```

示例演示了今天日期的多种表示方法：

```
$ python datetime_date.py
2008-03-13
ctime: Thu Mar 13 00:00:00 2008
tuple: (2008, 3, 13, 0, 0, 0, 3, 73, -1)
ordinal: 733114
Year: 2008
Mon : 3
Day : 13
```

使用整数（从阳历的第1年1月1号开始）或者POSIX标准时间戳可以类实例。

```
import datetime
import time

o = 733114
print 'o:', o
print 'fromordinal(o):', datetime.date.fromordinal(o)
t = time.time()
print 't:', t
print 'fromtimestamp(t):', datetime.date.fromtimestamp(t)
```

示例显示了函数 `fromordinal()` 和 `fromtimestamp()` 返回了不同的结果。

```
$ python datetime_date_fromordinal.py
o: 733114
fromordinal(o): 2008-03-13
t: 1205436039.53
fromtimestamp(t): 2008-03-13
```

日期的最大和最小范围可以使用属性max和min来表示。

```
import datetime

print 'Earliest :', datetime.date.min
print 'Latest :', datetime.date.max
print 'Resolution:', datetime.date.resolution
```

一个日期的单位就是1天。

```
$ python datetime_date_minmax.py
Earliest  : 0001-01-01
Latest    : 9999-12-31
Resolution: 1 day, 0:00:00
```

对于一个存在的日期，可使用replace函数可以创建出一个新的日期实例。比如你可以改变年数，只保留月份和日。

```
import datetime

d1 = datetime.date(2008, 3, 12)
print 'd1:', d1

d2 = d1.replace(year=2009)
print 'd2:', d2
```

```
$ python datetime_date_replace.py
d1: 2008-03-12
d2: 2009-03-12
```

timedelta

除了 `replace()` 函数可以计算过去或者未来的时间，还可以使用timedelta类对日期值进行基本运算。通过timedelta可以加减一个日期来产生另外一个日期。timedelta中的内部值可以用天、秒和微秒来表示。

```
import datetime

print "microseconds:", datetime.timedelta(microseconds=1)
print "milliseconds:", datetime.timedelta(milliseconds=1)
print "seconds :", datetime.timedelta(seconds=1)
print "minutes :", datetime.timedelta(minutes=1)
print "hours :", datetime.timedelta(hours=1)
print "days :", datetime.timedelta(days=1)
print "weeks :", datetime.timedelta(weeks=1)
```

传递给构造器的中间值被转换为天、秒和微秒。

```
$ python datetime_timedelta.py

microseconds: 0:00:00.000001
milliseconds: 0:00:00.001000
seconds : 0:00:01
minutes : 0:01:00
hours : 1:00:00
days : 1 day, 0:00:00
weeks : 7 days, 0:00:00
```

比较

时间和日期值都可以通过标准的操作符来进行比较。

```
import datetime
import time

print 'Times:'
t1 = datetime.time(12, 55, 0)
print '\tt1:', t1
t2 = datetime.time(13, 5, 0)
print '\tt2:', t2
print '\tt1 < t2:', t1 < t2

print 'Dates:'
d1 = datetime.date.today()
print '\td1:', d1
d2 = datetime.date.today() + datetime.timedelta(days=1)
print '\td2:', d2
print '\td1 > d2:', d1 > d2
```

```
$ python datetime_comparing.py
Times:
     t1: 12:55:00
     t2: 13:05:00
     t1 < t2: True

Dates:
     d1: 2008-03-13
     d2: 2008-03-14
     d1 > d2: False
```

日期和时间组合

使用datetime类可以存储日期和时间的组合部分，类似于使用date。有多种方法可以创建datetime。

```
import datetime

print 'Now :', datetime.datetime.now()
print 'Today :', datetime.datetime.today()
print 'UTC Now:', datetime.datetime.utcnow()

d = datetime.datetime.now()
for attr in [ 'year', 'month', 'day', 'hour', 'minute', 'second', 'microsecond']:
    print attr, ':', getattr(d, attr)
```

同时，datetime实例拥有date和time对象的所有属性。

```
$ python datetime_datetime.py
Now : 2008-03-15 22:58:14.770074
Today : 2008-03-15 22:58:14.779804
UTC Now: 2008-03-16 03:58:14.779858
year : 2008
month : 3
day : 15
hour : 22
minute : 58
second : 14
microsecond : 780399
```

datetime类提供了一些类方法来创建新的实例，当然它也包含 `fromordinal()` 和 `fromtimestamp()` ，如果你已经有一个日期实例和时间实例，并需要创建datetime的话，combine()方法比较有用。

```
import datetime

t = datetime.time(1, 2, 3)
print 't :', t
d = datetime.date.today()
print 'd :', d
dt = datetime.datetime.combine(d, t)
print 'dt:', dt
```

```
$ python datetime_datetime_combine.py
t : 01:02:03
```

```
d : 2008-03-16
dt: 2008-03-16 01:02:03
```

格式化和解析

datetime对象的字符串表示方法默认使用的是ISO 8601格式（YYYY-MM-DDTHH:MM:SS.mmmmmm），使用 `strftime()` 可以产生其他格式，同样，如果你的输入值是用 `time.strptime()` 解析的时间戳，那么 `strptime()` 是一个合适的方法来把它转换为datetime实例。

```
import datetime

format = "%a %b %d %H:%M:%S %Y"

today = datetime.datetime.today()
print 'ISO :', today

s = today.strftime(format)
print 'strftime:', s

d = datetime.datetime.strptime(s, format)
print 'strptime:', d.strftime(format)


$ python datetime_datetime_strptime.py
ISO      : 2008-03-16 08:08:16.275134
strftime: Sun Mar 16 08:08:16 2008
strptime: Sun Mar 16 08:08:16 2008
```

时区

时区是由子类datetime.tzinfo来表示的，tzinfo是一个抽象的基类，你需要定义子类，并提供相应的方法去实现一些方法。很可惜，dateime不包含任何实际可用的实现，可以参考 文档 来获取一些示例。

参考

- PLEAC - Dates and Times
- WikiPedia: Proleptic Gregorian calendar

0.2.4 PyMOTW: linecache

- 模块：linecache
- 目的: 从文件或者导入模块中检索文本行，对结果采用缓冲来提高读文件的效率。
- python版本：1.4+

描述

python标准库处理python源文件中linecache模块被广泛的使用。缓冲的实现是读取文件的内容，并解析成行，保存在内存的字典中。API 根据索引返回列表中的请求行。在读取文件和寻找需要的行信息上可以节省一定时间。这对于从同一个文件中查询多行内容是非常有用的，比如为一个error report产生trackback。

示例

```
import linecache
import os
import tempfile
```

我们使用Lorem Ipsum generator产生的文本作为输入样例：

```
lorem = '''Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Vivamus eget elit. In posuere mi non risus. Mauris id quam posuere

lectus sollicitudin varius. Praesent at mi. Nunc eu velit. Sed augue
massa, fermentum id, nonummy a, nonummy sit amet, ligula. Curabitur
eros pede, egestas at, ultricies ac, pellentesque eu, tellus.

Sed sed odio sed mi luctus mollis. Integer et nulla ac augue convallis
accumsan. Ut felis. Donec lectus sapien, elementum nec, condimentum ac,
interdum non, tellus. Aenean viverra, mauris vehicula semper porttitor,
ipsum odio consectetuer lorem, ac imperdiet eros odio a sapien. Nulla
mauris tellus, aliquam non, egestas a, nonummy et, erat. Vivamus

sagittis porttitor eros.'''

# 创建一个临时文件
fd, temp_file_name = tempfile.mkstemp()

os.close(fd)
f = open(temp_file_name, 'wt')

try:
    f.write(lorem)
finally:
    f.close()
```

现在我们有了一个可用的临时文件，让我们更深入一步。从文件中读取的第5行是单一行。注意，在linecache中的行号是从1开始的。但是我们自己对字符串进行分割，那么索引号是从0开始。我们还需要从缓冲中返回的值进行过滤去除换行符。

```
# 原始形式和cache方式提取统一行。
# 注意linecache计算从1开始。
print 'SOURCE: ', lorem.split('\n')[4]
print 'CACHE : ', linecache.getline(temp_file_name, 5).rstrip()
```

下一步看下，如果我们需要的行为空将会发生什么。

```
# 新的一行中为空行。
print '\nBLANK : "%s"' % linecache.getline(temp_file_name, 6)
```

如果请求的行号超过了文件中有效行号的范围，那么linecache会返回一个空字符串。

```
# cache总会返回一个字符串，如果字符串为空即标明指代的行是不存在的。
not_there = linecache.getline(temp_file_name, 500)
print '\nNOT THERE: "%s" includes %d characters' %  (not_there, len(not_there))
```

即使这个文件不存在，模块也不会抛出任何异常。

```
# 即使linecache不能找到这个文件，Error也会被隐藏掉。
no_such_file = linecache.getline('this_file_does_not_exist.txt', 1)
print '\nNO FILE: ', no_such_file
```

虽然linecache模块经常用在输出tracebacks上，另一个重要特性是可以通过指定模块名在sys.path中寻找python模块源码。如果在当前路径中无法找到文件，那么linecache中的缓冲直接搜索sys.path中的模块。

```
# 利用内置子的sys.path，查询linecache中的模块名
module_line = linecache.getline('linecache.py', 3)
print '\nMODULE : ', module_line
```

示例输出

```
SOURCE:  eros pede, egestas at, ultricies ac, pellentesque eu, tellus.
CACHE :  eros pede, egestas at, ultricies ac, pellentesque eu, tellus.

BLANK : "
"

NOT THERE: "" includes 0 characters

NO FILE:

MODULE :  This is intended to read lines from modules imported -- hence if a filename
```

参考

- PyMOTW

0.2.5 PyMOTW: optparse

- 模块：optparse
- 目的：命令行参数解析，可以取代getopt
- python版本：2.3

描述

optparse是一个当前可选的命令行解析模块，它提供了一些在getopt中不含有的特性，如type conversion(类型转换)，option callbacks(参数回调)以及automatice help generation(自动化帮助生成)。本文没有详细介绍optparse的很多特性，但它可以帮助你在写命令行程序时能够快速入门。

创建一个OptionParser

optparser解析参数需要经过2个阶段。首先，构建OptionParser实例并配置相关的选项，然后填入一个参数序列并处理。

```
import optparse
parser = optparse.OptionParser()
```

通常，一旦分析器被建立，每一个选项需要明确的添加到parser中，并说明当命令行遇到相关的选项时需要如何处理。在构建OptionParser时也可以传入一个选项列表，但这种形式不经常使用。

定义选项

利用add_option()方法可以每次增加一个选项。在参数列表的开始，任何未命名的字符串参数都将被视为选项名。如果要为一个选项创建别名，比如为同一个选项增加一个短的或长的命名，那么简单传递同名字符串即可。

不同于getopt，只能分析选项，optparse是一个完整的选项分析库，Option(选项)可以被不同的方法处理，通过在add_option()方法中指定action(行为)参数。支持的行为包括存储参数(单独或作为列表的一部分)，当一个选项出现时(包括对布尔开关true/false的特殊处理)存储其常量值，计算一个选项出现的次数以及调用一个callback(回调函数)。

默认的行为是存储这个选项的参数。如果给定了type(类型)，那么在存储前，这个参数值将被转化成这个类型。如果给定了dest(目标参数)，那么当命令行参数被解析时，选项值被存储在该选项对象的dest中。

分析一个命令行

一旦所有的选项被定义好，命令行被作为一个参数字符串传递给parse_args()方法。一般，参数可以从sys.argv[1:]中得到，当然你可以传递自己的列表。选项处理时使用GNU/POSIX语法，因此，选项和参数值可以在参数序列中混合使用。

从parse_args()方法返回的是一个二维元组，包含一个optparse Values实例和在命令行中未被解析的参数列表。Values实例将选项值作为属性，如果你定义了一个选项的dest为“myoption”，可以通过 option.myoption访问该选项的值。

简单示例

如下一个简单例子有三个不同的选项，一个布尔选项(-a)，一个字符串选项(-b)和一个整型选项(-c)。

```
import optparse
parser = optparse.OptionParser()
parser.add_option('-a', action="store_true", default=False)
parser.add_option('-b', action="store", dest="b")
parser.add_option('-c', action="store", dest="c", type="int")
print parser.parse_args(['-a', '-bval', '-c', '3'])
```

命令行中选项解析的规则和getopt.gnu_getopt()一样，因此有两种方法传递单字符串选项的值，上述示例使用了两种方法-bval和-c val

```
$ python optparse_short.py
(<Values at 0xe29b8: {'a': True, 'c': 3, 'b': 'val'}>, [])
```

注意，c所关联的值的类型是整型，OptionParser在存储之前会转换成指定类型。不同于getopt，optparse处理长选项名时和短选项名是没有任何区别的。

```
parser = optparse.OptionParser()
parser.add_option('--noarg', action="store_true", default=False)
parser.add_option('--witharg', action="store", dest="witharg")
```

```
parser.add_option('--witharg2', action="store", dest="witharg2", type="int")
print parser.parse_args([ '--noarg', '--witharg', 'val', '--witharg2=3' ])
```

结果相同的:

```
$ python optparse_long.py
(<Values at 0xd3ad0: {'noarg': True, 'witharg': 'val', 'witharg2': 3}>, [])
```

与getopt的比较

如下实现一个与getopt之前示例相同功能的optparse例子

```
import optparse
import sys
print 'ARGV      :', sys.argv[1:]
parser = optparse.OptionParser()
parser.add_option('-o', '--output',
            dest="output_filename",
            default="default.out",
            )
parser.add_option('-v', '--verbose',
            dest="verbose",
            default=False,
            action="store_true",
            )
parser.add_option('--version',
            dest="version",
            default=1.0,
            type="float",
            )
options, remainder = parser.parse_args()
print 'VERSION   :', options.version
print 'VERBOSE   :', options.verbose
print 'OUTPUT    :', options.output_filename
print 'REMAINING :', remainder
```

注意，-o和–output选项是如何在同一时刻被定义的，命令行中可以使用任何一种选项。

```
$ python optparse_getoptcomparison.py -o output.txt
ARGV      : ['-o', 'output.txt']
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : output.txt
REMAINING : []
$ python optparse_getoptcomparison.py --output output.txt
ARGV      : ['--output', 'output.txt']
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : output.txt
REMAINING : []
```

另外，长选项名的唯一前缀也可以被使用。

```
$ python optparse_getoptcomparison.py --out output.txt
ARGV      : ['--out', 'output.txt']
VERSION   : 1.0
VERBOSE   : False
OUTPUT    : output.txt
REMAINING : []
```

Option Callbacks(选项回调)

除了直接为选项存储参数，另一种选择是定义callback function，当命令行中出现该选项时调用，选项的callbacks有4个参数，分别是引起callback的optparse.Option实例，命令行中的选项字符串，选项关联的参数值以及处理解析工作的optparse.OptionParser实例。

```
import optparse

def flag_callback(option, opt_str, value, parser):
    print 'flag_callback:'
    print '\toption:', repr(option)
    print '\topt_str:', opt_str
    print '\tvalue:', value
    print '\tparser:', parser
    return

def with_callback(option, opt_str, value, parser):
    print 'with_callback:'
    print '\toption:', repr(option)
    print '\topt_str:', opt_str
    print '\tvalue:', value
    print '\tparser:', parser
    return

parser = optparse.OptionParser()
parser.add_option('--flag', action="callback", callback=flag_callback)
parser.add_option('--with',
    action="callback",
    callback=with_callback,
    type="string",
    help="Include optional feature")
parser.parse_args(['--with', 'foo', '--flag'])
```

在这个例子中，–with选项被配置成处理字符串参数(当然，其他类型也是同样支持的)。

```
$ python optparse_callback.py
with_callback:
     option: <Option at 0x78b98: --with>
     opt_str: --with
     value: foo
     parser: <optparse.OptionParser instance at 0x78b48>
flag_callback:
     option: <Option at 0x7c620: --flag>
     opt_str: --flag
     value: None
     parser: <optparse.OptionParser instance at 0x78b48>
```

帮助信息

OptionParser自动为每个选项集合包含一个help选项，因此，用户在运行程序时在命令行输入–help来看介绍，帮助信息为所有选项指示它们是否需要传入一个参数，也可以通过在add_option()中定义帮助文本来为一个选项定义更多的描述。

```
parser = optparse.OptionParser()
parser.add_option('--no-foo', action="store_true",
    default=False,
    dest="foo",
    help="Turn off foo",
)
parser.add_option('--with', action="store", help="Include optional feature")
parser.parse_args()
```

选项按字母顺序显示，别名显示在同一行，当选项带有一个参数时，dest值将作为参数名字出现在help输出中，帮助信息将出现在这列中。

```
$ python optparse_help.py --help
Usage: optparse_help.py [options]

Options:
-h, --help   show this help message and exit
--no-foo     Turn off foo
--with=WITH  Include optional feature
```

利用nargs选项可以配置callbacks接收多个参数。

```
def with_callback(option, opt_str, value, parser):
    print 'with_callback:'
    print '\toption:', repr(option)
    print '\topt_str:', opt_str
    print '\tvalue:', value
    print '\tparser:', parser
    return

parser = optparse.OptionParser()
parser.add_option('--with',
    action="callback",
    callback=with_callback,
    type="string",
    nargs=2,
    help="Include optional feature")
parser.parse_args(['--with', 'foo', 'bar'])
```

在这个例子中，参数作为一个元组传递给callback function的value参数。

```
$ python optparse_callback_nargs.py
with_callback:
     option: <Option at 0x7c4e0: --with>
     opt_str: --with
     value: ('foo', 'bar')
     parser: <optparse.OptionParser instance at 0x78a08>
```

0.2.6 PyMOTW: os

- 模块：os
- 目的：为访问操作系统的特定属性提供方便。
- python版本：1.4+

描述

os模块提供了对特定平台模块(如posix，nt，mac)的封装，函数提供的api在很多平台上都可以相同使用，所以使用os模块会变得很方便。但不是所有函数在所有平台上都可用，比如在本文中到的一些管理函数在windows上无法使用。

在Python文档中os模块的副标题是"操作系统混合接口"，模块包含的大部分函数用于创建和管理活动进程和文件系统（文件和目录），当然除此之外还有其它一些函数。本文中，我们对如何获取和设置进程参数来进行讨论。

Note: 以下示例代码有的只能在linux平台上工作。

属主处理

首先讨论用来检查和改变进程属主id的函数。在守护进程和特殊的系统程序需要改变执行权限而不使用root情况下这往往是非常有用的。这里我不会太过详细的解释linux的安全，进程属主等具体含义，这些可以见参考中的文章来获得更详细的介绍。

我们给出一段脚本来获取一个进程的有效用户和组信息，然后改变这些有效值。这类似于一个守护进程在系统启动时以root身份启动加载，然后降低权限并作为一个普通用户运行。如果你下载示例并试运行，你可以设置user相应的值为TEST_GID和TEST_UID。

```
import os

TEST_GID=501
TEST_UID=527

def show_user_info():
     print 'Effective User  :', os.geteuid()
     print 'Effective Group :', os.getegid()
     print 'Actual User    :', os.getuid(), os.getlogin()
     print 'Actual Group   :', os.getgid()
     print 'Actual Groups  :', os.getgroups()
     return

print 'BEFORE CHANGE:'
show_user_info()
print
try:
     os.setegid(TEST_GID)
except OSError:
     print 'ERROR: Could not change effective group.  Re-run as root.'
else:
     print 'CHANGED GROUP:'
     show_user_info()
     print

try:
     os.seteuid(TEST_UID)
```

```
except OSError:
    print 'ERROR: Could not change effective user.  Re-run as root.'
else:
    print 'CHANGE USER:'
    show_user_info()
    print
```

当我运行在DELL D630 Ubuntu上时，得到的结果如下：

```
~ 16:51:33> ./a.py
BEFORE CHANGE:
Effective User  : 1000
Effective Group : 1000
Actual User     : 1000 cjj
Actual Group  : 1000
Actual Groups   : [4, 20, 24, 25, 29, 30, 44, 46, 104, 108, 110, 115, 117, 1000]

ERROR: Could not change effective group.  Re-run as root.
ERROR: Could not change effective user.  Re-run as root.
```

注意，当我使用非root运行时，值未被改变，我所启动的进程不可以改变他们自身有效的属性。如果我试图设置其他的用户名和组id，那么会抛出OSError异常。

下面，我们以root权限来运行这段脚本:

```
~ 16:51:10> sudo ./a.py
[sudo] password for cjj:
BEFORE CHANGE:
Effective User  : 0
Effective Group : 0
Actual User     : 0 cjj
Actual Group  : 0
Actual Groups   : [0]

CHANGED GROUP:
Effective User  : 0
Effective Group : 501
Actual User     : 0 cjj
Actual Group  : 0
Actual Groups   : [0]

CHANGE USER:
Effective User  : 527
Effective Group : 501
Actual User     : 0 cjj
Actual Group  : 0
Actual Groups   : [0]
```

在这个例子中，如果我们以root权限运行，那么我们可以改变这个进程的用户和组属性。一旦我们改变了UID，那么进程将受这个用户的权限限制，非root用户是无法改变有效用户组，所以首先我们需要改变用户组，然后再改变用户名。

除了查找和改变进程属主，还有其他函数可以获取当前进程和父进程的id，查找和改变其进程用户组和会话id，与控制终端id是一样的。在你编写复杂程序（如自己的终端命令行程序）中使用这些函数可以帮助你在进程之间传递信号。

环境处理

通过os模块，你的程序可以访问的另一个操作系统特性是系统环境。通过os.environ和os.getenv()可以访问在环境中设置的变量字符串。环境变量常用来作为配置像搜索路径，文件路径、调试标志的值。下面的示例检索了一个环境变量，并且通过子进程改变了这个值。

```
print 'Initial value:', os.environ.get('TESTVAR', None)
print 'Child process:'
os.system('echo $TESTVAR')

os.environ['TESTVAR'] = 'THIS VALUE WAS CHANGED'

print
print 'Changed value:', os.environ['TESTVAR']
print 'Child process:'
os.system('echo $TESTVAR')

del os.environ['TESTVAR']

print
print 'Removed value:', os.environ.get('TESTVAR', None)
print 'Child process:'
os.system('echo $TESTVAR')
```

os.environ对象遵循标准的Python映射API以便检索和设置值。 os.environ值的改变将被输出到子进程中。

```
$ python os_environ_example.py
Initial value: None
Child process:


Changed value: THIS VALUE WAS CHANGED
Child process:
THIS VALUE WAS CHANGED

Removed value: None
Child process:
```

工作目录处理

在操作系统的文件系统结构中有一个概念是“当前工作目录”。在文件系统中，当前进程在访问用相对路径表示的文件时，就把这个目录当作默认目录位置。

```
print 'Starting:', os.getcwd()
print os.listdir(os.curdir)

print 'Moving up one:', os.pardir
os.chdir(os.pardir)

print 'After move:', os.getcwd()
print os.listdir(os.curdir)
```

注意os.curdir()和os.pardir()是指向当前目录和父目录的一种快捷方式。结果很显然：

```
Starting: /Users/dhellmann/Documents/PyMOTW/PyMOTW/os
['.svn', '__init__.py', 'os_cwd_example.py', 'os_environ_example.py',
'os_process_id_example.py', 'os_process_user_example.py']
Moving up one: ..
After move: /Users/dhellmann/Documents/PyMOTW/PyMOTW
['.svn', '__init__.py', 'bisect', 'ConfigParser', 'fileinput', 'linecache',
'locale', 'logging', 'os', 'Queue', 'StringIO', 'textwrap']
```

后续...

这里我们仅介绍了os模块中查找和设置进程参数的一些函数。下一次，我们将介绍os模块来管理文件系统对象。

参考

- Python Reference Manual, Process Parameters
- Speaking UNIX, Part 8: UNIX processes
- geteuid
- getsid
- setpgrp

0.2.7 PyMOTW: os(2)

描述

上一部分，我们讨论了进程参数，现在我们讨论一下os模块提供的输入/输出特性。

管道

os模块提供了一些函数，这些函数利用管道来管理子进程的IO操作。这些函数的工作方式基本相同，但根据输入/输出的需求类型返回不同的文件句柄。相对于2.4版本中的 subprocess 模块这些函数是过时了，但这是一个很好的机会，你可以在已有的代码中看到它们。

管道中经常使用的是popen()函数，它创建一个新的进程用于运行给定的命令并且根据模式选项附加给这个进程一个单一的输入输出数据流 。虽然在Windows中可以使用popen()，但以下例子假设以Unix shell方式运行，其中流的概念也是unix技术。

stdin：进程（文件描述符0）的标准输入流，对于这个进程来说是可读的，通常指终端输入。

stdout：进程（文件描述符1）的标准输出流，对于这个进程来说是可写的，通常用于给用户显示非错误信息。

stderr：进程（文件描述符2）的标准错误流，对于这个进程来说是可写的，通常用于传递错误信息。

```
import os

print '\npopen, read:'
pipe_stdout = os.popen('echo "to stdout"', 'r')
try:
```

```
    stdout_value = pipe_stdout.read()
finally:
    pipe_stdout.close()
print '\tstdout:', repr(stdout_value)

print '\npopen, write:'
pipe_stdin = os.popen('cat -', 'w')
try:
    pipe_stdin.write('\tstdin: to stdin\n')
finally:
    pipe_stdin.close()


popen, read:
     stdout: 'to stdout\n'

popen, write:
     stdin: to stdin
```

从子进程的流中读取或者写入的方法是比较受限的，popen提供了额外的流，如stdin、stdout、stderr来以便使用。

比如，popen2()函数返回一个与子进程标准输入绑定的只写流和一个与子进程标准输出绑定的只读流。

```
print '\npopen2:'
pipe_stdin, pipe_stdout = os.popen2('cat -')
try:
    pipe_stdin.write('through stdin to stdout')
finally:
    pipe_stdin.close()

try:
    stdout_value = pipe_stdout.read()
finally:
    pipe_stdout.close()
print '\tpass through:', repr(stdout_value)
```

这个简单例子解释了双向通信方式，从stdin写入的值被cat命令读取（'-'参数的作用），然后由stdout输出。显然，一个复杂的进程通过管道可以来回传递其它类型的信息，甚至是序列化对象。

```
popen2:
     pass through: 'through stdin to stdout'
```

有些情况下，希望同时访问stdout和stderr，stdout常用于输出信息，stderr常用于抛出错误。因此分别读取他们可以减少解析错误消息的复杂度，而popen3函数返回一个新进程的3个流stdin、stdout、stderr。

```
print '\npopen3:'
pipe_stdin, pipe_stdout, pipe_stderr = os.popen3('cat -; echo ";to stderr" 1>&2')
try:
    pipe_stdin.write('through stdin to stdout')
finally:
    pipe_stdin.close()
try:
    stdout_value = pipe_stdout.read()
finally:
    pipe_stdout.close()
print '\tpass through:', repr(stdout_value)
```

```
try:
    stderr_value = pipe_stderr.read()
finally:
    pipe_stderr.close()
print '\tstderr:', repr(stderr_value)
```

注意，我们需要分别读取和关闭这些流，在处理多进程的IO中，还涉及到流程控制和排序，I/O即为缓冲器，如果想读取流中的所有数据，那么子进程必须关闭这个流来表示文件的结束，更多信息可以参考Python库文档 Flow Control Issues

```
popen3:
     pass through: 'through stdin to stdout'
     stderr: ';to stderr\n'
```

最后，popen4()返回两个流，stdin和stdout/stderr的组合，这对于命令的结果需要被记录，但不需要解析是很有用的。

```
print '\npopen4:'
pipe_stdin, pipe_stdout_and_stderr = os.popen4('cat -; echo ";to stderr" 1>&2')
try:
    pipe_stdin.write('through stdin to stdout')
finally:
    pipe_stdin.close()
try:
    stdout_value = pipe_stdout_and_stderr.read()
finally:
    pipe_stdout.close()
print '\tcombined output:', repr(stdout_value)


popen4:
     combined output: 'through stdin to stdout;to stderr\n'
```

另外，除了接收简单的字符串命令来传递给shell解析，popen2()、popen3()、popen4()函数同样接收字符串序列（命令，加参数），这种情况中，参数不是传递给shell的。

```
print '\npopen2, cmd as sequence:'
pipe_stdin, pipe_stdout = os.popen2(['cat', '-'])
try:
    pipe_stdin.write('through stdin to stdout')
finally:
    pipe_stdin.close()
try:
    stdout_value = pipe_stdout.read()
finally:
    pipe_stdout.close()
print '\tpass through:', repr(stdout_value)


popen2, cmd as sequence:
     pass through: 'through stdin to stdout'
```

后续

下次，我们将讨论如何来控制文件描述符。

参考

- Unix Concepts for more discussion of stdin, stdout, and stderr
- File Object Creation with the os module
- subprocess
- Flow Control Issues

0.2.8 PyMOTW: os(3)

描述

前面讲述了如何来处理进程参数和输入/输出，本周我将探讨一些操作文件和目录的函数。

文件描述符

os模块中包含了一些函数集用于处理底层的文件描述符（当前进程打开属主文件所使用的整型），相比file()对象来说这些是更底层的API，在本文中将不会解释什么是文件描述符，它通常可以很好的和file()对象协同工作，更多细节可以参考 这里 来了解如何使用文件描述符。

文件系统权限

os.access()可以测试一个进程对一个文件是否有可访问权限。

```
import os

print 'Testing:', __file__
print 'Exists:', os.access(__file__, os.F_OK)
print 'Readable:', os.access(__file__, os.R_OK)
print 'Writable:', os.access(__file__, os.W_OK)
print 'Executable:', os.access(__file__, os.X_OK)
```

这个结果将取决于你如何来运行这个示例程序，可能会显示如下：

```
$ python os_access.py
Testing: os_access.py
Exists: True
Readable: True
Writable: True
Executable: False
```

os.access()模块包含了2个特殊的含义，首先，在实际使用open()函数之前使用os.access()函数来判断一个文件是否可访问是没有意义的。这里有个小事实，在函数的两次调用之间可能会改变文件的权限。另外一个含义是该函数适合于大部分扩展的POSIX许可语义的网络文件系统。文件系统对于一个进程对文件有访问权限的POSIX调用会做出响应，在调用open()时，因为某些原因没有通过POSIX的调用测试，那么会报告失败。总之，最好时在特定的模式中使用open()，如果出现错误还可以捕获IOError异常。

如果想获得更多关于文件的信息，可以查阅stat()或者os.lstat（如果你查看的文件一个动态链接的话）。

```
import os
import sys
import time
```

```
if len(sys.argv) == 1:
  filename = __file__
  else:
    filename = sys.argv[1]

stat_info = os.stat(filename)

print 'os.stat(%s):' % filename
print '\tSize:', stat_info.st_size
print '\tPermissions:', oct(stat_info.st_mode)
print '\tOwner:', stat_info.st_uid
print '\tDevice:', stat_info.st_dev
print '\tLast modified:', time.ctime(stat_info.st_mtime)
```

再次申明，你得到的结果将取决于你运行的方式，可以尝试向os_stat.py传递不同的文件名看看。

```
$ python os_stat.py
os.stat(os_stat.py):
    Size: 1547
    Permissions: 0100644
    Owner: 527
    Device: 234881026
    Last modified: Sun Jun 10 08:13:26 2007
```

在Unix类型系统上，文件权限可以由chmod()来修改，以整形形式传递。形式值可以用stat模块的常量值来b表示。以下示例了如何来触发用户的可执行权限位。

```
import os
import stat

# 使用stat来获取权限设置的时间
existing_permissions = stat.S_IMODE(os.stat(__file__).st_mode)

if not os.access(__file__, os.X_OK):
  print 'Adding execute permission'
  new_permissions = existing_permissions | stat.S_IXUSR
else:
  print 'Removing execute permission'
  # 使用xor来删除用户的可执行权限
  new_permissions = existing_permissions ^ stat.S_IXUSR

os.chmod(__file__, new_permissions)
```

运行该脚本前假设你有修改文件模式的权限。

```
$ python os_stat_chmod.py
Adding execute permission
$ python os_stat_chmod.py
Removing execute permission
```

目录

同样提供了一些处理文件系统中目录的函数，包括创建内容列表和删除它们。

```
import os

dir_name = 'os_directories_example'

print 'Creating', dir_name
os.makedirs(dir_name)

file_name = os.path.join(dir_name, 'example.txt')
print 'Creating', file_name
f = open(file_name, 'wt')
try:
  f.write('example file')
finally:
  f.close()

print 'Listing', dir_name
print os.listdir(dir_name)

print 'Cleaning up'
os.unlink(file_name)
os.rmdir(dir_name)
```

```
$ python os_directories.py
Creating os_directories_example
Creating os_directories_example/example.txt
Listing os_directories_example
['example.txt']
Cleaning up
```

有2个函数集用来创建和删除目录，当使用os.mkdir()创建一个新的目录时，其父目录必须存在。当使用os.rmdir()来删除一个目录时候，那么只有目录树的叶子节点（目录的最后一级）可以被删除。相比下，os.makedirs()和os.removedirs()可以操作当前路径下的所有目录，os.makedirs()可以创建路径不存在的目录，os.removedirs()可以删除包含父目录的子目录（当然前提有这个权限）。

符号链接

很多文件系统和平台都支持它，同样有一些函数可以用来处理它们。

```
import os, tempfile

link_name = tempfile.mktemp()

print 'Creating link %s->%s' % (link_name, __file__)
os.symlink(__file__, link_name)

stat_info = os.lstat(link_name)
print 'Permissions:', oct(stat_info.st_mode)

print 'Points to:', os.readlink(link_name)

# Cleanup
os.unlink(link_name)
```

虽然os中包含了os.temparm()来创建临时文件，当时相比tempfile模块还不够安全，在使用中可能会产生RuntimeWarning信息，更好的方法使用tempfile模块。

```
$ python os_symlinks.py
Creating link /tmp/tmpRxRiHn->os_symlinks.py
Permissions: 0120755
Points to: os_symlinks.py
```

访问目录树

os.walk()可以递归遍历一个目录,对于每一个目录，可以产生一个包含目录路径、当前路径的子目录列表，以及在子目录中的文件，以下示例展示了一个遍历目录的简单方法:

```
import os, sys

# 如果没有给定目录列表，那么将使用/tmp
if len(sys.argv) == 1:
  root = '/tmp'
else:
  root = sys.argv[1]

for dir_name, sub_dirs, files in os.walk(root):
  print '\n', dir_name
  # 每个子目录名以"/"结尾
  sub_dirs = [ '%s/' % n for n in sub_dirs ]
  # 子目录内容的组合
  contents = sub_dirs + files
  contents.sort()
  # 显示内容
  for c in contents:
    print '\t%s' % c
```

```
$ python os_walk.py

/tmp
   .KerberosLogin-0--1074266944 (inited,root,local)/
   .KerberosLogin-527-4839472 (inited,gui,tty,local)/
   527/
   cs_cache_lock_527
   cs_cache_lock_92
   emacs527/
   fry.log
   hsperfdata_dhellmann/
   objc_sharing_ppc_4294967294
   objc_sharing_ppc_527
   objc_sharing_ppc_92
   svn.arg.1835159
   var_backups/

/tmp/.KerberosLogin-527-4839472 (inited,gui,tty,local)
   KLLCCache.lock

/tmp/527
   /tmp/emacs527
   server
   /tmp/hsperfdata_dhellmann
   976
```

```
/tmp/var_backups
   infodir.bak
   local.nidump
```

后续

下次，我们讨论os模块中创建和管理进程的函数。

参考

- Working with Files and Directories
- tempfile module

0.2.9 PyMOTW: os(4)

描述

这周,我总结整个os模块(但保留os.path的内容作为将来独立的一篇)并讨论一些有利于处理多进程的函数.我在part2中已经介绍了管道的使用, 这周我们来看下system(),fork(),exec()这3个函数和他们之间的关系.

申明

这里的许多函数都有可移植性限制.可以查看subprocess模块以获得一种更一致的平台独立的处理进程方式.

运行外部命令

最简单的运行一条单独命令,没有一点交互的方式是使用os.system(). 他获取一个字符串,这个字符串就是一将被命令行执行的命令,通过一个shell中的子进程来执行.

```
import os

# 简单的一条命令
os.system('ls -l')
```

```
$ python os_system_example.py
total 168
-rw-r--r-- 1 dhellman dhellman 0 May 27 06:58 __init__.py
-rw-r--r-- 1 dhellman dhellman 1391 Jun 10 09:36 os_access.py
-rw-r--r-- 1 dhellman dhellman 1383 May 27 09:23 os_cwd_example.py
-rw-r--r-- 1 dhellman dhellman 1535 Jun 10 09:36 os_directories.py
-rw-r--r-- 1 dhellman dhellman 1613 May 27 09:23 os_environ_example.py
-rw-r--r-- 1 dhellman dhellman 2816 Jun 3 08:34 os_popen_examples.py
-rw-r--r-- 1 dhellman dhellman 1438 May 27 09:23 os_process_id_example.py
-rw-r--r-- 1 dhellman dhellman 1887 May 27 09:23 os_process_user_example.py
-rw-r--r-- 1 dhellman dhellman 1545 Jun 10 09:36 os_stat.py
-rw-r--r-- 1 dhellman dhellman 1638 Jun 10 09:36 os_stat_chmod.py
-rw-r--r-- 1 dhellman dhellman 1452 Jun 10 09:36 os_symlinks.py
-rw-r--r-- 1 dhellman dhellman 1279 Jun 17 12:17 os_system_example.py
-rw-r--r-- 1 dhellman dhellman 1672 Jun 10 09:36 os_walk.py
```

由于命令是直接被传递到处理shell中,所以它可以包含shell语法,比如通配符或环境变量:

```
# 有shell扩展的命令
os.system('ls -l $HOME')
```

```
total 40
-rwx------ 1 dhellman dhellman 1328 Dec 13 2005 %backup%~
drwx------ 11 dhellman dhellman 374 Jun 17 12:11 Desktop
drwxr-xr-x 15 dhellman dhellman 510 May 27 07:50 Devel
drwx------ 29 dhellman dhellman 986 May 31 17:01 Documents
drwxr-xr-x 45 dhellman dhellman 1530 Jun 17 12:12 DownloadedApps
drwx------ 55 dhellman dhellman 1870 May 22 14:53 Library
drwx------ 8 dhellman dhellman 272 Mar 4 2006 Movies
drwx------ 10 dhellman dhellman 340 Feb 14 10:54 Music
drwx------ 12 dhellman dhellman 408 Jun 17 01:00 Pictures
drwxr-xr-x 5 dhellman dhellman 170 Oct 1 2006 Public
drwxr-xr-x 15 dhellman dhellman 510 May 12 15:19 Sites
drwxr-xr-x 4 dhellman dhellman 136 Jan 23 2006 iPod
-rw-r--r-- 1 dhellman dhellman 105 Mar 7 11:48 pgadmin.log
drwxr-xr-x 3 dhellman dhellman 102 Apr 29 16:32 tmp
```

除非你是直接在后台中运行这命令, 不然的话,直到命令执行完毕,调用 `os.system()` 的程序都会处于阻断状态. 子进程中的标准输入,输出,和错误输出默认被绑定到调用者的合适的流中. 但是也可以通过shell语法重定向到其他地方.

```
import os
import time

print 'Calling...'
os.system('date; (sleep 3; date) &')

print 'Sleeping...'
time.sleep(5)
```

这就是shell的魔力,尽管还有更好的实现方式.

```
$ python os_system_background.py
Calling...
Sun Jun 17 12:27:20 EDT 2007
Sleeping...
Sun Jun 17 12:27:23 EDT 2007
```

使用os.fork()创建进程

符合POSIX标准的函数 `fork()` 和 `exec*()` (在Mac OS X, Linux和其他类UNIX系统上可用)通过os模块都是可用的. 很多书已经很全面可靠的描述了这些函数的使用,所以检查你的库手册,或者去书店寻找进一步细节.

创建一个新进程作为当前进程的一个复本,可以使用 `os.fork()`:

```
pid = os.fork()

if pid:
    print 'Child process id:', pid
else:
    print 'I am the child'
```

每次运行这个事例代码时,你的输出变化给予你系统的当前状态, 但是它应该看起来像如下:

```
$ python os_fork_example.py
Child process id: 5883
I am the child
```

当fork之后, 你结束这两个运行着相同代码的进程. 可以检查返回值来直到你在哪个进程中. 如果它是0, 表示你在子进程中,如果不是0, 则表示你在父进程中,它返回的值是其子进程的进程id.

对于父进程来说,发送给子进程信号是有必要的. 这个的设置有点复杂, 使用signal模块, 让我们通过具体代码来描述其使用吧. 首先我们定义一个信号处理句柄, 以便在收到相应信号时调用.

```
import os
import signal
import time

def signal_usr1(signum, frame):
    pid = os.getpid()
    print 'Received USR1 in process %s' % pid
```

然后我们创建子进程, 并在父进程中,通过 `os.kill()` 发送一个USR1信号之前暂停一段时间.这短的暂停让子进程有时间去设置信号处理句柄.

```
print 'Forking...'
child_pid = os.fork()
if child_pid: ## 这个是父进程执行的代码
    print 'PARENT: Pausing before sending signal...'
    time.sleep(1)
    print 'PARENT: Signaling %s' % child_pid
    os.kill(child_pid, signal.SIGUSR1)
```

在子进程中,我们设置信号处理句柄后睡眠一段时间来让父进程有时间去发送信号给我们:

```
else:
    print 'CHILD: Setting up signal handler'
    signal.signal(signal.SIGUSR1, signal_usr1)
    print 'CHILD: Pausing to wait for signal'
    time.sleep(5)
```

当然，在实际的程序中,你也可能不需要(不想)调用sleep。

```
$ python os_kill_example.py
Forking...
PARENT: Pausing before sending signal...
CHILD: Setting up signal handler
CHILD: Pausing to wait for signal
PARENT: Signaling 6053
Received USR1 in process 6053
```

正如你所看到的, 一个简单的处理子进程各自行为的方式是简单 `fork()` 函数的返回值并使用if分支实现. 对于更复杂的行为, 就需要更多的分离(独立)的代码,而不是简单的分支. 在其他的例子中,你可以使用一个已经封装好的程序. 对于这两种情况,你可以使用 `os.exec*()` 系列函数来运行其他程序. 当你"exec"一个程序, 程序中的代码会代替你进程中已存在的那些代码.

```
child_pid = os.fork()
if child_pid:
    os.waitpid(child_pid, 0)
else:
    os.execlp('ls', 'ls', '-l', '/tmp/') ## 执行多个子进程


$ python os_exec_example.py

total 40
drwxr-xr-x 2 dhellman wheel 68 Jun 17 14:35 527
prw------- 1 root wheel 0 Jun 15 19:24 afpserver_PIPE
drwx------ 3 dhellman wheel 102 Jun 17 12:13 emacs527
drwxr-xr-x 2 dhellman wheel 68 Jun 16 05:01 hsperfdata_dhellmann
-rw------- 1 nobody wheel 12 Jun 17 13:55 objc_sharing_ppc_4294967294
-rw------- 1 dhellman wheel 144 Jun 17 14:32 objc_sharing_ppc_527
-rw------- 1 security wheel 24 Jun 17 07:09 objc_sharing_ppc_92
drwxr-xr-x 4 dhellman dhellman 136 Jun 8 03:16 var_backups
```

有很多 `exec*()` 的变种, 它们依赖于你可能使用的参数, 如,你是否想要路径和父进程的环境变量都被复制到子进程中,等等. 细节可参见库文档.

对于所有变种, 它们的第一个参数是一个路径或者文件名, 剩下的参数控制如何运行相应程序. 它们要么作为命令行参数被传递, 要么覆盖进程"环境"(可查看 `os.environ` 和 `os.getenv`).

等待一个子进程

假设说你使用了多个进程来突破Python的线程限制和 `GIL`.如果你开始了多个进程来运行各自的任务, 你希望在开始新的进程之前等待其中一个或多个的结束, 以此来避免服务器的超载. 这里有一些使用 `wait()` 和相关函数来实现它的不同方法.

如果你不关心, 或者你已经知道, 哪个可能会首先退出 `os.wait()` 的子进程,并且这个子进程会尽快的返回任何存在:

```
import os
import sys
import time

for i in range(3):
    print 'PARENT: Forking %s' % i
    worker_pid = os.fork()
    if not worker_pid:
        print 'WORKER %s: Starting' % i
        time.sleep(2 + i)
        print 'WORKER %s: Finishing' % i
        sys.exit(i)

 for i in range(3):
     print 'PARENT: Waiting for %s' % i
     done = os.wait()
     print 'PARENT:', done
```

Note: `os.wait()` 的返回值是包含进程号和退出状态(一个16位的数字, 它的低字节是一个杀死该进程的信号数字, 它的高字节是退出状态)的一个元组.

```
$ python os_wait_example.py

PARENT: Forking 0
PARENT: Forking 1
PARENT: Forking 2
PARENT: Waiting for 0
WORKER 0: Starting
WORKER 1: Starting
WORKER 2: Starting
WORKER 0: Finishing
PARENT: (6501, 0)
PARENT: Waiting for 1
WORKER 1: Finishing
PARENT: (6502, 256)
PARENT: Waiting for 2
WORKER 2: Finishing
PARENT: (6503, 512)
```

如果你想等待一个特定的进程, 可以使用 `os.waitpid()`.

```
import os
import sys
import time

workers = []
for i in range(3):
    print 'PARENT: Forking %s' % i
    worker_pid = os.fork()
    if not worker_pid:
        print 'WORKER %s: Starting' % i
        time.sleep(2 + i)
        print 'WORKER %s: Finishing' % i
        sys.exit(i)
    workers.append(worker_pid)

for pid in workers:
    print 'PARENT: Waiting for %s' % pid
    done = os.waitpid(pid, 0)
    print 'PARENT:', done
```

```
$ python os_waitpid_example.py
PARENT: Forking 0
WORKER 0: Starting
PARENT: Forking 1
WORKER 1: Starting
PARENT: Forking 2
WORKER 2: Starting
PARENT: Waiting for 6547
WORKER 0: Finishing
PARENT: (6547, 0)
PARENT: Waiting for 6548
WORKER 1: Finishing
PARENT: (6548, 256)
PARENT: Waiting for 6549
WORKER 2: Finishing
PARENT: (6549, 512)
```

`wait3()` 和 `wait4()` 函数也是类似的方式, 但它们返回更多关于子进程的细节信息,如进程号, 退出状态, 资源使用情况等.

Spawn（孵化）

方便起见, `os.spawn*()` 系列函数将 `fork()` 和 `exec*()` 调用写在一条语句中:

```
os.spawnlp(os.P_WAIT, 'ls', 'ls', '-l', '/tmp/')
```

```
$ python os_exec_example.py
total 40
drwxr-xr-x 2 dhellman wheel 68 Jun 17 14:35 527
prw------- 1 root wheel 0 Jun 15 19:24 afpserver_PIPE
drwx------ 3 dhellman wheel 102 Jun 17 12:13 emacs527
drwxr-xr-x 2 dhellman wheel 68 Jun 16 05:01 hsperfdata_dhellmann
-rw------- 1 nobody wheel 12 Jun 17 13:55 objc_sharing_ppc_4294967294
-rw------- 1 dhellman wheel 144 Jun 17 14:32 objc_sharing_ppc_527
-rw------- 1 security wheel 24 Jun 17 07:09 objc_sharing_ppc_92
drwxr-xr-x 4 dhellman dhellman 136 Jun 8 03:16 var_backups
```

结论

还有其他很多在处理多进程时需要考虑的东西,比如,信号处理, 多进程文件读写等.所有这些话题都在参考书目(如 Advanced Programming in the UNIX(R) Environment)中有讲述.

参考

- Delve into UNIX process creation

0.2.10 PyMOTW: pickle & cPickle

Python对象序列化

- 模块：pickle 和 cPickle
- 目的: Python对象序列化
- python版本：pickle至少1.4, cPickle 至少1.5

描述

pickle模块可以实现任意的Python对象转换为一系列字节(即序列化对象)的算法。这些字节流可以被传输或存储，接着也可以重构为一个和原先对象具有相同特征的新对象。

cPickle模块实现了同样的算法，但它是用c而不是python。因此，它比python实现的快上好几倍，但是不允许使用者去继承Pickle。如果继承对于你的使用不是很重要，那么你大可以使用cPickle。

Note: pickle的文档清晰的表明它不提供安全保证。所以慎用pickle来作为内部进程通信或者数据存储，也不要相信那些你不能验证安全性的数据。

例子

第一个pickle示例是将一个数据结构编码为一个字符串，然后将其输出到控制台。

```
try:
   import cPickle as pickle
except:
   import pickle
   import pprint
```

我们首先尝试导入cPickle，并指定别名为"pickle"。如果因为某种原因导入pickle失败，我们则导入由Python实现的pickle模块。如果cPickle是可用的，会给我们带来更快的实现，但如果不可用，也会有正确的实现。

接下来，我们定义一个完全由基本类型组成的数据结构。任何类的实例都可被pickle，这会在下一个例子中表述。我选择基本数据类型以便更简单的示范。

```
data = [ { 'a':'A', 'b':2, 'c':3.0 } ]
print 'DATA:',
pprint.pprint(data)
```

现在我们就可以使用pickle.dumps()来创建数据值的字符串表示。

```
data_string = pickle.dumps(data)
print 'PICKLE:', data_string
```

默认情况下，pickle仅使用ASCII字符。也可以使用高效的二进制格式。但这些示例依然使用了ASCII格式 。

```
$ python pickle_string.py
DATA:[{'a': 'A', 'b': 2, 'c': 3.0}]
PICKLE: (lp1
(dp2
S'a'
S'A'
sS'c'
F3
sS'b'
I2
sa.
```

一旦数据被序列化，你就可以把他写入到文件、socket、管道等等中。之后你可以读取这个文件，unpickle这些数据来构造具有相同值的新对象。

```
data1 = [ { 'a':'A', 'b':2, 'c':3.0 } ]
print 'BEFORE:',
pprint.pprint(data1)

data1_string = pickle.dumps(data1)

data2 = pickle.loads(data1_string)
print 'AFTER:',
pprint.pprint(data2)

print 'SAME?:', (data1 is data2)
print 'EQUAL?:', (data1 == data2)
```

正像你看到的那样，新构造的对象等于原来的对象，但他们又不是相同的对象。这里不足为奇。

```
$ python pickle_unpickle.py
BEFORE:[{'a': 'A', 'b': 2, 'c': 3.0}]
AFTER:[{'a': 'A', 'b': 2, 'c': 3.0}]
SAME?: False
EQUAL?: True
```

pickle除了提供dumps()和loads()，还提供非常方便的函数用于操作类文件流。支持同时写多个对象到同一个流中，然后在不知道有多少个对象或不知道它们有多大时，能够从这个流中读取多个对象也是可能的，

```
try:
   import cPickle as pickle
except:
   import pickle
import pprint
from StringIO import StringIO

class SimpleObject(object):

   def __init__(self, name):
      self.name = name
      l = list(name)
      l.reverse()
      self.name_backwards = ''.join(l)
      return

data = []
data.append(SimpleObject('pickle'))
data.append(SimpleObject('cPickle'))
data.append(SimpleObject('last'))

# 使用StringIO来模拟一个文件
out_s = StringIO()

# 写入到流中
for o in data:
   print 'WRITING: %s (%s)' % (o.name, o.name_backwards)
   pickle.dump(o, out_s)
   out_s.flush()

# 设置可读流
in_s = StringIO(out_s.getvalue())

# 读取数据
while True:
   try:
      o = pickle.load(in_s)
   except EOFError:
      break
   else:
      print 'READ: %s (%s)' % (o.name, o.name_backwards)
```

这个例子使用StringIO缓冲区来模拟流，因此我们在建立可读流时得玩点小花样。一个简单数据库格式也可以使用pickle来存储对象，虽然使用shelve模块可能会更简单。

```
$ python pickle_stream.py
WRITING: pickle (elkcip)
WRITING: cPickle (elkciPc)
```

```
WRITING: last (tsal)
READ: pickle (elkcip)
READ: cPickle (elkciPc)
READ: last (tsal)
```

除了用于存储数据，pickle在用于内部进程通信时是非常灵活的。比如，使用os.fork()和os.pipe()，可以建立一些工作进程，它们从一个管道中读取任务说明并把结果输出到另一个管道。操作这些工作池、发送任务和接受反应的核心代码可以重复利用，因为任务和反应对象不是一个特殊的类。如果你使用管道或者sockets，就不要忘记在dump每个对象后刷新它们并通过其间的连接将数据推送到另外一个进程。

在处理自定义类时，你应该保证这些被pickled的类会在进程名空间出现 。只有数据实例才能被pickle，而不能是定义的类。在unpickle时，类的名字被用于寻找构造器以便创建新对象。接下来这个例子，是将一个类实例写入到文件中：

```
try:
   import cPickle as pickle
except:
   import pickle
   import sys

class SimpleObject(object):

   def __init__(self, name):
      self.name = name
      l = list(name)
      l.reverse()
      self.name_backwards = ''.join(l)
      return

if __name__ == '__main__':
   data = []
   data.append(SimpleObject('pickle'))
   data.append(SimpleObject('cPickle'))
   data.append(SimpleObject('last'))
   try:
      filename = sys.argv[1]
      except IndexError:
      raise RuntimeError('Please specify a filename as an argument to %s' % sys.argv[0])
   out_s = open(filename, 'wb')
   try:
      # 写入到流中
      for o in data:
         print 'WRITING: %s (%s)' % (o.name, o.name_backwards)
         pickle.dump(o, out_s)
   finally:
      out_s.close()
```

当我运行这个脚本时，它会创建名为我在命令行中输入的参数的文件：

```
$ python pickle_dump_to_file_1.py test.dat
WRITING: pickle (elkcip)
WRITING: cPickle (elkciPc)
WRITING: last (tsal)
```

一个简单的尝试将刚才的pickle结果对象装载进来可以是如下的样子：

```
try:
   import cPickle as pickle
except:
   import pickle
import pprint
from StringIO import StringIO
import sys

try:
   filename = sys.argv[1]
except IndexError:
   raise RuntimeError('Please specify a filename as an argument to %s' % sys.argv[0])

in_s = open(filename, 'rb')
try:
   # 读取数据
   while True:
      try:
         o = pickle.load(in_s)
      except EOFError:
         break
      else:
         print 'READ: %s (%s)' % (o.name, o.name_backwards)
finally:
   in_s.close()
```

这个版本失败了，因为这里没有可用的SimpleObject类

```
$ python pickle_load_from_file_1.py test.dat
Traceback (most recent call last):
File "pickle_load_from_file_1.py", line 52, in
   o = pickle.load(in_s)
AttributeError: 'module' object has no attribute 'SimpleObject'
```

一个正确版本，它从pickle_dump_to_file_1导入了SimpleObject类，可以成功运行。 增加：

```
from pickle_dump_to_file_1 import SimpleObject
```

到导入列表的最后，然后运行这个脚本：

```
$ python pickle_load_from_file_2.py test.dat
READ: pickle (elkcip)
READ: cPickle (elkciPc)
READ: last (tsal)
```

在pickle那些不能被pickle的数据(如sockets、文件句柄、数据库连接等等)时，需要考虑一些特殊之处。那些不能被pickle的类可以定义 __getstate__() 和 __setstate__() 来返回实例在被pickle时的状态。新风格的类也可以定义__getnewargs__()，它返回传递给类内存分配者 (C.__new__())的参数。关于这些特征的更详细的使用描述可以在标准库文档中找到。

参考

- Pickle: An interesting stack language by Alexandre Vassalotti

0.2.11 PyMOTW: Queue

- 模块：Queue
- 目的：提供一个线程安全的FIFO功能。
- python版本：1.4+

描述

Queue提供了FIFO功能，一般常用于多线程编程，它可以在生产者和消费者线程中安全的传递消息或者其他数据。调用者会自动创建锁，当使用Queue对象，你可以根据需求创建多个线程。一个Queue的大小(元素的个数)受可用内存的限制。

本文假设你已经了解基本的Queue特点，如果你还不清楚，可以阅读参考后继续后面内容:

- Queue data structures
- FIFO

示例

举例说明如何在多线程中使用Queue对象，我们创建一个简单的 podcasting 客户端，这个客户端读取一个或者多个RSS feeds，依次将需下载的内容置于队列中，然后采用多线程模式同时处理多个下载。这比较简单，也许没有多大实用价值，但这个框架代码很好的说明了如何来利用Queue模块。

开始，我们加载一些有用的模块:

```
from Queue import Queue

from threading import Thread
import time

import feedparser
```

首先，需要创建一些运行参数，通常这些来自用户输入(可以任何东西，比如参数，数据库)，在我们的例子中，我们硬编码几个值。

```
# 设置两个全局变量
num_fetch_threads = 2
enclosure_queue = Queue()

# 实际情况中，一般不使用硬编码数据
feed_urls = [ 'http://www.castsampler.com/cast/feed/rss/guest',]
```

接下来，我们需要在工作线程中定义相应函数来处理下载。再次，这里为了便于说明模拟下载，实际下载可以参考 urllib 模块(这再以后会介绍)。在这个示例中，我们只根据线程id，使其sleep一段时间。

```
def downloadEnclosures(i, q):
  """这是一个工作线程函数.它按队列中的先后次序来处理各个项. 这个守护进程将进入一个无限循环, 当主线程退出时它才退出
  """
  while True:
    print '%s: Looking for the next enclosure' % i
    url = q.get()
    print '%s: Downloading:' % i, url
```

```
    time.sleep(i + 2) # 这里只是模拟，不是实际下载

    q.task_done()
```

一旦定义好目标函数，我们就可以启动工作线程。注意，函数downloadEnclosures()在“url = q.get()”会阻塞，直到队列有东西返回，因此，当队列中有东西时，启动线程总是安全的。

```
# 建立多个线程去抓取内容
for i in range(num_fetch_threads):
  worker = Thread(target=downloadEnclosures, args=(i, enclosure_queue,))
  worker.setDaemon(True)
  worker.start()
```

现在，我们开始检索feed的内容（使用Mark Pilgrim的 feedparser 模块）和一个url集合。当第一个url添加到队列后，一个工作线程即可选择它并启动下载。循环将继续运行并添加相应的feed，直到全部加完，工作线程将轮流取出url去下载它们。

```
# 下载feed，并且把url添加到队列中
for url in feed_urls:
  response = feedparser.parse(url, agent='fetch_podcasts.py')
  for entry in response['entries']:
    for enclosure in entry.get('enclosures', []):
      print 'Queuing:', enclosure['url']
      enclosure_queue.put(enclosure['url'])
```

剩下就可以等待队列为空。

```
# 等待队列为空，表明我们已经处理完所有下载。
print '*** Main thread waiting'
enclosure_queue.join()
print '*** Done'
```

下载如下 示例代码 ，运行即可看到如下输出：

```
0: Looking for the next enclosure
1: Looking for the next enclosure
Queuing: http://http.earthcache.net/htc-01.media.globix.net/COMP009996MOD1/Danny_Meyer.mp3
Queuing: http://feeds.feedburner.com/~r/drmoldawer/~5/104445110/moldawerinthemorning_show34_032607.mp
Queuing: http://www.podtrac.com/pts/redirect.mp3/twit.cachefly.net/MBW-036.mp3
Queuing: http://media1.podtech.net/media/2007/04/PID_010848/Podtech_calacaniscast22_ipod.mp4
Queuing: http://media1.podtech.net/media/2007/03/PID_010592/Podtech_SXSW_KentBrewster_ipod.mp4
Queuing: http://media1.podtech.net/media/2007/02/PID_010171/Podtech_IDM_ChrisOBrien2.mp3
Queuing: http://feeds.feedburner.com/~r/drmoldawer/~5/96188661/moldawerinthemorning_show30_022607.mp3
*** Main thread waiting
0: Downloading: http://http.earthcache.net/htc-01.media.globix.net/COMP009996MOD1/Danny_Meyer.mp3
1: Downloading: http://feeds.feedburner.com/~r/drmoldawer/~5/104445110/moldawerinthemorning_show34_03
0: Looking for the next enclosure
0: Downloading: http://www.podtrac.com/pts/redirect.mp3/twit.cachefly.net/MBW-036.mp3
1: Looking for the next enclosure
1: Downloading: http://media1.podtech.net/media/2007/04/PID_010848/Podtech_calacaniscast22_ipod.mp4
0: Looking for the next enclosure
0: Downloading: http://media1.podtech.net/media/2007/03/PID_010592/Podtech_SXSW_KentBrewster_ipod.mp4
0: Looking for the next enclosure
0: Downloading: http://media1.podtech.net/media/2007/02/PID_010171/Podtech_IDM_ChrisOBrien2.mp3
1: Downloading: http://feeds.feedburner.com/~r/drmoldawer/~5/96188661/moldawerinthemorning_show30_022
```

```
1: Looking for the next enclosure
*** Done
```

0.2.12 PyMOTW: shutil

- 模块：shutil
- 目的: 高层次的文件操作.
- python版本：1.4+

shutil模块提供了一些高层次的文件操作,比如复制,设置权限等等.

描述:

shutil模块提供了一些用于复制和删除整个文件的函数.

复制文件:

copyfile() 将源文件内容完全复制给目标文件.如果没有写入目标文件的权限,会引起IOError.由于该函数是为了读取文件内容而打开此输入文件,而不管它的类型是什么,特殊类型的文件使用copyfile()是不能拷贝的，比如管道文件。

```
import os
from shutil import *

print 'BEFORE:', os.listdir(os.getcwd())
copyfile('shutil_copyfile.py', 'shutil_copyfile.py.copy')
print 'AFTER:', os.listdir(os.getcwd())
```

```
$ python shutil_copyfile.py
BEFORE: ['__init__.py', 'shutil_copyfile.py']
AFTER: ['__init__.py', 'shutil_copyfile.py', 'shutil_copyfile.py.copy']
```

copyfile()底层调用了copyfileobj()函数.文件名参数传递给copyfile()后,进而将此文件句柄传递给copyfileobj().第三个可选参数是一个缓冲区长度,以块读入(默认情况下,一次性读取整个文件).

```
import os
from StringIO import StringIO
import sys
from shutil import *

class VerboseStringIO(StringIO):
    def read(self, n=-1):
        next = StringIO.read(self, n)
        print 'read(%d) =>' % n, next
        return next

lorem_ipsum = '''Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
              Vestibulum aliquam mollis dolor. Donec vulputate nunc ut diam.
              Ut rutrum mi vel sem. Vestibulum ante ipsum.'''
```

```
print 'Default:'
input = VerboseStringIO(lorem_ipsum)
output = StringIO()
copyfileobj(input, output)

print

print 'All at once:'
input = VerboseStringIO(lorem_ipsum)
output = StringIO()
copyfileobj(input, output, -1)

print

print 'Blocks of 20:'
input = VerboseStringIO(lorem_ipsum)
output = StringIO()
copyfileobj(input, output, 20)
```

默认的行为是以大块读取:

```
$ python shutil_copyfileobj.py
Default:
read(16384) => Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Vestibulum aliquam mollis dolor. Donec vulputate nunc ut diam.
Ut rutrum mi vel sem. Vestibulum ante ipsum.
read(16384) =>
```

使用-1表示一次性读取所有输入:

```
All at once:
read(-1) => Lorem ipsum dolor sit amet, consectetuer adipiscing elit.
Vestibulum aliquam mollis dolor. Donec vulputate nunc ut diam.
Ut rutrum mi vel sem. Vestibulum ante ipsum.
read(-1) =>
```

使用一个正整数设置块大小:

```
Blocks of 20:
read(20) => Lorem ipsum dolor si
read(20) => t amet, consectetuer
read(20) =>  adipiscing elit.
V
read(20) => estibulum aliquam mo
read(20) => llis dolor. Donec vu
read(20) => lputate nunc ut diam
read(20) => .
Ut rutrum mi vel
read(20) => sem. Vestibulum ante
read(20) =>  ipsum.
read(20) =>
```

copy()函数类似于Unix命令cp.如果目标参数是一个目录而不是一个文件,那么在这个目录中复制一个源文件副本(它与源文件同名).文件的权限也随之复制.

```
import os
from shutil import *

os.mkdir('example')
print 'BEFORE:', os.listdir('example')
copy('shutil_copy.py', 'example')
print 'AFTER:', os.listdir('example')


$ python shutil_copy.py
BEFORE: []
AFTER: ['shutil_copy.py']
```

copy2()函数类似于copy(),但是它将一些元信息,如文件最后一次被读取时间和修改时间等,也复制至新文件中.

```
import os
from shutil import *

def show_file_info(filename):
    stat_info = os.stat(filename)
    print '\tMode    :', stat_info.st_mode
    print '\tCreated :', time.ctime(stat_info.st_ctime)
    print '\tAccessed:', time.ctime(stat_info.st_atime)
    print '\tModified:', time.ctime(stat_info.st_mtime)

os.mkdir('example')
print 'SOURCE:'
show_file_info('shutil_copy2.py')
copy2('shutil_copy2.py', 'example')
print 'DEST:'
show_file_info('example/shutil_copy2.py')


$ python shutil_copy2.py

SOURCE:
        Mode    : 33188
        Created : Sun Oct 21 15:16:07 2007
        Accessed: Sun Oct 21 15:16:11 2007
        Modified: Sun Oct 21 15:16:07 2007
        DEST:
        Mode    : 33188
        Created : Sun Oct 21 15:16:11 2007
        Accessed: Sun Oct 21 15:16:11 2007
        Modified: Sun Oct 21 15:16:07 2007
```

复制文件元信息:

默认情况下,在Unix下,一个新创建的文件的权限会根据当前用户的umask值来设置.把一个文件的权限复制给另一个文件，可以使用copymode()函数.

```
from commands import *
from shutil import *

print 'BEFORE:', getstatus('file_to_change.txt')
```

```
copymode('shutil_copymode.py', 'file_to_change.txt')
print 'AFTER :', getstatus('file_to_change.txt')
```

首先,需要创建一个文件.然后对权限做些修改:

```
$ touch file_to_change.txt
$ chmod ugo+w file_to_change.txt
```

然后,运行刚才的示例脚本会改变之前的权限:

```
$ python shutil_copymode.py
BEFORE: -rw-rw-rw-   1 dhellman  dhellman  0 Oct 21 14:43 file_to_change.txt
AFTER : -rw-r--r--   1 dhellman  dhellman  0 Oct 21 14:43 file_to_change.txt
```

复制文件的其他元信息(权限,最后读取时间,最后修改时间)可以使用copystat().

```
import os
from shutil import *
import time

def show_file_info(filename):
    stat_info = os.stat(filename)
    print '\tMode    :', stat_info.st_mode
    print '\tCreated :', time.ctime(stat_info.st_ctime)
    print '\tAccessed:', time.ctime(stat_info.st_atime)
    print '\tModified:', time.ctime(stat_info.st_mtime)

print 'BEFORE:'
show_file_info('file_to_change.txt')
copystat('shutil_copystat.py', 'file_to_change.txt')
print 'AFTER :'
show_file_info('file_to_change.txt')
```

```
$ python shutil_copystat.py
BEFORE:
        Mode    : 33206
        Created : Sun Oct 21 15:01:23 2007
        Accessed: Sun Oct 21 14:43:26 2007
        Modified: Sun Oct 21 14:43:26 2007
        AFTER :
        Mode    : 33188
        Created : Sun Oct 21 15:01:44 2007
        Accessed: Sun Oct 21 15:01:43 2007
        Modified: Sun Oct 21 15:01:39 2007
```

目录树:

shutil模块包含3个操作目录树的函数.使用copytree()来复制目录,它会递归复制整个目录结构.目标目录必须不存在.其中, symlinks参数控制符号链接是否作为链接或文件被复制,默认是将其内容复制成一个新文件.如果此选项为true,新的链接会在目标目录树中创建.

Note: 注意:copytree()文档中说,它一般作为一个样本实现,而不是一个工具.你可以修改其源码,让它变得更稳定,或者增加一些功能,比如说进度条.

```
from commands import *
from shutil import *

print 'BEFORE:'
print getoutput('ls -rlast /tmp/example')
copytree('example', '/tmp/example')
print 'AFTER:'
print getoutput('ls -rlast /tmp/example')
```

```
$ python shutil_copytree.py
BEFORE:
ls: /tmp/example: No such file or directory
AFTER:
total 8
8 -rw-r--r--    1 dhellman  wheel  1627 Oct 21 15:16 shutil_copy2.py
0 drwxr-xr-x    3 dhellman  wheel   102 Oct 21 15:16 .
0 drwxrwxrwt   18 root      wheel   612 Oct 21 15:26 ..
```

使用rmtree()可以删除整个目录树.里面若产生错误会作为异常抛出.但是如果它的第二个参数是目录树,那么错误会被忽略,第三个参数可以指定为一个特殊出错处理函数句柄.

```
from commands import *
from shutil import *

print 'BEFORE:'
print getoutput('ls -rlast /tmp/example')
rmtree('example', '/tmp/example')
print 'AFTER:'
print getoutput('ls -rlast /tmp/example')
```

```
$ python shutil_rmtree.py

BEFORE:
total 8
8 -rw-r--r--    1 dhellman  wheel  1627 Oct 21 15:16 shutil_copy2.py
0 drwxr-xr-x    3 dhellman  wheel   102 Oct 21 15:16 .
0 drwxrwxrwt   18 root      wheel   612 Oct 21 15:26 ..
AFTER:
ls: /tmp/example: No such file or directory
```

移动文件或目录可以使用move(),这很类似于Unix命令mv.如果源文件或目录和目标文件或目录在同一个文件系统下,那么源文件或目录会直接重命名.否则源文件或目录会复制到目标文件或目录,接着删除源文件或目录.

```
import os
from shutil import *

print 'BEFORE: example : ', os.listdir('example')
move('example', 'example2')
print 'AFTER : example2: ', os.listdir('example2')
```

```
$ python shutil_move.py
BEFORE: example :  ['shutil_copy.py']
AFTER : example2:  ['shutil_copy.py']
```

参考

- PyMOTW

0.2.13 PyMOTW: smtplib

- 模块：smtplib
- 目的: 与smtp服务器交互，提供邮件发送
- python版本：1.5.2+

smtplib包含了SMTP类，用于与邮件服务器进行邮件通信。

Note: 在以下示例中，邮件地址、主机名称、ip地址都是虚假的，但是举例说明的命令副本和响应的信息都是存在的。

发送一封邮件

SMTP最常用的方法是连接服务器并发送一封邮件，在构造器中指定邮件服务器名和端口名，或者你可以使用connect()方法来指定。一旦建立连接，就可以调用sendmail()函数，并附带信封体参数和消息内容，消息文本应该与RFC2822兼容。smtplib不会自动修改消息内容和头信息，这就意味着你需要自己添加From和To等头信息。

```
import smtplib
import email.utils
from email.mime.text import MIMEText

# Create the message创建消息
msg = MIMEText('This is the body of the message.')
msg['To'] = email.utils.formataddr(('Recipient', 'recipient@example.com'))
msg['From'] = email.utils.formataddr(('Author', 'author@example.com'))
msg['Subject'] = 'Simple test message'

server = smtplib.SMTP('mail')
server.set_debuglevel(True) # show communication with the server显示与服务器的通信情况
try:
    server.sendmail('author@example.com', ['recipient@example.com'], msg.as_string())
finally:
    server.quit()
```

在这个示例中，调试开关被打开了，这样可以显示客户端和服务端之间的通讯信息，否则，示例不会显示任何信息。

```
$ python smtplib_sendmail.py
send: 'ehlo localhost.local\r\n'
reply: '250-mail.example.com Hello [192.168.1.17], pleased to meet you\r\n'
reply: '250-ENHANCEDSTATUSCODES\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-8BITMIME\r\n'
reply: '250-SIZE\r\n'
reply: '250-DSN\r\n'
reply: '250-ETRN\r\n'
reply: '250-AUTH GSSAPI DIGEST-MD5 CRAM-MD5\r\n'
reply: '250-DELIVERBY\r\n'
```

```
reply: '250 HELP\r\n'
reply: retcode (250); Msg: mail.example.com Hello [192.168.1.17], pleased to meet you
ENHANCEDSTATUSCODES
PIPELINING
8BITMIME
SIZE
DSN
ETRN
AUTH GSSAPI DIGEST-MD5 CRAM-MD5
DELIVERBY
HELP
send: 'mail FROM:<author@example.com> size=266\r\n'
reply: '250 2.1.0 <author@example.com>... Sender ok\r\n'
reply: retcode (250); Msg: 2.1.0 <author@example.com>... Sender ok
send: 'rcpt TO:<recipient@example.com>\r\n'
reply: '250 2.1.5 <recipient@example.com>... Recipient ok\r\n'
reply: retcode (250); Msg: 2.1.5 <recipient@example.com>... Recipient ok
send: 'data\r\n'
reply: '354 Enter mail, end with "." on a line by itself\r\n'
reply: retcode (354); Msg: Enter mail, end with "." on a line by itself
data: (354, 'Enter mail, end with "." on a line by itself')
send: 'From nobody Sun Sep 28 10:02:48 2008\r\nContent-Type: text/plain; charset="us-ascii"\r\nMIME-V
reply: '250 2.0.0 m8SE2mpc015614 Message accepted for delivery\r\n'
reply: retcode (250); Msg: 2.0.0 m8SE2mpc015614 Message accepted for delivery
data: (250, '2.0.0 m8SE2mpc015614 Message accepted for delivery')
send: 'quit\r\n'
reply: '221 2.0.0 mail.example.com closing connection\r\n'
reply: retcode (221); Msg: 2.0.0 mail.example.com closing connection
```

注意sendmail的第二个参数，收件人信息需要是一个list结构，你可以在list写上很多的邮件地址，message会依次把消息发送给他们. 由于信封信息和和邮件头是分开的，所以你可以通过一些方法参数来指定密送一些人，但不可以在邮件头中设置。

认证和加密

SMTP同样可以处理认证和TLS(一种底层通讯的安全协议)加密。如果服务器支持它们，你可以自己来检测服务器是否支持TLS，可以直接调用ehlo()来鉴定并询问服务器支持何种类型扩展 。然后通过调用has_extn()来检查结果。一旦启动TLS，你可以在认证之前再次调用ehlo()。

```
import smtplib
import email.utils
from email.mime.text import MIMEText
import getpass

# Prompt the user for connection info提示用户输入连接信息
to_email = raw_input('Recipient: ')
servername = raw_input('Mail server name: ')
username = raw_input('Mail user name: ')
password = getpass.getpass("%s's password: " % username)

# Create the message创建信息
msg = MIMEText('Test message from PyMOTW.')
msg.set_unixfrom('author')
msg['To'] = email.utils.formataddr(('Recipient', to_email))
msg['From'] = email.utils.formataddr(('Author', 'author@example.com'))
msg['Subject'] = 'Test from PyMOTW'
```

```
server = smtplib.SMTP(servername)
try:
    server.set_debuglevel(True)

    # identify ourselves, prompting server for supported features标识自己身份,确定服务器支持的特征
    server.ehlo()

    # If we can encrypt this session, do it如果我们可以加密这个会话,就进行
    if server.has_extn('STARTTLS'):
         server.starttls()
         server.ehlo() # re-identify ourselves over TLS connection再次标识身份通过TLS连接

    server.login(username, password)
    server.sendmail('author@example.com', [to_email], msg.as_string())
finally:
    server.quit()
```

注意STARTTLS不会出现在扩展列表中，因为启动了TLS。

```
$ python smtplib_authenticated.py
Recipient: recipient@example.com
Mail server name: smtpauth.isp.net
Mail user name: user@isp.net
user@isp.net's password:
send: 'ehlo localhost.local\r\n'
reply: '250-elasmtp-isp.net Hello localhost.local [<your IP here>]\r\n'
reply: '250-SIZE 14680064\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-AUTH PLAIN LOGIN CRAM-MD5\r\n'
reply: '250-STARTTLS\r\n'
reply: '250 HELP\r\n'
reply: retcode (250); Msg: elasmtp-isp.net Hello localhost.local [<your IP here>]
SIZE 14680064
PIPELINING
AUTH PLAIN LOGIN CRAM-MD5
STARTTLS
HELP
send: 'STARTTLS\r\n'
reply: '220 TLS go ahead\r\n'
reply: retcode (220); Msg: TLS go ahead
send: 'ehlo localhost.local\r\n'
reply: '250-elasmtp-isp.net Hello localhost.local [<your IP here>]\r\n'
reply: '250-SIZE 14680064\r\n'
reply: '250-PIPELINING\r\n'
reply: '250-AUTH PLAIN LOGIN CRAM-MD5\r\n'
reply: '250 HELP\r\n'
reply: retcode (250); Msg: elasmtp-isp.net Hello farnsworth.local [<your IP here>]
SIZE 14680064
PIPELINING
AUTH PLAIN LOGIN CRAM-MD5
HELP
send: 'AUTH CRAM-MD5\r\n'
reply: '334 PDExNjkyLjEyMjI2MTI1NzlAZWxhc210cC1tZWFseS5hdGwuc2EuZWFydGhsaW5rLm5ldD4=\r\n'
reply: retcode (334); Msg: PDExNjkyLjEyMjI2MTI1NzlAZWxhc210cC1tZWFseS5hdGwuc2EuZWFydGhsaW5rLm5ldD4=
send: 'ZGhlbGxtYW5uQGVhcnRobGluay5uZXQgN2Q1YjAyYTRmMGQ1YzZjM2NjOTNjZDc1MDQxN2ViYjg=\r\n'
reply: '235 Authentication succeeded\r\n'
reply: retcode (235); Msg: Authentication succeeded
```

```
send: 'mail FROM:<author@example.com> size=221\r\n'
reply: '250 OK\r\n'
reply: retcode (250); Msg: OK
send: 'rcpt TO:<recipient@example.com>\r\n'
reply: '250 Accepted\r\n'
reply: retcode (250); Msg: Accepted
send: 'data\r\n'
reply: '354 Enter message, ending with "." on a line by itself\r\n'
reply: retcode (354); Msg: Enter message, ending with "." on a line by itself
data: (354, 'Enter message, ending with "." on a line by itself')
send: 'Content-Type: text/plain; charset="us-ascii"\r\nMIME-Version: 1.0\r\nContent-Transfer-Encoding
reply: '250 OK id=1KjxNj-00032a-Ux\r\n'
reply: retcode (250); Msg: OK id=1KjxNj-00032a-Ux
data: (250, 'OK id=1KjxNj-00032a-Ux')
send: 'quit\r\n'
reply: '221 elasmtp-isp.net closing connection\r\n'
reply: retcode (221); Msg: elasmtp-isp.net closing connection
```

验证一个邮件地址

SMTP协议包含一个命令可以询问服务器一个邮件地址是否合法，通常VRFY是关闭的，以防止垃圾邮件发送者找到合法的邮件地址，但是，如果它打开，你可以向服务器询问这个邮件地址并接受一个状态码，如果是可用的，那么会返回一个可用的完整用户名。

```
import smtplib

server = smtplib.SMTP('mail')
server.set_debuglevel(True) # show communication with the server显示与服务器的通信情况
try:
    dhellmann_result = server.verify('dhellmann')
    notthere_result = server.verify('notthere')
finally:
    server.quit()

print 'dhellmann:', dhellmann_result
print 'notthere :', notthere_result
```

最后二行输出中表示，地址 dhellmann是合法的，notthere是非法的。

```
$ python smtplib_verify.py
send: 'vrfy <dhellmann>\r\n'
reply: '250 2.1.5 Doug Hellmann <dhellmann@mail.example.com>\r\n'
reply: retcode (250); Msg: 2.1.5 Doug Hellmann <dhellmann@mail.example.com>
send: 'vrfy <notthere>\r\n'
reply: '550 5.1.1 <notthere>... User unknown\r\n'
reply: retcode (550); Msg: 5.1.1 <notthere>... User unknown
send: 'quit\r\n'
reply: '221 2.0.0 mail.example.com closing connection\r\n'
reply: retcode (221); Msg: 2.0.0 mail.example.com closing connection
dhellmann: (250, '2.1.5 Doug Hellmann <dhellmann@mail.example.com>')
notthere : (550, '5.1.1 <notthere>... User unknown')
```

参考

- RFC 821: Simple Mail Transfer Protocol
- RFC 822: Standard for the Format of ARPA Internet Text Messages
- RFC 1869: SMTP Service Extensions
- RFC 2822: Internet Message Format
- 标准库文档: smtplib

0.2.14 PyMOTW: Struct

- 模块：Struct
- 目的: 实现字符串和二进制数据之间的相互转换
- python版本：1.4 +

struct模块包含了实现字符串字节和Python本地数据类型(如数字和字符串)间的相互转换的函数.

函数 vs Struct类

这里有一系列用来处理结构化数值的模块级函数, 同样存在Struct类(Python 2.5中新加入的). 格式化描述即将字符串格式转化为可编译形式,类似于正则式的方式. 这种转换需要消耗一些资源,所以一旦创建Struct实例后并调用Struct实例的方法而不使用模块级的方法,这样是更有效的.下面举些使用Struct类的例子.

封装和解封

Structs支持将数据封装成字符串,也能够通过格式化描述(它是由一些代表特定数据类型的字符,可选的个数和字节序指示符组成的)从字符串中解封数据. 进一步的细节,可以参考 标准库文档

在下面的这个例子中, 格式化描述调用了一个整型或者长整型数, 一个2个字符组成的字符串和一个浮点数. 为了清晰描述，在格式化描述中包含了空格，但格式被编译后将忽略这个空格.

```
import struct
import binascii

values = (1, 'ab', 2.7)
s = struct.Struct('I 2s f')
packed_data = s.pack(*values)

print 'Original values:', values
print 'Format string  :', s.format
print 'Uses           :', s.size, 'bytes'
print 'Packed Value   :', binascii.hexlify(packed_data)
```

封装后的值被转换成16进制字节流输出,所以有些字符显示为空

```
$ python struct_pack.py
Original values: (1, 'ab', 2.7000000000000002)
Format string  : I 2s f
Uses           : 12 bytes
Packed Value   : 0100000061620000cdcc2c40
```

如果我们将封装后的值传递给unpack(), 可以基本上得到原来的数值(注意其中浮点数的差异).

```
import struct
import binascii

packed_data = binascii.unhexlify('0100000061620000cdcc2c40')

s = struct.Struct('I 2s f')
unpacked_data = s.unpack(packed_data)
print 'Unpacked Values:', unpacked_data
```

```
$ python struct_unpack.py
Unpacked Values: (1, 'ab', 2.7000000476837158)
```

字节序

默认情况下,使用标准C库中"字节序"的概念将数值编码.通过在字符串格式中直接指定一个明确的字节序可以简单的覆盖这个选项.

```
import struct
import binascii

values = (1, 'ab', 2.7)
print 'Original values:', values

endianness = [
    ('@', 'native, native'),
    ('=', 'native, standard'),
    ('<', 'little-endian'),
    ('>', 'big-endian'),
    ('!', 'network'),
]


for code, name in endianness:
    s = struct.Struct(code + ' I 2s f')
    packed_data = s.pack(*values)
    print
    print 'Format string  :', s.format, 'for', name
    print 'Uses           :', s.size, 'bytes'
    print 'Packed Value   :', binascii.hexlify(packed_data)
    print 'Unpacked Value :', s.unpack(packed_data)
```

```
$ python struct_endianness.py
Original values: (1, 'ab', 2.7000000000000002)

Format string  : @ I 2s f for native, native
Uses           : 12 bytes
Packed Value   : 0100000061620000cdcc2c40
Unpacked Value : (1, 'ab', 2.7000000476837158)

Format string  : = I 2s f for native, standard
Uses           : 10 bytes
Packed Value   : 010000006162cdcc2c40
Unpacked Value : (1, 'ab', 2.7000000476837158)
```

```
Format string  : < I 2s f for little-endian
Uses           : 10 bytes
Packed Value   : 0100000006162cdcc2c40
Unpacked Value : (1, 'ab', 2.7000000476837158)

Format string  : > I 2s f for big-endian
Uses           : 10 bytes
Packed Value   : 000000016162402ccccd
Unpacked Value : (1, 'ab', 2.7000000476837158)

Format string  : ! I 2s f for network
Uses           : 10 bytes
Packed Value   : 000000016162402ccccd
Unpacked Value : (1, 'ab', 2.7000000476837158)
```

缓冲

在高性能的敏感情况或者通过通过第三方模块来传递数据经常会要求对二进制数据进行封装.一种优化的方式是避免为每一个封装结构分配新的缓冲区.函数pack_into()和unpack_from()支持直接写入到预分配的缓冲区中.

```
import struct
import binascii

s = struct.Struct('I 2s f')
values = (1, 'ab', 2.7)
print 'Original:', values

print
print 'ctypes string buffer'

import ctypes
b = ctypes.create_string_buffer(s.size)
print 'Before  :', binascii.hexlify(b.raw)
s.pack_into(b, 0, *values)
print 'After   :', binascii.hexlify(b.raw)
print 'Unpacked:', s.unpack_from(b, 0)

print
print 'array'

import array
a = array.array('c', '\0' *s.size)
print 'Before  :', binascii.hexlify(a)
s.pack_into(a, 0, *values)
print 'After   :', binascii.hexlify(a)
print 'Unpacked:', s.unpack_from(a, 0)
```

```
$ python struct_buffers.py

Original: (1, 'ab', 2.7000000000000002)

ctypes string buffer
Before  : 000000000000000000000000
```

```
After   : 0100000061620000cdcc2c40
Unpacked: (1, 'ab', 2.7000000476837158)

array
Before  : 000000000000000000000000
After   : 0100000061620000cdcc2c40
Unpacked: (1, 'ab', 2.7000000476837158)
```

参考

- struct
- array：用于处理固定类型的序列.
- binascii：用于产生二进制数据的ASCII表示.
- WikiPedia: Endianness

字节序,其实是数据的二进制形式的排列顺序,在内存存贮顺序,或者是传输时的顺序,或者还有其他特殊的规定.

0.2.15 PyMOTW: textwrap

- 模块：textwrap
- 目的: 通过调整段落中的换行符位置来格式化文本
- python版本：2.5

描述

textwrap模块可以用来格式化文本，使其在某些场合输出更美观。他提供了一些类似于在很多文本编辑器中都有的段落包装或填充特性的程序功能.

例子

```
import textwrap

# Provide some sample text 提供样本文本
sample_text = '''

 The textwrap module can be used to format text for output in situations
 where pretty-printing is desired.  It offers programmatic functionality similar
 to the paragraph wrapping or filling features found in many text editors.

'''
```

fill()将文本作为输入，格式化文本作为输出。让我们看下面是如何对样本文本进行格式化的

```
print 'No dedent:\n'
print textwrap.fill(sample_text)
```

结果比我们想要的结果要少：

```
No dedent:

        The textwrap module can be used to format text for output in
        situations         where pretty-printing is desired.  It offers
        programmatic functionality similar         to the paragraph wrapping
        or filling features found in many text editors.
```

Note: 注意嵌入的tab符号和多余的空格被混合在输出文本中。这个看起来是很粗糙的。当然，我们可以做的更好。我们想在样本文本中的每一行的开始处删掉所有普通空格前缀。这个允许我们在去除代码本身的格式化时直接从Python代码中使用文档字符串或者嵌入式多行字符串。下面的样本字符串引入了一个人工的缩进层次以便更好的说明这个特征。

删除样本行中的普通空格前缀

```
dedented_text = textwrap.dedent(sample_text).strip()
print 'Dedented:\n'
print dedented_text
```

结果看上去似乎好点：

```
Dedented:

The textwrap module can be used to format text for output in situations
where pretty-printing is desired.  It offers programmatic functionality similar
to the paragraph wrapping or filling features found in many text editors.
```

由于“dedent”是“indent”的相反，结果就是将每行开始的普通空白符删除了。如果某行已经比另一行多了个缩进层次，那么对应的空格不会被去掉。

```
>>> a="""
...     one tab
...             two tab
...     one tab
... """
>>> import textwrap
>>> dedented_text = textwrap.dedent(a).strip()
>>> print dedented_text
one tab
    two tab
one tab
>>> print a
    one tab
        two tab
    one tab
>>>
```

接下来，让我们看下如果我们传递非缩进格式的文本给fill()，并使用一些不同的宽度值，会发生什么。

使用不同行宽值进行格式化输出：

```
for width in [ 20, 60, 80 ]:
    print
    print '%d Columns:\n' % width
    print textwrap.fill(dedented_text, width=width)
```

在指定不同宽度时会有以下不同的输出结果：

20 Columns:

```
The textwrap module
can be used to
format text for
output in situations
where pretty-
printing is desired.
It offers
programmatic
functionality
similar to the
paragraph wrapping
or filling features
found in many text
editors.
```

60 Columns:

```
The textwrap module can be used to format text for output in
situations where pretty-printing is desired.  It offers
programmatic functionality similar to the paragraph wrapping
or filling features found in many text editors.
```

80 Columns:

```
The textwrap module can be used to format text for output in situations where
pretty-printing is desired.  It offers programmatic functionality similar to the
paragraph wrapping or filling features found in many text editors.
```

除了制定输出中的宽度，你可以控制首行缩进，他独立于接下来的行。

```
# 演示怎样去产生悬挂缩进
print '\nHanging indent:\n'
print textwrap.fill(dedented_text, initial_indent='', subsequent_indent='    ')
```

这个看起来很容易就能实现文本的悬挂缩进，也就是首行要比后继行有少的缩进。

```
Hanging indent:

The textwrap module can be used to format text for output in
   situations where pretty-printing is desired.  It offers
   programmatic functionality similar to the paragraph wrapping or
   filling features found in many text editors.
```

缩进值也可以是非空格字符，因此，可以用*作为前缀产生bullet点，等等。在我转换老的zwiki内容以便将其导入到trac中是很灵活的。我使用Zope中的StructuredText包来解析zwiki数据，然后创建一个格式化器产生一个trac的wiki标记作为输出。使用textwrap就可以格式化输出页，因此转换后就几乎不需要再进行手工操作整个转换过程几乎没有手工进行。

参考

- textwrap_example.py

0.2.16 PyMOTW: time

time模块提供了操作日期和时间的函数

- 模块：time
- 目的：操作times的函数
- python版本：1.4+

描述

time模块是利用了c函数来处理日期和时间，也就是说它绑定了c的实现，一些特定的细节（比如纪元的开始时间、日期的最大值）是和平台相关的，具体可以参考 这里 。

Wall Clock Time

time模块的核心函数之一就是time.time()函数，它返回一个自公元开始的总秒数（浮点型）。

本工具包含三个文件：

```
import time
print 'The time is:', time.time()
```

虽然返回的值是浮点型，但精度是依赖于不同的系统平台的。

```
$ python time_time.py
The time is: 1205079300.54
```

当存储和比较日期时，浮点型一般是很有用的，但这种方式不易阅读，为了更有用的记录和输出时间可以使用time.ctime()。

```
import time
print 'The time is      :', time.ctime()
later = time.time() + 15
print '15 secs from now :', time.ctime(later)
```

上面第二行示范了如何来利用ctime()函数对当前时间进行格式化。

```
$ python time_ctime.py
The time is      : Sun Mar  9 12:18:02 2008
15 secs from now : Sun Mar  9 12:18:17 2008
```

处理器时钟

time()函数返回的是现实世界的时间，而clock()函数返回的是cpu时钟。clock()函数返回值常用作性能测试，benchmarking等。它们常常反映了程序运行的真实时间，比time()函数返回的值要精确。

```
import hashlib
import time

# 用于计算md5校验和的数据
data = open(__file__, 'rt').read()
```

```
for i in range(5):
    h = hashlib.sha1()
    print time.ctime(), ': %0.3f %0.3f' % (time.time(), time.clock())
    for i in range(100000):
        h.update(data)
    cksum = h.digest()
```

在这个例子中，ctime()把time()函数返回的浮点型表示转化为标准时间，每个迭代循环使用了clock()。如果想在机器上测试这个例子，那么可以增加循环次数，或者处理大一点的数据，这样才能看到不同点。

```
$ python time_clock.py
Sun Mar  9 12:41:53 2008 : 1205080913.260 0.030
Sun Mar  9 12:41:53 2008 : 1205080913.682 0.440
Sun Mar  9 12:41:54 2008 : 1205080914.103 0.860
Sun Mar  9 12:41:54 2008 : 1205080914.518 1.270
Sun Mar  9 12:41:54 2008 : 1205080914.932 1.680
```

一般，如果程序没有做任何事情，处理器时钟是不会计时。

```
import time

for i in range(6, 1, -1):
    print '%s %0.2f %0.2f' % (time.ctime(), time.time(), time.clock())
    print 'Sleeping', i
    time.sleep(i)
```

在这个例子中，每次迭代，循环中处理了很少的任务就进入了sleep，当进程在睡眠中时，time.time()函数的返回值依然会增加。但是time.clock()是不会增加的。

```
$ python time_clock_sleep.py
Sun Mar  9 12:46:36 2008 1205081196.20 0.02
Sleeping 6
Sun Mar  9 12:46:42 2008 1205081202.20 0.02
Sleeping 5
Sun Mar  9 12:46:47 2008 1205081207.20 0.02
Sleeping 4
Sun Mar  9 12:46:51 2008 1205081211.20 0.02
Sleeping 3
Sun Mar  9 12:46:54 2008 1205081214.21 0.02
Sleeping 2
```

time.sleep函数控制当前的线程，让它等待直到系统重新唤醒它，如果应用中只有一个线程，那么它会阻塞当前进程，使其不做任何事情。

struct_time

某些时候，使用逝去的秒数来表示时间是很有用的。有时候你需要获取日期的单独部分（如年、月等等），time模块定义了struct_time来存储日期和时间值并作为其部分以便获取。提供了多种函数将struct_time转化为float。

```
import time

print 'gmtime   :', time.gmtime()
print 'localtime:', time.localtime()
```

```
print 'mktime   :', time.mktime(time.localtime())

print
t = time.localtime()
print 'Day of month:', t.tm_mday
print ' Day of week:', t.tm_wday
print ' Day of year:', t.tm_yday
```

gmtime()返回当前的UTC时间，localtime()返回当前时间域的当前时间，mktime()接收struct_time参数并将其转化为浮点型来表示。

```
$ python time_struct.py
gmtime   : (2008, 3, 9, 16, 58, 19, 6, 69, 0)
localtime: (2008, 3, 9, 12, 58, 19, 6, 69, 1)
mktime   : 1205081899.0

Day of month: 9
Day of week: 6
Day of year: 69
```

解析和格式化时间

函数strptime()和strftime()可以使struct_time和时间值字符串相互转化。有一个很长的格式化说明列表可以用来支持输入和输出不同的风格。完整的列表在time模块的的库文档中有介绍。

下面示例把当前时间（字符串)转化为struct_time实例，然后再转化为字符串。

```
import time

now = time.ctime()
print now
parsed = time.strptime(now)
print parsed
print time.strftime("%a %b %d %H:%M:%S %Y", parsed)
```

输出和输入字符串不是完全的一致，主要表现在月份前加了一个0前缀。

```
$ python time_strptime.py
Sun Mar  9 13:01:19 2008
(2008, 3, 9, 13, 1, 19, 6, 69, -1)
Sun Mar 09 13:01:19 2008
```

使用Time Zone(时区)

无论是你的程序，还是为系统使用一个默认的时区，检测当前时间的函数依赖于当前Time Zone（时间域)的设置。改变时区设置是不会改变实际时间，只会改变表示时间的方法。

通过设置环境变量TZ可以改变时区，然后调用tzset()。环境变量TZ可以对时区来详细的设置，比如白天保存时间的起始点。通常使用时区名称是比较简单的，如果需要了解更多信息可以参考库。

下面这个示例改变了time zone中的一些值，展示了这种改变如何来影响time模块中的其它设置。

```
import time
import os

def show_zone_info():
    print '\tTZ    :', os.environ.get('TZ', '(not set)')
    print '\ttzname:', time.tzname
    print '\tZone  : %d (%d)' % (time.timezone, (time.timezone / 3600))
    print '\tDST   :', time.daylight
    print '\tTime  :', time.ctime()
    print

print 'Default :'
show_zone_info()

for zone in [ 'US/Eastern', 'US/Pacific', 'GMT', 'Europe/Amsterdam' ]:
    os.environ['TZ'] = zone
    time.tzset()
    print zone, ':'
    show_zone_info()
```

我的时区是US/Eastern，所以设置TZ不会起作用。如果是其它时区，则会改变tzname、daylight flag以及timezone偏移值。

```
$ python time_timezone.py
Default :
  TZ    : (not set)
  tzname: ('EST', 'EDT')
  Zone  : 18000 (5)
  DST   : 1
  Time  : Sun Mar  9 13:06:53 2008

US/Eastern :
  TZ    : US/Eastern
  tzname: ('EST', 'EDT')
  Zone  : 18000 (5)
  DST   : 1
  Time  : Sun Mar  9 13:06:53 2008

US/Pacific :
  TZ    : US/Pacific
  tzname: ('PST', 'PDT')
  Zone  : 28800 (8)
  DST   : 1
  Time  : Sun Mar  9 10:06:53 2008

GMT :
  TZ    : GMT
  tzname: ('GMT', 'GMT')
  Zone  : 0 (0)
  DST   : 0
  Time  : Sun Mar  9 17:06:53 2008

Europe/Amsterdam :
  TZ    : Europe/Amsterdam
  tzname: ('CET', 'CEST')
  Zone  : -3600 (-1)
  DST   : 1
```

```
  Time   : Sun Mar  9 18:06:53 2008
```

参考

- datetime module
- locale module
- calendar module

0.2.17 PyMOTW: Trace

- 模块：Trace
- 目的: 监控程序语句和函数运行情况,并且产生报告信息.
- python版本：2.3+

trace - 跟踪正在执行的Python语句
trace模块帮助你明白程序的运行过程.你可以跟踪执行的语句,产生报表,也能获取函数间的调用关系.

命令行接口

可以很简单的直接从命令行使用trace.给定以下的Python脚本:

```
from recurse import recurse

def main():
     print 'This is the main program.'
     recurse(2)
     return

if __name__ == '__main__':
     main()

def recurse(level):
     print 'recurse(%s)' % level
     if level:
         recurse(level-1)
     return

def not_called():
     print 'This function is never called.'
```

跟踪时的异常

我们可以使用–trace选项来查看程序运行时哪条语句正在被执行.

```
$ python -m trace --trace trace_example/main.py
--- modulename: threading, funcname: settrace
threading.py(70): _trace_hook = func
```

```
--- modulename: trace, funcname: <module>
<string>(1): --- modulename: trace, funcname: <module>
main.py(7): """
main.py(12): from recurse import recurse
--- modulename: recurse, funcname: <module>
recurse.py(7): """
recurse.py(12): def recurse(level):
main.py(14): def main():
main.py(19): if __name__ == '__main__':
main.py(20): main()
--- modulename: trace, funcname: main
main.py(15): print 'This is the main program.'
This is the main program.
main.py(16): recurse(2)
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(2)
recurse.py(14): if level:
recurse.py(15): recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(1)
recurse.py(14): if level:
recurse.py(15): recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(0)
recurse.py(14): if level:
recurse.py(16): return
recurse.py(16): return
recurse.py(16): return
main.py(17): return
```

输出结构的第一部分表明了trace的一个安装操作.剩下来的输出显示了每个函数的入口信息,包括函数位于哪个模块,然后是原脚本文件中的语句行.你可以看到函数recurse()被进入了3次,正如你在main()中调用的那样.

代码报告

从命令行中运行trace并使用–count选项可以产生代码信息报告,因此可以看到哪些行是被执行的,哪些被跳过了.因为你的程序通常是多个文件组成,那就会为每个文件产生独立的报表.默认下,报表文件在和模块的同一目录下被创建,并以模块名命名,而且使用.cover后缀名替换.py.

```
$ python -m trace --count trace_example/main.py
This is the main program.
recurse(2)
recurse(1)
recurse(0)
```

两个输出文件, trace_example/main.cover:

```
1: from recurse import recurse

1: def main():
1:     print 'This is the main program.'
1:     recurse(2)
1:     return
```

```
1: if __name__ == '__main__':
1:     main()
```

trace_example/recurse.cover:

```
1: def recurse(level):
3:     print 'recurse(%s)' % level
3:     if level:
2:         recurse(level-1)
3:     return
```

Note: 虽然代码行def recurse(level):有一个1数值, 这不意味着这个函数仅运行一次,而是意味着这个函数definition仅被执行一次. 使用不同的选项来多次运行程序是有可能的,并且保存报告数据,产生一个联合报告.

```
$ python -m trace --coverdir coverdir1 --count --file coverdir1/coverage_report.dat trace_example/mai
This is the main program.
recurse(2)
recurse(1)
recurse(0)
Skipping counts file 'coverdir1/coverage_report.dat': [Errno 2] No such file or directory: 'coverdir1


$ python -m trace --coverdir coverdir1 --count --file coverdir1/coverage_report.dat trace_example/mai
This is the main program.
recurse(2)
recurse(1)
recurse(0)


$ python -m trace --coverdir coverdir1 --count --file coverdir1/coverage_report.dat trace_example/mai
This is the main program.
recurse(2)
recurse(1)
recurse(0)


$ find coverdir1
coverdir1
coverdir1/coverage_report.dat
```

一旦报告信息被记录到.cover文件中,你可以使用–report选项产生报告.

```
$ python -m trace --coverdir coverdir1 --report --summary --missing --file coverdir1/coverage_report.
lines cov% module (path)
533 0% threading (/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/threading.py)
8 100% trace_example.main (trace_example/main.py)
8 87% trace_example.recurse (trace_example/recurse.py)
$ find coverdir1
coverdir1
coverdir1/coverage_report.dat
coverdir1/threading.cover
coverdir1/trace_example.main.cover
coverdir1/trace_example.recurse.cover
```

程序一共运行了3次,因此在报告中显示的值要比第一份报告中的值高3倍.–summary选项在输出信息中增加了百分比信息.模块recurse只有 87%被报告.从这个报告中还可看到not_called()这个函数从未被运行,这个是由前缀>>>>>>表示.

```
3: def recurse(level):
9:     print 'recurse(%s)' % level
9:     if level:
6:         recurse(level-1)
9:     return

3: def not_called():
>>>>>> print 'This function is never called.'
```

调用关系

除了以上覆盖信息,trace还可以收集函数间调用关系.使用–listfuncs可以在结果中输出简单的函数调用关系:

```
$ python -m trace --listfuncs trace_example/main.py
This is the main program.
recurse(2)
recurse(1)
recurse(0)

functions called:
filename: /Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/threading.py, modulename: t
filename: <string>, modulename: <string>, funcname: <module>
filename: trace_example/main.py, modulename: main, funcname: <module>
filename: trace_example/main.py, modulename: main, funcname: main
filename: trace_example/recurse.py, modulename: recurse, funcname: <module>
filename: trace_example/recurse.py, modulename: recurse, funcname: recurse
```

可以使用–trackcalls获得更多信息,比如说谁调用了函数.

```
$ python -m trace --listfuncs --trackcalls trace_example/main.py
This is the main program.
recurse(2)
recurse(1)
recurse(0)

calling relationships:

*** /Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/trace.py ***
--> /Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/threading.py
trace.Trace.run -> threading.settrace
--> <string>
trace.Trace.run -> <string>.<module>

*** <string> ***
--> trace_example/main.py
<string>.<module> -> main.<module>

*** trace_example/main.py ***
main.<module> -> main.main
--> trace_example/recurse.py
main.<module> -> recurse.<module>
main.main -> recurse.recurse
```

```
*** trace_example/recurse.py ***
recurse.recurse -> recurse.recurse
```

编程接口

通过trace接口增加更多的控制,你可以在你的程序中使用Trace对象.Trace可以让你设置fixtures和其他依赖关系在运行单个函数前或执行一个用于跟踪的Python命令.

```
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=False, trace=True)
tracer.run('recurse(2)')
```

由于例子只跟踪到recurse()函数, 所以结果中没有把main.py的信息包含进来.

```
$ python trace_run.py
--- modulename: threading, funcname: settrace
threading.py(70): _trace_hook = func
--- modulename: trace_run, funcname: <module>
<string>(1): --- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(2)
recurse.py(14): if level:
recurse.py(15): recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(1)
recurse.py(14): if level:
recurse.py(15): recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(0)
recurse.py(14): if level:
recurse.py(16): return
recurse.py(16): return
recurse.py(16): return
```

使用runfunc()也可以得到上述同样的输出.runfunc()接收任意位置和关键字参数,他们在函数被tracer调用时都被传递给函数.

```
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=False, trace=True)
tracer.runfunc(recurse, 2)
```

```
$ python trace_runfunc.py
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(2)
recurse.py(14): if level:
recurse.py(15): recurse(level-1)
--- modulename: recurse, funcname: recurse
```

```
recurse.py(13): print 'recurse(%s)' % level
recurse(1)
recurse.py(14): if level:
recurse.py(15): recurse(level-1)
--- modulename: recurse, funcname: recurse
recurse.py(13): print 'recurse(%s)' % level
recurse(0)
recurse.py(14): if level:
recurse.py(16): return
recurse.py(16): return
recurse.py(16): return
```

保存结果数据

就像在命令行中使用一样, 计算和报告信息也可以被记录下来.使用Trace对象的CoverageResults可以将这些数据明确的保存下来.

```
import trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=True, trace=False)
tracer.runfunc(recurse, 2)

results = tracer.results()
results.write_results(coverdir='coverdir2')
```

```
$ python trace_CoverageResults.py
recurse(2)
recurse(1)
recurse(0)

$ find coverdir2
coverdir2/
coverdir2//trace_example.recurse.cover

$ cat coverdir2/trace_example.recurse.cover
#!/usr/bin/env python
# encoding: utf-8
#
# Copyright (c) 2008 Doug Hellmann All rights reserved.
#
"""
"""

#__version__ = "$Id: recurse.py 1732 2008-10-12 14:50:28Z dhellmann $"
#end_pymotw_header

>>>>>> def recurse(level):
3: print 'recurse(%s)' % level
3: if level:
2: recurse(level-1)
3: return

>>>>>> def not_called():
>>>>>> print 'This function is never called.'
```

为了在生成报告时也保存计算数据,可以使用参数infile和outfile.

```
mport trace
from trace_example.recurse import recurse

tracer = trace.Trace(count=True, trace=False, outfile='trace_report.dat')
tracer.runfunc(recurse, 2)

report_tracer = trace.Trace(count=False, trace=False, infile='trace_report.dat')
results = tracer.results()
results.write_results(summary=True, coverdir='/tmp')
```

传递给参数infile一个文件名来余弦读取存储的数据, 参数outfile指定在跟踪之后需要新建的一个结果文件名.如果infile和outfile是相同的,那么,就相当于在原有文件中增加新的数据.

```
$ python trace_report.py
recurse(2)
recurse(1)
recurse(0)
lines cov% module (path)
7 57% trace_example.recurse (trace_example/recurse.py)
```

Trace选项

Trace构造器可以带多个可选参数以便更好的控制运行行为.

- count: 布尔型.打开行号计数.默认是True.
- countfuncs: 布尔型.打开运行中函数调用列表.默认是False
- countcallers: 布尔型.打开跟踪时的调用者和被调用者信息.默认是False.
- ignoremods: 序列.在跟踪报告中需要忽略的模块或包列表.默认是一个空元祖.
- ignoredirs: 序列.在跟踪报告中需要忽略的目录(其中包含模块或包)列表.默认是一个空元祖.
- infile: 包含缓存信息的文件名,作为输入.默认是None.
- outfile: 用于存储缓存信息的文件名,作为输入.默认是None,也就是数据不被存储.

参考

- 标准库文档: trace

0.2.18 PyMOTW: webbrowser

利用webbrowser模块可以向用户显示web页面。

- 模块：webbrowser
- 目的：在浏览器中打开web页面
- python版本：python2.1.3+

描述

在一个交互式的浏览程序中，webbrowser模块提供了一些用于打开URL链接的函数。在系统中安装的浏览器，通过模块的许多选项可以来获取利用他们。也可通过环境变量BROWSER来控制。

简单示例

在浏览器中打开一个页面，可以使用open()函数。

```
import webbrowser

webbrowser.open('http://docs.python.org/lib/module-webbrowser.html')
```

这个url会在一个新窗口中打开，当然如果当前已经有一个浏览器窗口，那么会做为一个新标签打开。

窗口 Vs 标签

如果你只想在新窗口中打开页面，那么可以使用 `open_new()` 。

```
import webbrowser

webbrowser.open_new('http://docs.python.org/lib/module-webbrowser.html')
```

如果你想在新的标签中打开，那么可以使用 `open_new_tab()` 。

使用特定的浏览器

因为某些原因，你的应用程序可能需要使用特定的浏览器，可以利用get()函数来取得浏览器访问对象，该对象提供了open()、open_new()和open_new_tab()函数，下面示例演示了如何利用lynx浏览器。

```
import webbrowser

b = webbrowser.get('lynx')
b.open('http://docs.python.org/lib/module-webbrowser.html')
```

可以参考模块文档来了解浏览器类型列表。

BROWSER 变量

用户可以在你的应用程序之外为BROWSER变量设置浏览器名字或者命令来控制webbrowser模块，值可以由一系列的浏览器名字组成，中间由系统分割符os.pathsep来分割。如果名字中包含%s，那么将被解释成命令，并且在URL中%s将被直接替换执行。否则，值将被传递给get()来获取控制对象。

举个例子，如下示例是打开一个lynx浏览器，假设它是可以获取的，不管是否还存在其它的浏览器。

```
BROWSER=lynx python webbrowser_open.py
```

如果BROWSER中的名字中没有一个可以正确工作，那么webbrowse会返回执行它的默认行为。

命令行接口

webbrowser模块的所有特性可以通过命令行获取，类似运行一个python程序。

```
$ python -m webbrowser
Usage: /Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/webbrowser.py [-n | -t] url
   -n: open new window
   -t: open new tab
```

0.2.19 PyMOTW: locale

- 模块：locale
- 目的：POSIX标准的本地化API
- python版本：1.5,在2.5版本中有所扩展

描述

locale模块是Python国际化和本地化支持库的一部分. 他提供一种用于处理那些可能依赖于你用户语言或位置的操作的标准方式. 例如, 货币格式化, 比较字符串以便排序, 处理时间日期. 他没有包含翻译(可参见gettext模块)或Unicode编码.

由于可以在应用程序范围内改变本地化设置, 所以推荐用户避免在库中改变值而是让应用程序一次性设置. 在下面的例子中, 我会改变本地的一些时间以便说明目的.这更像是一旦你的应用程序启动就去设置本地化参数.

例子

让用户改变一个应用程序的本地设置的最一般的方式是通过一个环境变量（LC__ALL, LC__CTYPE, LANG, 或LANGUAGE, 这依赖于你的平台). 然后程序会调用locale.setlocale(), 没有使用硬编码值, 而是使用环境变量.

```
import locale
import os
import pprint

print 'Environment settings:'
for env_name in [ 'LC_ALL', 'LC_CTYPE', 'LANG', 'LANGUAGE' ]:
    print '\t%s = %s' % (env_name, os.environ.get(env_name, ''))

# What is the default locale?默认的本地设置是什么?
print
print 'Default locale:', locale.getdefaultlocale()

# Default settings based on the user's environment.根据用户的环境做些默认设置
locale.setlocale(locale.LC_ALL, '')

# If we do not have a locale, assume US English.如果没有本地设置, 假设为US English
print 'From environment:', locale.getlocale()

pprint.pprint(locale.localeconv())
```

在我的Mac上, 这个程序输出类似如下:

```
$ python locale_env_example.py
Environment settings:
 LC_ALL =
 LC_CTYPE =
 LANG =
 LANGUAGE =

Default locale: (None, 'mac-roman')
From environment: (None, None)
{'currency_symbol': '',
 'decimal_point': '.',
 'frac_digits': 127,
 'grouping': [127],
 'int_curr_symbol': '',
 'int_frac_digits': 127,
 'mon_decimal_point': '',
 'mon_grouping': [127],
 'mon_thousands_sep': '',
 'n_cs_precedes': 127,
 'n_sep_by_space': 127,
 'n_sign_posn': 127,
 'negative_sign': '',
 'p_cs_precedes': 127,
 'p_sep_by_space': 127,
 'p_sign_posn': 127,
 'positive_sign': '',
 'thousands_sep': ''}
```

现在如果我们设置好LANG值后再运行同样的脚本, 可以看到本地设置和默认编码因此改变:

法国:

```
$ LANG=fr_FR python locale_env_example.py
Environment settings:
 LC_ALL =
 LC_CTYPE =
 LANG = fr_FR
 LANGUAGE =

Default locale: (None, 'mac-roman')
From environment: ('fr_FR', 'ISO8859-1')
{'currency_symbol': 'Eu',
 'decimal_point': ',',
 'frac_digits': 2,
 'grouping': [127],
 'int_curr_symbol': 'EUR ',
 'int_frac_digits': 2,
 'mon_decimal_point': ',',
 'mon_grouping': [3, 3, 0],
 'mon_thousands_sep': ' ',
 'n_cs_precedes': 0,
 'n_sep_by_space': 1,
 'n_sign_posn': 2,
 'negative_sign': '-',
 'p_cs_precedes': 0,
 'p_sep_by_space': 1,
 'p_sign_posn': 1,
```

```
 'positive_sign': '',
 'thousands_sep': ''}
```

西班牙:

```
$ LANG=es_ES python locale_env_example.py
Environment settings:
 LC_ALL =
 LC_CTYPE =
 LANG = es_ES
 LANGUAGE =

Default locale: (None, 'mac-roman')
From environment: ('es_ES', 'ISO8859-1')
{'currency_symbol': 'Eu',
 'decimal_point': ',',
 'frac_digits': 2,
 'grouping': [127],
 'int_curr_symbol': 'EUR ',
 'int_frac_digits': 2,
 'mon_decimal_point': ',',
 'mon_grouping': [3, 3, 0],
 'mon_thousands_sep': '.',
 'n_cs_precedes': 1,
 'n_sep_by_space': 1,
 'n_sign_posn': 1,
 'negative_sign': '-',
 'p_cs_precedes': 1,
 'p_sep_by_space': 1,
 'p_sign_posn': 1,
 'positive_sign': '',
 'thousands_sep': ''}
```

葡萄牙:

```
$ LANG=pt_PT python locale_env_example.py
Environment settings:
 LC_ALL =
 LC_CTYPE =
 LANG = pt_PT
 LANGUAGE =

Default locale: (None, 'mac-roman')
From environment: ('pt_PT', 'ISO8859-1')
{'currency_symbol': 'Eu',
 'decimal_point': ',',
 'frac_digits': 2,
 'grouping': [127],
 'int_curr_symbol': 'EUR ',
 'int_frac_digits': 2,
 'mon_decimal_point': '.',
 'mon_grouping': [3, 3, 0],
 'mon_thousands_sep': '.',
 'n_cs_precedes': 0,
 'n_sep_by_space': 1,
 'n_sign_posn': 1,
 'negative_sign': '-',
```

```
 'p_cs_precedes': 0,
 'p_sep_by_space': 1,
 'p_sign_posn': 1,
 'positive_sign': '',
 'thousands_sep': ' '}
```

波兰:

```
$ LANG=pl_PL python locale_env_example.py
Environment settings:
 LC_ALL =
 LC_CTYPE =
 LANG = pl_PL
 LANGUAGE =

Default locale: (None, 'mac-roman')
From environment: ('pl_PL', 'ISO8859-2')
{'currency_symbol': 'z?\x82',
 'decimal_point': ',',
 'frac_digits': 2,
 'grouping': [3, 3, 0],
 'int_curr_symbol': 'PLN ',
 'int_frac_digits': 2,
 'mon_decimal_point': ',',
 'mon_grouping': [3, 3, 0],
 'mon_thousands_sep': ' ',
 'n_cs_precedes': 1,
 'n_sep_by_space': 2,
 'n_sign_posn': 4,
 'negative_sign': '-',
 'p_cs_precedes': 1,
 'p_sep_by_space': 2,
 'p_sign_posn': 4,
 'positive_sign': '',
 'thousands_sep': ' '}
```

所以你可以看到货币符号(currency_symbol)设置改变了, 从小数中分离出整个数字的分割字符(decimal_point)也改变了, 等等. 现在以不同的地区设置(US 美元, 欧元, 和Polish złoty)格式输出同样的信息:

```
sample_locales = [ ('USA', 'en_US'),
                   ('France', 'fr_FR'),
                   ('Spain', 'es_ES'),
                   ('Portugal', 'pt_PT'),
                   ('Poland', 'pl_PL'),
                 ]

for name, loc in sample_locales:
    locale.setlocale(locale.LC_ALL, loc)
    print '%20s: %s' % (name, locale.currency(1234.56))
```

输出一个小的表格:

```
$ python locale_currency_example.py
USA: $1234.56
France: 1234,56 Eu
```

```
Spain: Eu 1234,56
Portugal: 1234.56 Eu
Poland: zł 1234,56
```

除了以不同的格式输出外, 本地化模块还可以帮助解析输入. 不同的文化对数字格式化使用不同的转换(上面已列出). 本地化模块提供atoi()和atof()函数分别用来进行字符串与整数和浮点数之间的转换.

```
sample_data = [ ('USA', 'en_US', '1234.56'),
                ('France', 'fr_FR', '1234,56'),
                ('Spain', 'es_ES', '1234,56'),
                ('Portugal', 'pt_PT', '1234.56'),
                ('Poland', 'pl_PL', '1234,56'),
              ]

for name, loc, a in sample_data:
    locale.setlocale(locale.LC_ALL, loc)
    f = locale.atof(a)
    locale.setlocale(locale.LC_ALL, 'en_US')
    print '%20s: %7s => %f' % (name, a, f)
```

```
$ python locale_atof_example.py
USA: 1234.56 => 1234.560000
France: 1234,56 => 1234.560000
Spain: 1234,56 => 1234.560000
Portugal: 1234.56 => 1234.560000
Poland: 1234,56 => 1234.560000
```

另一个本地化的重要方面是时间和日期的格式化:

```
import locale
import time

sample_locales = [ ('USA', 'en_US'),
                   ('France', 'fr_FR'),
                   ('Spain', 'es_ES'),
                   ('Portugal', 'pt_PT'),
                   ('Poland', 'pl_PL'),
                 ]

for name, loc in sample_locales:
    locale.setlocale(locale.LC_ALL, loc)
    print '%20s: %s' % (name, time.strftime(locale.nl_langinfo(locale.D_T_FMT)))
```

```
$ python locale_date_example.py

USA: Sun May 20 10:19:54 2007
France: Dim 20 mai 10:19:54 2007
Spain: dom 20 may 10:19:54 2007
Portugal: Dom 20 Mai 10:19:54 2007
Poland: ndz 20 maj 10:19:54 2007
```

这个星期我只阐述了本地化模块中的一些高层函数. 还有其他低层(格式化字符串)或那些管理你应用程序本地化的函数(resetlocale). 和往常一样, 你可以参考Python库文档来查看些细节.

参考

- Locale - Wikipedia
- Internationalization and localization - Wikipedia
- OpenI18N.org - The Free standards Group Open Internationalisation Initiative
- MSDN - National Language Support Constants
- Internationalizing Python

0.2.20 PyMOTW: urllib

urllib模块提供了一个访问网络资源的简单接口。

- 模块：urllib
- 目的：访问不需要认证的远程资源
- python版本：1.4+

虽然urllib可以与gopher和ftp协议一起使用, 但下面的例子都是用了http协议。

HTTP GET:

这些例子的测试服务器是在BaseHTTPServer_GET.py中, 这个脚本在PyMOTW例子的BaseHTTPServer模块中.在一个终端窗口中启动服务器, 然后在另一个窗口中运行以下这些例子.

HTTP GET 是urllib最简单的操作。简单把URL传递给urlopen()来获取一个用于操作远程数据的类文件句柄。

```
import urllib

response = urllib.urlopen('http://localhost:8080/')
print 'RESPONSE:', response
print 'URL :', response.geturl()

headers = response.info()
print 'DATE :', headers['date']
print 'HEADERS :'
print '---------'
print headers

data = response.read()
print 'LENGTH :', len(data)
print 'DATA :'
print '---------'
print data
```

该示例服务器取得传入的值，并且返回格式化的纯文本response。从urlopen()返回的值通过info()方法给出HTTP服务器的headers的入口，并且通过read()和readlines()等方法获得远程资源的数据。

```
$ python urllib_urlopen.py
RESPONSE: <addinfourl at 10180248 whose fp = <socket._fileobject object at 0x935c30>>
URL : http://localhost:8080/
DATE : Sun, 30 Mar 2008 16:27:10 GMT
```

```
HEADERS :
---------
Server: BaseHTTP/0.3 Python/2.5.1
Date: Sun, 30 Mar 2008 16:27:10 GMT

LENGTH : 221
DATA :
---------
CLIENT VALUES:
client_address=('127.0.0.1', 54354) (localhost)
command=GET
path=/
real path=/
query=
request_version=HTTP/1.0

SERVER VALUES:
server_version=BaseHTTP/0.3
sys_version=Python/2.5.1
protocol_version=HTTP/1.0
```

类文件对象也是可以迭代的:

```
import urllib

response = urllib.urlopen('http://localhost:8080/')
for line in response:
    print line.rstrip()
```

因为返回的每一行都有换行符和完整的框架回车符 -艳 盛 11/21/08 1:31 PM ，所以在输出之前先去掉他们。

```
$ python urllib_urlopen_iterator.py
CLIENT VALUES:
client_address=('127.0.0.1', 54380) (localhost)
command=GET
path=/
real path=/
query=
request_version=HTTP/1.0

SERVER VALUES:
server_version=BaseHTTP/0.3
sys_version=Python/2.5.1
protocol_version=HTTP/1.0
```

编码参数:

将参数编码并且追加在URL之后，传给服务器。

```
import urllib
query_args = { 'q':'query string', 'foo':'bar' }
encoded_args = urllib.urlencode(query_args) ##这个编码, 是将其转换为a=aa&b=bb的形式
print 'Encoded:', encoded_args
```

```
url = 'http://localhost:8080/?' + encoded_args
print urllib.urlopen(url).read()
```

注意query，在客户端的值的列表中包含了已编码的参数query。

```
$ python urllib_urlencode.py
Encoded: q=query+string&foo=bar
CLIENT VALUES:
client_address=('127.0.0.1', 54415) (localhost)
command=GET
path=/?q=query+string&foo=bar
real path=/
query=q=query+string&foo=bar
request_version=HTTP/1.0

SERVER VALUES:
server_version=BaseHTTP/0.3
sys_version=Python/2.5.1
protocol_version=HTTP/1.0
```

在查询字符串中使用单独的变量来传递值序列时, 需传递doseq=True给urlencode()。

```
import urllib
query_args = { 'foo':['foo1', 'foo2'] }
print 'Single :', urllib.urlencode(query_args)
print 'Sequence:', urllib.urlencode(query_args, doseq=True)
```

```
$ python urllib_urlencode_doseq.py
Single : foo=%5B%27foo1%27%2C+%27foo2%27%5D
Sequence: foo=foo1&foo=foo2
```

为了解码查询字符串，可查看cgi模块中的FieldStorage类。

在查询参数里的一些特别字符，在传递给urlencode()后，在服务器端可能和URL一起引起解析错误。可以直接使用quote()或者quote_plus()函数在本地引用他们以生成安全的字符串。

```
import urllib

url = 'http://localhost:8080/~dhellmann/'
print 'urlencode() :', urllib.urlencode({'url':url})
print 'quote() :', urllib.quote(url)
print 'quote_plus():', urllib.quote_plus(url)
```

Note: quote_plus()能够替换更多的特殊字符。

```
$ python urllib_quote.py
urlencode() : url=http%3A%2F%2Flocalhost%3A8080%2F%7Edhellmann%2F
quote() : http%3A//localhost%3A8080/%7Edhellmann/
quote_plus(): http%3A%2F%2Flocalhost%3A8080%2F%7Edhellmann%2F
```

视情况而定，用unquote()或者unquote_plus()来还原quote操作。

```
import urllib
```

```
print urllib.unquote('http%3A//localhost%3A8080/%7Edhellmann/')
print urllib.unquote_plus('http%3A%2F%2Flocalhost%3A8080%2F%7Edhellmann%2F')
```

```
$ python urllib_unquote.py
http://localhost:8080/~dhellmann/
http://localhost:8080/~dhellmann/
```

HTTP POST:

这些例子的测试服务器是在BaseHTTPServer_POST.py中, 这个脚本在PyMOTW例子的BaseHTTPServer模块中.在一个终端窗口中启动服务器, 然后在另一个窗口中运行以下这些例子.

通过POST代替GET方式传递数据给远程服务器，仅仅是把已编码的查询参数当作数据传递给urlopen().

```
import urllib
query_args = { 'q':'query string', 'foo':'bar' }
encoded_args = urllib.urlencode(query_args)
url = 'http://localhost:8080/'
print urllib.urlopen(url, encoded_args).read()
```

```
$ python urllib_urlopen_post.py
Client: ('127.0.0.1', 54545)
Path: /
Form data:
     q=query string
     foo=bar
```

如果服务器需要的不是已编码url形式的参数，你可以传递任一字节字符串作为发送的数据。

Paths vs. URLs:

一些操作系统使用不同的方法分离本地文件路径和URL。为了使代码简捷，你应该反复地使用函数pathname2url() 和 url2pathname()。因为我在Mac上工作，我必须明确引入Windows上的函数版本。使用由urllib导出的函数版本可以让你默认在正确平台下，因此就不用自己做了。

```
import os

from urllib import pathname2url, url2pathname

print '== Default =='
path = '/a/b/c'
print 'Original:', path
print 'URL :', pathname2url(path)
print 'Path :', url2pathname('/d/e/f')
print

from nturl2path import pathname2url, url2pathname

print '== Windows, without drive letter =='
path = path.replace('/', '\\')
print 'Original:', path
print 'URL :', pathname2url(path)
print 'Path :', url2pathname('/d/e/f')
```

```
print

print '== Windows, with drive letter =='
path = 'C:\\' + path.replace('/', '\\')
print 'Original:', path
print 'URL :', pathname2url(path)
print 'Path :', url2pathname('/d/e/f')
```

有两个Windows例子，分别是路径的前缀中有和没有驱动器名。

```
$ python urllib_pathnames.py
== Default ==
Original: /a/b/c
URL : /a/b/c
Path : /d/e/f

== Windows, without drive letter ==
Original: \a\b\c
URL : /a/b/c
Path : \d\e\f

== Windows, with drive letter ==
Original: C:\\a\b\c
URL : ///C|/a/b/c
Path : \d\e\f
```

带Cache简单检索：

检索数据是常见的操作，urllib包括urlretrieve()函数，因此你不用自己写它。urlretrieve()带有URL中的参数, 一个用于存储数据的临时文件, 一个用于报告下载进度的函数, 和URL中要POST数据。如果没有给定文件名，urlretrieve()就建立一个临时文件。你自己能删除它，或者把它看作一个cache，可以用urlcleanup()移除它。

这个例子使用GET从web服务器中检索数据。

```
import urllib
import os

def reporthook(blocks_read, block_size, total_size):
    if not blocks_read:
        print 'Connection opened'
        return
    if total_size < 0:
        # Unknown size 未知大小
        print 'Read %d blocks' % blocks_read
    else:
        amount_read = blocks_read * block_size
        print 'Read %d blocks, or %d/%d' % (blocks_read, amount_read, total_size)
        return


try:
    filename, msg = urllib.urlretrieve('http://blog.doughellmann.com/', reporthook=reporthook)
    print
    print 'File:', filename
    print 'Headers:'
```

```
    print msg
    print 'File exists before cleanup:', os.path.exists(filename)
finally:
    urllib.urlcleanup()
    print 'File still exists:', os.path.exists(filename)
```

由于服务器没有返回header中的Content-length，urlretrieve()不知道数据应该有多大，所以将-1传给reporthook()中的参数total_size。

```
$ python urllib_urlretrieve.py
Connection opened
Read 1 blocks
Read 2 blocks
Read 3 blocks
Read 4 blocks
Read 5 blocks
Read 6 blocks
Read 7 blocks
Read 8 blocks
Read 9 blocks
Read 10 blocks
Read 11 blocks
Read 12 blocks
Read 13 blocks
Read 14 blocks
Read 15 blocks
Read 16 blocks
Read 17 blocks
Read 18 blocks
Read 19 blocks

File: /var/folders/9R/9R1t+tR02Raxzk+F71Q50U+++Uw/-Tmp-/tmp3HRpZP
Headers:
Content-Type: text/html; charset=UTF-8
Last-Modified: Tue, 25 Mar 2008 23:09:10 GMT
Cache-Control: max-age=0 private
ETag: "904b02e0-c7ff-47f6-9f35-cc6de5d2a2e5"
Server: GFE/1.3
Date: Sun, 30 Mar 2008 17:36:48 GMT
Connection: Close

File exists before cleanup: True
File still exists: False
```

URLopener:

urllib提供了一个URLopener基类，并且默认使用FancyURLopener处理支持的协议。如果你想要改变其行为，你可能需要查看Python2.1中新加的urllib2模块（PyMOTW将会阐述）。

参考

- RFC 2616 - HTTP Specification
- cgi - For decoding query arguments

- PyMOTW: BaseHTTPServer
- urllib2 - For more complex URL access needs
- Python Module of the Week Home

0.2.21 PyMOTW: logging

- 模块：logging
- 目的：为python模块提供状态、错误、信息输出的标准接口
- python版本：2.3

描述

logging模块定义了一个标准API, 用于报告所有你使用的模块的错误和状态信息.标准库模块中提供logging API的最重要意义是所有python模块可以参与到日志记录中, 因此你的应用程序日志可以包含来自第三方模块的信息.

当然, 在不同层次上或因不同目的来记录日志信息是有必要的. 将日志信息写入到文件, 如，HTTP GET/POST的地理信息, 通过SMTP发送的邮件, 一般的sockets, 或者特定OS的日志机制都是被标准模块支持的. 如果你有特殊需求, 任何内置模块都不能满足的话, 你也可以创建你自己的日志目标类.

例子

大多数应用程序可能会将日志写入到文件中, 所以让我从这个例子开始讲述. 我们使用basicConfig()函数来设置默认的处理用于将调试信息写入到文件.

```
import logging
LOG_FILENAME = '/tmp/logging_example.out'
logging.basicConfig(filename=LOG_FILENAME,level=logging.DEBUG,)

logging.debug('This message should go to the log file')
```

现在如果我们打开这个文件, 看看里面是什么, 我们应该可以找到以下的日志信息:

```
f = open(LOG_FILENAME, 'rt')
try:
    body = f.read()
finally:
    f.close()
    print 'FILE:'
    print body
    print


FILE:
DEBUG:root:This message should go to the log file
```

如果我们重复运行之前的脚本, 那么另外的日志信息会附加到文件末尾. 为了每次能够创建一个新的文件, 你可以传递一个filemode参数值为 'w' 给basicConfig().尽管你自己不能控制这个日志文件大小, 但, 可以使用RotatingFileHandler, 这更方便:

```
import glob
import logging
import logging.handlers

LOG_FILENAME = '/tmp/logging_rotatingfile_example.out'

# 设置一个我们希望的输出层次的日志记录器.
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

# 增加记录器的日志消息处理
handler = logging.handlers.RotatingFileHandler(LOG_FILENAME, maxBytes=20, backupCount=5)
my_logger.addHandler(handler)

for i in range(20):
    my_logger.debug('i = %d' % i)

# 查看新创建的文件
logfiles = glob.glob('%s*' % LOG_FILENAME)

for filename in logfiles:
    print filename
```

结果应该是6个独立的文件, 每个都含有应用程序的日志历史:

```
/tmp/logging_rotatingfile_example.out
/tmp/logging_rotatingfile_example.out.1
/tmp/logging_rotatingfile_example.out.2
/tmp/logging_rotatingfile_example.out.3
/tmp/logging_rotatingfile_example.out.4
/tmp/logging_rotatingfile_example.out.5 ##生成5个备份是由于之前设置了backupCount=5
```

当前日志文件总是为/tmp/logging_rotatingfile_example.out, 每次当文件大小达到限制时, 就以后缀.1来重命名. 每个已存的备份文件也依次重命名为原先后缀增一(如, .1成为.2), .5文件会被擦除.

显然的, 这个例子中设置了日志的长度太太太小了. 所以在实际程序下, 你可以为maxBytes设置一个合适的值.

使用日志API的另外一个有用的地方是能够在不同日志层次上产生不同的信息. 这能够让你书写的代码中带有调试信息, 例如, 降低日志层次以便这些调试信息不输出到你的生产系统中.

```
CRITICAL 50
ERROR 40
WARNING 30
INFO 20
DEBUG 10
UNSET 0
```

日志记录器, handler, 日志信息可以分别调用不同的层次 . 一条日志信息, 只有当处理和日志记录器被设置为和它一样的层次或比它低层次时, 才被输出. 例如, 如果一个信息是CRITICAL, 记录器被设置为ERROR, 那么这个消息会输出来. 如果一个信息是WARNING, 记录器被设置为ERROR, 那么这个信息不被输出.

```
import logging
import sys

LEVELS = { 'debug':logging.DEBUG,
           'info':logging.INFO,
           'warning':logging.WARNING,
```

```
            'error':logging.ERROR,
            'critical':logging.CRITICAL,
}

if len(sys.argv) > 1:
    level_name = sys.argv[1]
    level = LEVELS.get(level_name, logging.NOTSET)
    logging.basicConfig(level=level)

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical error message')
```

运行这个脚本时指定参数, 如 'debug' 或 'warning', 看看在不同层次上, 哪些信息会显示出来:

```
$ python logging_level_example.py debug
DEBUG:root:This is a debug message
INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical error message

$ python logging_level_example.py info
INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical error message
```

你可能不会注意到这些日志信息中都含有 'root'. 这个日志模块支持一个不同名字日志记录器的层次结构. 一个告知某条日志信息来自于哪个日志器的简单方式是对每个模块使用独立的日志器对象. 每个新的日志器从它的父亲中"继承"一些配置, 日志信息发送到一个包含父日志器名字的日志器. 可选的, 每个日志器可以配置不同, 以便让来自不同模块的信息按不同的方式处理. 让我们看个简单的例子看怎样记录来自不同模块的信息, 这也便于追踪信息的对应源代码:

```
import logging

logging.basicConfig(level=logging.WARNING)

logger1 = logging.getLogger('package1.module1')
logger2 = logging.getLogger('package2.module2')

logger1.warning('This message comes from one module')
logger2.warning('And this message comes from another module')
```

输出为:

```
$ python logging_modules_example.py
WARNING:package1.module1:This message comes from one module
WARNING:package2.module2:And this message comes from another module
```

还有许多许多许多配置日志记录的选项, 包括不同日志信息格式化选项, 将信息发送到多个记录器, 使用socket接口改变一个正在运行的长时间程序的配置. 所有这些选项进一步在 库模块文档 中深入.

参考

- PEP 282
- Python Standard Logging

0.2.22 PyMOTW: bisect

- 模块：bisect
- 目的：维持一个有序列表，当每次增加一个元素到列表时无需调用sort过程。
- python版本：1.4+

描述

bisect模块实现了一个算法，用于向一个有序列表中插入一个元素。这比重复排序一个列表, 或重构一个很大的有序列表要高效的多.

示例

使用 `bisect.insort()` 的简单示例，插入元素到一个有序列表中。

```
import bisect
import random

# 设置一个常数种子, 这在以后的循环中, 产生同样的伪随机数
random.seed(1)
# 产生20个随机数, 并依次插入到有序列表中
l = []
for i in range(1, 20):
    r = random.randint(1, 100)
    position = bisect.bisect(l, r)
    bisect.insort(l, r)
    print '%2d %2d' % (r, position), l
```

上述脚本的输出如下：

```
14  0 [14]
85  1 [14, 85]
77  1 [14, 77, 85]
26  1 [14, 26, 77, 85]
50  2 [14, 26, 50, 77, 85]
45  2 [14, 26, 45, 50, 77, 85]
66  4 [14, 26, 45, 50, 66, 77, 85]
79  6 [14, 26, 45, 50, 66, 77, 79, 85]
10  0 [10, 14, 26, 45, 50, 66, 77, 79, 85]
3  0 [3, 10, 14, 26, 45, 50, 66, 77, 79, 85]
84  9 [3, 10, 14, 26, 45, 50, 66, 77, 79, 84, 85]
44  4 [3, 10, 14, 26, 44, 45, 50, 66, 77, 79, 84, 85]
77  9 [3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]
1  0 [1, 3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]
45  7 [1, 3, 10, 14, 26, 44, 45, 45, 50, 66, 77, 77, 79, 84, 85]
73 10 [1, 3, 10, 14, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85]
23  4 [1, 3, 10, 14, 23, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85]
```

```
95 17 [1, 3, 10, 14, 23, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85, 95]
91 17 [1, 3, 10, 14, 23, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85, 91, 95]
```

第一列显示了新的随机数，第二列显示了数被插入到列表中的位置。最后是当前排序列表中的元素。

这是一个很简单的示例，我们处理的速度由于列表规模小以及每次只需排序一次, 变的非常快速。但对于一个很长的list，利用这种方法能得到时间和内存上的节省。

你可能会注意到上述结果中存在一些重复值（45和77）.bisect模块提供了2种方法来处理重复，新值可以插入到已经存在值的左边或者右边。对应的是 insort_right()函数，可以将值插入已有值的后面（右边），insort_left()函数可以插入到之前（左边）。

如果我们使用bisect_left()和bisect_right()来处理同样的数据，那么最后获得的list是相同的，只是中间插入的位置会有不同。

```
# 重设种子
random.seed(1)

# 使用bisect_left 和 insort_left
l = []
for i in range(1, 20):
    r = random.randint(1, 100)
    position = bisect.bisect_left(l, r)
    bisect.insort_left(l, r)
    print '%2d %2d' % (r, position), l
```

```
14 0 [14]
85 1 [14, 85]
77 1 [14, 77, 85]
26 1 [14, 26, 77, 85]
50 2 [14, 26, 50, 77, 85]
45 2 [14, 26, 45, 50, 77, 85]
66 4 [14, 26, 45, 50, 66, 77, 85]
79 6 [14, 26, 45, 50, 66, 77, 79, 85]
10 0 [10, 14, 26, 45, 50, 66, 77, 79, 85]
3 0 [3, 10, 14, 26, 45, 50, 66, 77, 79, 85]
84 9 [3, 10, 14, 26, 45, 50, 66, 77, 79, 84, 85]
44 4 [3, 10, 14, 26, 44, 45, 50, 66, 77, 79, 84, 85]
77 8 [3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]
1 0 [1, 3, 10, 14, 26, 44, 45, 50, 66, 77, 77, 79, 84, 85]
45 6 [1, 3, 10, 14, 26, 44, 45, 45, 50, 66, 77, 77, 79, 84, 85]
73 10 [1, 3, 10, 14, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85]
23 4 [1, 3, 10, 14, 23, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85]
95 17 [1, 3, 10, 14, 23, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85, 95]
91 17 [1, 3, 10, 14, 23, 26, 44, 45, 45, 50, 66, 73, 77, 77, 79, 84, 85, 91, 95]
```

除了python实现外，还有一个更快的c实现，如果c版本存在，那么 `import bisect` 模块时会自动调用c版本而不是调用python版本。

参考

- Insertion Sort

0.2.23 PyMOTW: ConfigParser

- 模块：ConfigParser
- 目的：读取/写入配置文件,类似于Windows的INI文件
- python版本：1.5+

描述

ConfigParser模块可以为你的应用程序创建用户可编辑的配置文件. 这个配置文件由一个个节组成,每个节可以包含配置数据的名字-值对.支持通过使用Python的格式化字符串进行值的插入, 以此来构建那些依赖于其他值的数据值(这对路径或URL来说是尤其方便的).

在工作中,当我们把东西移动到svn和 trac 之前, 我们开发推出了自己的用于进行分布式代码复查的工具. 为了准备好需要复查的代码,一个开发者常常需要写完一个"approach"摘要文件,然后附上被修改后的代码(即和原代码有区别的地方).这个 approach文档支持通过Web页面添加注释, 因此开发者不在我们的主要办公室里也可以复查代码. 但是唯一的麻烦之处是, 发表代码的不同之处让人感到有点痛苦. 想要让部分处理过程变得简单些,我写了一个命令行工具, 运行他可以针对CVS沙盒,自动的寻找出并发表代码的不同之处.

为了能够让这个工具即时更新approach文档中的区别, 需要知道怎样到达存放approach文档的网络服务器. 由于我们开发者不总是在办公室, 从任意给定主机到达服务器的URL可能是通过SSH端口转发过来的. 为了不强迫每个开发者都使用同样的端口转发协议?这个工具应使用一个简单的配置文件来记住这个URL.

一个开发者的配置文件可能会是这个样子:

```
[portal]
url = http://%(host)s:%(port)s/Portal
username = dhellmann
host = localhost
password = SECRET
port = 8080
```

portal小节表示approach文件的网址. 一旦代码的区别被发送到这个网址, 我们的工具应该下载这个配置文件,通过ConfigParser模块来访问URL.这可能看起来像这样:

```
from ConfigParser import ConfigParser
import os

filename = os.path.join(os.environ['HOME'], '.approachrc')

config = ConfigParser()
config.read([filename])

url = config.get('portal', 'url')
```

上述的例子中, 变量url的值为"http://localhost:8080/Portal".配置文件中的变量url中包含两个格式化字符串:"%(host)s" 和 "%(port)s". 通过 `get()` 方法, 自动地将变量host和port的值替换到格式化字符串中.

当然, 这是基于Python2.1的旧代码. 在近来的版本中,ConfigParser模块已经被改进了很多.SafeConfigParser类是a drop?用来取代ConfigParser, 以改善插值处理.

对于这个工具,我只需要字符串选项. ConfigParser支持其他的选项类型:整型, 浮点型和布尔型. 由于可选文件格式不提供直接使用一个值来关联一种类型的方式,所以调用者需要知道何时需要使用一种不同的函数来查询那些其他类型的选项.例如,为了查找一个布尔选项, 使用 `getboolean()` 函数而不是 `get()` 函数.函数的参数是一样的,但是选项的值在返回之前被转换为一个布尔类型.类似地,还有独立的 `getint()` 和 `getfloat()` 函数.

ConfigParser类也支持增加和删除小节到指定文件并保存结果.这使得创建一个用于编辑程序的配置的用户界面,或是利用配置文件格式存放简单数据文件成为可能. 例如, 一个应用需要存储小量的类似于数据库格式的数据, 可以利用ConfigParser类,这样一来生成的文件也是人类可读的.

0.2.24 PyMOTW: shelve

- 模块：shelve
- 目的：shelve模块实现了对任意可被pickle的Python对象进行持久性存储，也提供类字典API给我们使用。
- python版本：1.4+

描述

当使用关系数据库是一种浪费的时候，shelve模块可以为Python对象提供一个简单的持久性存储选择。就像使用字典一样，通过关键字访问shelf对象。其值经过pickle，写入到由anydbm创建和管理的数据库。

创建一Shelf对象

最简单的使用shelve模块的方式是通过DbfilenameShelf类。 使用函数 `shelve.open()` （使用的是 anydbm ）来存储数据。你可以直接使用类，或者简单的调用：

```
import shelve

s = shelve.open('test_shelf.db')
try:
    s['key1'] = { 'int': 10, 'float':9.5, 'string':'Sample data' }
finally:
    s.close()
```

为了再次访问数据，打开shelf并且像字典一样使用它。

```
s = shelve.open('test_shelf.db')
try:
    existing = s['key1']
finally:
    s.close()

print existing
```

如果你运行了上面两个脚本，你应该看到：

```
$ python shelve_create.py
$ python shelve_existing.py
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

dbm模块不支持多个应用同时写入同一数据库。如果你确定客户端不会修改shelf， 请指定shelve以只读方式打开数据库。

```
s = shelve.open('test_shelf.db', flag='r')
try:
    existing = s['key1']
```

```
finally:
    s.close()

print existing
```

当数据库以只读方式打开时，你又尝试着更改数据库，这将引起一个访问出错异常。这一异常类型依赖于在创建数据库时被anydbm选择的数据库模块。

写回

默认情况下，Shelves不去追踪可变对象的修改。意思就是，如果你改变了已存储在shelf中的一个项目的内容，就必须重新存储该项目来更新shelf。

```
s = shelve.open('test_shelf.db')
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
finally:
    s.close()
```

在这个例子中，没有对字典里的关键字"key1"的内容进行存储，因此，重新打开shelf的时候，还没保存所做的改变。

```
$ python shelve_create.py
$ python shelve_withoutwriteback.py
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
```

为了自动捕捉储存在shelf中的可变对象所发生的改变，置writeback功能可用。writeback标志导致shelf使用一缓存来记住从数据库中调出的所有对象 。当shelf关闭的时候，每一个缓存中的对象也重新写入数据库。

```
s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
    s['key1']['new_value'] = 'this was not here before'
    print s['key1']
finally:
    s.close()

s = shelve.open('test_shelf.db', writeback=True)
try:
    print s['key1']
finally:
    s.close()
```

虽然使用writeback模式可以减少程序员出错机率，也能更加透明化对象持久性，但是，不是每种情况都要使用writeback 模式的。当shelf打开的时候，缓存就要占据额外的空间，并且，当shelf关闭的时候，也会花费额外的时间去将所有缓存中的对象写入到数据库中。即使不知道缓存中的对象有没有改变，都要写回数据库。如果你的应用程序读取数据多于写入数据，那么writeback模式将增大开销。

```
$ python shelve_create.py
$ python shelve_writeback.py
{'int': 10, 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
{'int': 10, 'new_value': 'this was not here before', 'float': 9.5, 'string': 'Sample data'}
```

指定Shelf类型

上面的例子全都使用了默认的shelf实现。使用 `shelve.open()` 直接代替一种shelf实现，是常见用法，尤其是在不关心用哪种数据库存储数据的时候。然而，有时常会关心它。如果是在这种情况下，通常就会直接使用DbfilenameShelf或者BsdDbShelf，更或者是通过Shelf的子类来解决问题。

参考

- feedcache uses shelve as a default storage option

0.2.25 PyMOTW: StringIO and cStringIO

- 模块：StringIO 和 cStringIO
- 目的: 类似于file操作的文本缓冲区API
- python版本：StringIO: 1.4+, cStringIO: 1.5+

描述

类StringIO提供了一个在内存中方便处理文本的类文件(读, 写等操作)API. 他有两个独立的实现, 一个是用c实现的cStringIP模块,速度较快, 另一个是StringIO模块,他用python实现的以增强其可移植性. 使用cStringIO来处理大字符串可以提高运行性能,优于其他字符串串联技术.

例子

这里是一个好的,标准的,简单的.使用StringIO缓冲的例子:

```
#!/usr/bin/env python

"""
使用StringIO模块简单的例子
"""

# 找到此平台上最好的StringIO实现模块

try:
    from cStringIO import StringIO
except:
    from StringIO import StringIO

# 写入一个缓冲区
output = StringIO()
output.write('This goes into the buffer. ') ##write方法将字符串写入缓冲区
```

```
print >>output, 'And so does this.' ##print 语句将字符串写入output
# 查询刚才被写入缓冲区的值
print output.getvalue()

output.close() # 关闭(丢弃)缓冲区

# 初始化一个只读缓冲区
input = StringIO('Inital value for read buffer')

# 从缓冲区中读取数据
print input.read()
```

这个例子中使用了 `read()` ,但当然,函数 `readline()` 和 `readlines()` 也都是可用的.类StringIO提供 `seek()` 函数,因此在读取数据时可以任意跳到某个点上, 这也当你使用某些前看解析算法可以回头读取.

现实世界中,StringIO的应用包括一个网络应用程序栈,栈的各个部分都可以增加文本到响应response对象中,或者测试由程序某段输出(典型是写入到文件)的数据.

我们想在创建的工程应用中包括一个shell脚本接口,他是以多个命令行程序的形式.这些程序中的某些是负责从数据库中取数据,然后将这些数据转储到控制台(可以是现实给用户,也可以是文本中,这样就可以作为另一个命令的输入). 这些命令他们共享了一组格式化插件以便产生一个对象的多种文本表示(XML, bash语法, 人可读的,等等).

因为格式化可以将输出数据标准化后写到标准输出,如果没有StringIO模块,所得的测试结果可能会有些奇怪. 而使用了StringIO来拦截格式化的输出,这样以便我们用一个简单的方式来收集内存中的输出数据,来对预期的结果做比较.

参考

- The StringIO module ::: www.effbot.org
- Efficient String Concatenation in Python

0.3 关于 PyMOTW

0.4 一些问题

0.5 版权所有

0.6 许可证

0.7 索引表

- 索引
- 模块索引
- 搜索页面