nsd1902_py02_day03

tarfile模块

实现打包压缩、解压缩，同时支持gzip / bzip2 / xz格式

```
# 压缩
>>> import tarfile
>>> tar = tarfile.open('/tmp/mytest.tar.gz', 'w:gz')  # 用gzip压缩
>>> tar.add('/etc/hosts')   # 压缩单个文件
>>> tar.add('/etc/security')  # 压缩目录
>>> tar.close()   # 关闭

# 解压
>>> tar = tarfile.open('/tmp/mytest.tar.gz')  # 自动识别压缩格式，以读方式打开
>>> tar.extractall(path='/var/tmp')  # 指定解压目录，默认解到当前目录
>>> tar.close()
```

备份程序

```
[root@room8pc16 day03]# mkdir /tmp/demo
[root@room8pc16 day03]# cp -r /etc/security/ /tmp/demo
[root@room8pc16 demo]# mkdir /tmp/demo/backup
```

- 将/tmp/demo/security备份到/tmp/demo/backup中
- 需要支持完全和增量备份
- 周一执行完全备份
- 其他时间执行增量备份

分析：

- 完全备份需要执行备份目录、计算每个文件的md5值
- 增量备份需要计算文件的md5值，把md5值与前一天的md5值比较，有变化的文件要备份；目录中新增的文件也要备份
- 备份的文件名，应该体现出：备份的是哪个目录，是增量还是完全，哪一天备份的

获取/tmp/demo/security目录下每个文件的绝对路径：

```
>>> import os
>>> os.walk('/tmp/demo/security')
<generator object walk at 0x7f6306b7edb0>
>>> files = list(os.walk('/tmp/demo/security'))
>>> import pprint
>>> pprint.pprint(files)
[('/tmp/demo/security',
  ['console.apps', 'console.perms.d', 'limits.d', 'namespace.d'],
  ['access.conf',
   'chroot.conf',
```

```
   'console.handlers',
   'console.perms',
   'group.conf',
   'limits.conf',
   'namespace.conf',
   'namespace.init',
   'opasswd',
   'pam_env.conf',
   'sepermit.conf',
   'time.conf',
   'pwquality.conf']),
 ('/tmp/demo/security/console.apps',
  [],
  ['config-util', 'xserver', 'liveinst', 'setup']),
 ('/tmp/demo/security/console.perms.d', [], []),
 ('/tmp/demo/security/limits.d', [], ['20-nproc.conf']),
 ('/tmp/demo/security/namespace.d', [], [])]
>>> len(files)
5
>>> files[0]
('/tmp/demo/security', ['console.apps', 'console.perms.d', 'limits.d', 'namespace.d'],
['access.conf', 'chroot.conf', 'console.handlers', 'console.perms', 'group.conf',
'limits.conf', 'namespace.conf', 'namespace.init', 'opasswd', 'pam_env.conf',
'sepermit.conf', 'time.conf', 'pwquality.conf'])
>>> files[1]
('/tmp/demo/security/console.apps', [], ['config-util', 'xserver', 'liveinst', 'setup'])
>>> files[2]
('/tmp/demo/security/console.perms.d', [], [])
# 经过分析，发现ow.walk它的返回值由多个元组构成。元组有三项，第一项是路径字符串，第二项是该路径下的目录列表，第三项是该目录下的文件列表。
# 要获得每个文件的绝对路径，只要把路径字符串和文件拼接起来即可
>>> for path, folders, files in os.walk('/tmp/demo/security'):
...   for file in files:
...     os.path.join(path, file)
...
'/tmp/demo/security/access.conf'
'/tmp/demo/security/chroot.conf'
'/tmp/demo/security/console.handlers'
'/tmp/demo/security/console.perms'
'/tmp/demo/security/group.conf'
'/tmp/demo/security/limits.conf'
'/tmp/demo/security/namespace.conf'
'/tmp/demo/security/namespace.init'
'/tmp/demo/security/opasswd'
'/tmp/demo/security/pam_env.conf'
'/tmp/demo/security/sepermit.conf'
'/tmp/demo/security/time.conf'
'/tmp/demo/security/pwquality.conf'
'/tmp/demo/security/console.apps/config-util'
'/tmp/demo/security/console.apps/xserver'
'/tmp/demo/security/console.apps/liveinst'
'/tmp/demo/security/console.apps/setup'
'/tmp/demo/security/limits.d/20-nproc.conf'
```

OOP：面向对象的编程

OOP实现了数据和行为的融合。在编程的时候，首先把现实世界中的事物抽象成一个类（class），该类拥有数据属性（相当于以前的变量），还拥有行为属性（相当于前面的函数）。然后，具体的行为可以根据class创建实例，实例自动拥有了class定义时的属性。

```
class Warrior:
    def __init__(self, name, weapon):
        '实例化时自动调用'
        self.name = name
        self.weapon = weapon

    def speak(self, words):
        print("I'm %s, %s" % (self.name, words))

    def show_me(self):
        print("我是%s, 我是一个战士" % self.name)

if __name__ == '__main__':
    gy = Warrior('关羽', '青龙偃月刀')
    gy.speak('过五关，斩六将')
    gy.show_me()
```

class定义时，类名推荐每个单词首字母是大写的。一般来说，class都有一个特殊的方法叫__init__，当实例化时，它自动调用，这个方法通常用于为实例绑定数据属性。绑定到实例上的属性，在类的任何方法中都可用。在类中定义的其他函数（称为方法），也绑定到了实例上，实例可以通过 「实例.属性」 的方式进行调用。

self不是关键字，名字可以是任意合法的字符串，但是习惯于写为self，表示实例而已。

在python中，一切对象。实际上，字符串是str的实例，列表是list的实例。

```
>>> s = 'hello'
>>> type(s)
<class 'str'>
>>> alist = [1, 2]
>>> type(alist)
<class 'list'>
```

组合

OOP编程常用方式之一。当两个类有明显的不同，其中一个类是另一个类的组件时，用组合。

继承

OOP编程常用方式之一。当两个类有非常多的相似之处，只有部分不一致的行为，用继承。

多重继承

子类可以有多个父类（基类），子类拥有所有父类的方法。如果有重名方法，查找方法的顺序是自下向上，自左向右。

```
class A:
    def func1(self):
        print('A func')

    def func3(self):
        print('A func3')

class B:
    def func2(self):
        print('B func')

    def func3(self):
        print('B func3')

class C(B, A):
    def func3(self):
        print('C func3')

if __name__ == '__main__':
    c1 = C()
    c1.func1()
    c1.func2()
    c1.func3()
```

特殊的方法

在定义class时，内部有一些以双下划线开头、结尾的特殊方法，被称作魔法方法（magic）。

re模块

正则表达式示例

有以下文本

```
192.168.1.1     000C29123456
192.168.1.2     525400A3B282
192.168.1.3     000C28123AC8
```

将第二列的mac地址改为以冒号分隔的形式。

思路：

1. 取出MAC地址
2. MAC地址分成6组
3. 在组之间加冒号

```
:%s/\(..\)\(..\)\(..\)\(..\)\(..\)\(..\)$/\1:\2:\3:\4:\5:\6/
```

re模块的应用

```
>>> import re
```

```
>>> re.match('f..', 'food')   # 匹配到返回匹配对象
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> re.match('f..', 'seafood')
>>> print(re.match('f..', 'seafood'))  # 匹配不到返回None
None

# re.match从字符串开头匹配
>>> m = re.match('f..', 'food')
>>> m.group()   # 通过对象的group方法取出匹配到的内容
'foo'

# search可以在任意位置匹配
>>> re.search('f..', 'food')
<_sre.SRE_Match object; span=(0, 3), match='foo'>
>>> re.search('f..', 'seafood')
<_sre.SRE_Match object; span=(3, 6), match='foo'>
>>> m = re.search('f..', 'seafood')
>>> m.group()
'foo'

# findall可以匹配到全部对象，返回列表
>>> re.findall('f..', 'seafood is food')
['foo', 'foo']

# finditer匹配到的对象放到生成器中
>>> list(re.finditer('f..', 'seafood is food'))
[<_sre.SRE_Match object; span=(3, 6), match='foo'>, <_sre.SRE_Match object; span=(11, 14),
match='foo'>]
>>> for m in re.finditer('f..', 'seafood is food'):
...     print(m.group())
...
foo
foo

# 以空格和.作为分割符分割字符串
>>> re.split(' |\.', 'hello greet welcome.ni.hao')
['hello', 'greet', 'welcome', 'ni', 'hao']

# 替换，把X换成bob
>>> re.sub('X', 'bob', 'Hi X. ni hao X')
'Hi bob. ni hao bob'

# 当有大量匹配的时候，提前把正则表达式的模式字符编译，会有更好的执行效率
>>> cpatt = re.compile('f..')
>>> m = cpatt.search('seafood')
>>> m.group()
'foo'
>>> cpatt.findall('food is seafood')
['foo', 'foo']
```