NSD Python DAY02

1 案例1：斐波那契数列

1.1 问题

编写fib.py脚本，主要要求如下：

- 输出具有10个数字的斐波那契数列
- 使用for循环和range函数完成
- 改进程序，要求用户输入一个数字，可以生成用户需要长度的斐波那契数列

1.2 方案

斐波那契数列就是某一个数，总是前两个数之和，比如0，1，1，2，3，5，8。由于输出是一串数字，可以用列表的结构存储。开始时，列表中有两个值，即0，1。然后通过循环向列表中追加元素，追加元素总是列表中最后两个元素值之和。

本例使用的是列表，不能使用元组，因为列表是一个可变类型，而元组是不可变类型。各种数据类型的比较如下：

按存储模型分类

标量类型：数值、字符串

容器类型：列表、元组、字典

按更新模型分类：

可变类型：列表、字典

不可变类型：数字、字符串、元组

按访问模型分类

直接访问：数字

顺序访问：字符串、列表、元组

映射访问：字典

由于循环次数是确定的，可以使用for循环。python中的for循环与传统的for循环（计数器循环）不太一样，它更象shell脚本里的foreach迭代。python中的for接受可迭代对象（例如序列或迭代器）作为其参数，每次迭代其中一个元素。

for循环经常与range()函数一起使用。range函数语法如下：

```
01.    range([start,] stop[, step])
```

Top

range函数将返回一个列表，如果列表没有给定起始值，那么起始值从0开始，结束值是给定的结束值的前一个值，另外还可以设置步长值。例：

```
>>> range(10)          #没有给定起始值则从0开始，结束值为9
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,10)       #给定起始值，列表的第一个值就是给定的起始值
[5, 6, 7, 8, 9]
>>> range(0, 10, 2)   #给定步长值为2，列出10以内的偶数
[0, 2, 4, 6, 8]
>>> range(1, 10, 2)   #给定起始值、步长值，列出10以内的奇数
[1, 3, 5, 7, 9]
```

当将脚本改成交互式运行时，需要注意，用户输入的数字，python仍然将其视为字符而不是整型数字，需要使用int()将用户输入做类型转换。

在进行运算时，应使得参与运算的对象属于同一类型，否则如果python无法进行隐匿类型转换将会出现异常。

```
>>> '30' + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

上面的代码之所以出现错误，是因为不能将字符串与数字进行加法或拼接运算。如果希望将其拼接，可以使用下面的方式：

```
>>> '30' + str(3)
'303'
```

如果希望将其全作为数字进行相加，可以使用下面的方式：

```
>>> int('30') + 3
33
```

1.3 步骤

实现此案例需要按照如下步骤进行。

步骤一：编写固定输出的斐波那契数列

Top

```
[root@py01 bin]# vim fibs.py
```

```
02.
03.  #! /usr/bin/env python
04.
05.  fibs = [ 0, 1]
06.
07.  for i in range( 8) :
08.      fibs.append( fibs[ - 1] + fibs[ - 2] )
09.
10.  print fibs
```

执行结果如下：

```
01.  [ root@py01 bin] # ./fibs.py
02.  [ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

步骤二：将脚本改为交互式执行

如果希望生成用户指定个数的数列，需要与用户进行交互，通过raw_input()函数读取用户指定的长度。因为raw_input()函数读入的数据均为字符串，所以需要对其进行类型转换，通过int()函数将字符串类型的数值转换成数字类型。

为了能够进行加法计算，需要提前拥有两个数值。既然已经存在两个值，那么用户指定数列个数后，在循环时，应该少循环两次。

修改后的代码如下：

```
01.  [ root@py01 bin] # vim fibs2.py
02.
03.  #! /usr/bin/env python
04.
05.  fibs = [ 0, 1]
06.
07.  nums = int( raw_input( 'Input a number: ') )
08.
09.  for i in range( nums - 2) :
10.      fibs.append( fibs[ - 1] + fibs[ - 2] )
11.
12.  print fibs
```

Top

脚本运行结果如下：

```
[root@py01 bin]# ./fibs2.py
Input a number: 5
[0, 1, 1, 2, 3]
[root@py01 bin]# ./fibs2.py
Input a number: 20
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]
```

2 案例2：模拟cp操作

2.1 问题

编写cp.py脚本，实现以下目标：

1. 将/bin/ls“拷贝”到/root/目录下
2. 要求读取/bin/ls后，每次读取4096字节，依次写入到新文件
3. 不要修改原始文件

2.2 方案

“拷贝”一个文件，可以想像成谅是先在目标位置创建一个空文件，然后再将源文件读出，写入到新文件中。

在python中打开文件的方法是open()，或者使用file()，两个函数完全一样可以进行相互替换。open的语法如下：

```
open(name[,mode[,buffering]])
```

open的第一个参数是文件名，可以是相对路径，也可以是绝路径。最后的一个参数是缓冲设置，默认情况下使用缓冲区。硬盘是计算机系统中最慢的一个组件，如果每产生一个字符都立即写入磁盘，那么这种方式的工作效率将极其低下，可以先把数据放在内存的缓冲区中，当缓冲区的数据比较多的时候，再批量写入磁盘文件。当文件关闭的时候，缓冲区的数据也会写入磁盘。第二个模式参数的含义如下：

```
>>> f=open('abc.txt','r')          #文件不存在
Traceback(most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'abc.txt'

>>> f=open('/etc/hosts')
>>> f.write('hello')     #写入文件时报错，因为并非以写入方式打开
Traceback(most recent call last):
  File "<stdin>", line 1, in <module>
IOError: File not open for writing
```

Top

```
>>> f = open('abc.txt', 'w')   #文件不存在，首先创建了abc.txt
>>> f.read()                   #因为以写的方式打开，读取时出错
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: File not open for reading
```

打开文件时并不消耗太多内存，但是将文件内容读到变量中，会根据读入的数据大小消耗相应的内存。例：

```
#首先创建一个名为ttt.img的文件，大小为100MB
[root@py01 bin]# dd if=/dev/zero of=ttt.img bs=1M count=100

[root@py01 bin]# free -m   #观察已用内存，值为665M
             total       used       free     shared    buffers     cached
Mem:           994        665        329          0         24        339
-/+ buffers/cache:        301        693
Swap:         4031          0       4031

[root@py01 bin]# python   #从另一终端中打开python解释器
Python 2.6.6 (r266:84292, Oct 12 2012, 14:23:48)
[GCC 4.4.6 20120305 (Red Hat 4.4.6-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> f = open('ttt.img')   #打开该文件

[root@py01 bin]# free -m  #观察已用内存并没有变化
             total       used       free     shared    buffers     cached
Mem:           994        665        329          0         24        339
-/+ buffers/cache:        301        693
Swap:         4031          0       4031

>>> data = f.read()    #再回到python解释器，读入全部文件内容

[root@py01 bin]# free -m   #观察已用内存，增加了100MB
             total       used       free     shared    buffers     cached
Mem:           994        764        230          0         24        339
-/+ buffers/cache:        399        595
Swap:         4031          0       4031

>>> del data  #回到python解释器，删除data变量
```

Top

```
[root@py01 bin]# free -m  #已用内存又减少了100MB
             total       used       free     shared    buffers     cached
Mem:           994        663        330          0         24        339
-/+ buffers/cache:        299        695
Swap:         4031          0       4031
```

为了防止一次性读入大文件消耗太多的内存，可以采用循环的方式，多次少量读取数据。一般情况下可以设置每次读取4096字节即可。

2.3 步骤

实现此案例需要按照如下步骤进行。

步骤一：编写脚本

注意，通常文件操作有三个步骤：打开文件，处理文件，关闭文件。往往被忽略的是关闭文件，如果文件没有正常关闭，有可能造成数据丢失。

```
[root@py01 bin]# vim cp.py

#! /usr/bin/env python

dstfile = '/root/ls'
srcfile = '/bin/ls'

oldf = open(srcfile)            #以读方式打开老文件
newf = open(dstfile, 'w')       #以写方式打开新文件

while True:                     #因为不确定要循环多少次，设置条件永远为真
    data = oldf.read(4096)      #每次只读入4096字节
    if len(data) == 0:          #如果文件已经全部读取完毕则中断循环
        break
    newf.write(data)

oldf.close()
newf.close()
```

步骤二：通过md5值判断新老文件是否完全一致

```
[root@py01 bin]# ./cp.py
[root@py01 bin]# md5sum /root/ls
c98cff985579da387db6a2367439690a  /root/ls
```

Top

```
[root@py01 bin]# md5sum /bin/ls
c98cff985579da387db6a2367439690a  /bin/ls
[root@py01 bin]# ll /root/ls
-rw-r--r--. 1 root root 117024 Jun 27 15:40 /root/ls
[root@py01 bin]# chmod +x /root/ls
[root@py01 bin]# /root/ls /home/
demo  demo.tar.gz  lost+found
```

md5值是文件的指纹信息，如果两个文件的内容完全一样，那么其md5值一定相同，如果两个文件有微小的不同，其md5值也一定完全不一样。不过文件的权限不会影响md5值的判定。将拷贝后的目标文件加上执行权限，就可以像原始文件一样的工作了。

3 案例3：生成8位随机密码

3.1 问题

编写randpass.py脚本，实现以下目标：

- 使用random的choice函数随机取出字符
- 用于密码的字符通过string模块获得
- 改进程序，用户可以自己决定生成多少位的密码

3.2 方案

假定只使用大小写字母和数字作为密码，这些字符可以自己定义，不过string模块中已有这些属性。通过导入string模块可以减少代码量，同时提高效率。

```
>>> import string

>>> string.lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
```

这个程序的主要思路是，首先创建一个空字符串变量用于存储密码，然后每次在可用字符中随机选择一个字符，再把该字符与变量中的字符进行拼接。

3.3 步骤

Top

实现此案例需要按照如下步骤进行。

步骤一：编写脚本，生成8位随机密码

```
[root@py01 bin]# vim randpass.py

#! /usr/bin/env python

import string
import random

passwd = ''
passchs = string.letters + string.digits

for i in range(8):
    passwd += random.choice(passchs)

print passwd
```

脚本运行结果如下：

```
[root@py01 bin]# ./randpass.py
1U4MMBg3
[root@py01 bin]# ./randpass.py
ExvoT8Hi
```

步骤二：改进脚本，生成指定位数的随机密码

上面程序的主要问题是，第一，生成的密码倍数是固定的，如果希望生成其他倍数的密码，还需要重新改写代码；第二，如果生成密码这个功能在其他地方也需要使用，那么代码的重用性得不到体现。

改进的代码可以解以上两个问题，具体如下：

```
[root@py01 bin]# vim randpass2.py

#! /usr/bin/env python

import string
import random

allchs = string.letters + string.digits

def genPwd(num=8):
```

Top

```
11.         pwd = ''
12.         for i in range(num):
13.             pwd += random.choice(allchs)
14.         return pwd
15.
16.     if __name__ == '__main__':
17.         print genPwd()
18.         print genPwd(6)
```

代码执行结果如下：

```
01.     [root@py01 bin]# ./randpass2.py
02.     hUEDcvmc
03.     RgNMhu
04.     [root@py01 bin]# ./randpass2.py
05.     s9Ojwpp7
06.     70BOsU
07.
08.     [root@py01 bin]# python
09.     Python 2.6.6 (r266:84292, Oct 12 2012, 14:23:48)
10.     [GCC 4.4.6 20120305 (Red Hat 4.4.6-4)] on linux2
11.     Type "help", "copyright", "credits" or "license" for more information.
12.     >>> import randpass2
13.     >>> randpass2.genPwd(10)    #在其他位置也可以直接调用randpass2模中的函数
14.     'WobDgiHsC8'
```

Top