Python 学习笔记（未按知识点顺序排列）

目录

[运行Python程序的方法 2](#_Toc407028611)

[输入和输出 2](#_Toc407028612)

[输出 2](#_Toc407028613)

[输入 2](#_Toc407028614)

[文件的读写 2](#_Toc407028615)

[读文件 2](#_Toc407028616)

[写文件 3](#_Toc407028617)

[一个详细的例子 3](#_Toc407028618)

[随机数random 4](#_Toc407028619)

[Python变量在内存中的变化 4](#_Toc407028620)

[常量 4](#_Toc407028621)

[list[ ]的用法 5](#_Toc407028622)

[tuple( )的用法 5](#_Toc407028623)

[条件判断和循环 6](#_Toc407028624)

[判断： 6](#_Toc407028625)

[循环： 6](#_Toc407028626)

[range()函数 6](#_Toc407028627)

[\_\_name\_\_属性 7](#_Toc407028628)

[装饰器 7](#_Toc407028629)

[图形界面 8](#_Toc407028630)

# 运行Python程序的方法

* 新建一个XXX.py文件，并在里面输入代码。
* 进入命令控制台(cmd)，切换到XXX.py文件所在的目录下.
* 输入：python XXX.py

# 输入和输出

## 输出

* 单个字符串输出：print ‘XXXX’
* 多个字符串输出：print ‘XXX’,’xxx’,’XXX’(每次遇到“,”号都会输出一个空格)

## 输入

* 如果用input()输入的话，输入字符串要加引号
* raw\_input( )函数可以让用户输入（任何输入全部看成字符串）：

name = raw\_input( )

name = raw\_input(‘please enter your name: ‘ )

# 文件的读写

## 读文件

* 一个简单的示例

f=file(‘data.txt’)

data=f.read

print data

f.close()

* 读取文件内容的方法还有
  + readline() #读取一行内容
  + readlines() #把内容按行读取至一个list中

## 写文件

* python默认是以只读模式打开文件。如果想要写入内容，在打开文件的时候需要指定打开模式为写入：

f = file('output.txt', 'w')

* 以这种模式打开文件，原来文件中的内容会被你新写入的内容覆盖掉，如果文件不存在，会自动创建文件。
* 不加参数时，file为你默认为'r'，reading，只读模式，文件必须存在，否则引发异常。
* 另外还有一种模式是'a'，appending。它也是一种写入模式，但你写入的内容不会覆盖之前的内容，而是添加到文件中。
* 打开文件还有一种方法，就是open()，用法和file()是一致的。
* 示例：

data = 'I will be in a file.\n So cool!'

out = file('output.txt', 'w')

out.write(data)

out.close()

## 一个详细的例子

* 文档数据：

刘备 23 35 44 47 51  
关羽 60 77 68  
张飞 97 99 89 91  
诸葛亮 100

* 读文件：

f = file(‘aa.txt’)

* 取得文件的数据，因为每一行都是一条学生成绩的记录，所以用readlines，把每一行分开，便于之后的数据处理：

lines = f.readlines()

* 关闭文件

f.close()

* 对每一条数据进行处理。按照空格，把姓名、每次的成绩分割开：

for line in lines:

data = line.split()

* 对于每一条数据，都新建一个字符串，把学生的名字和算好的总成绩保存进去。最后再把这些字符串一起保存到文件中：

sum = 0  
for score in data[1:]:  
　sum += int(score)  
result = '%s\t: %d\n' % (data[0], sum)

# 随机数random

* 示例：

from random import \*

a = randint(1,100)

# Python变量在内存中的变化

a = 'ABC'

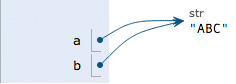
b = a

a = 'XYZ'

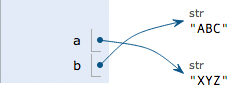
print b

py-var-code-1

执行b = a，解释器创建了变量b，并把b指向a指向的字符串'ABC'：



执行a = 'XYZ'，解释器创建了字符串'XYZ'，并把a的指向改为'XYZ'，但b并没有更改：



所以，最后打印变量b的结果自然是'ABC'了。

# 常量

* 在Python中，通常用全部大写的变量名表示常量

PI=3.14444444

但事实上PI仍然是一个变量，Python根本没有任何机制保证PI不会被改变，所以，用全部大写的变量名表示常量只是一个习惯上的用法，如果你一定要改变变量PI的值，也没人能拦住你。

# list[ ]的用法

* 列表是有序的集合，可以随时添加和删除其中的元素
* list中的元素的数据类型也可以不同
* list中的元素也可以是另一个list
* 生成一个列表：

classmate = [‘a’,’b’,’c’]

* 获取列表元素的个数：

len(classmates)

* 访问列表元素靠索引：

classmates[0]

* 访问列表最后一个元素：

classmates[-1]

* 按倒序访问列表元素：

classmates[-2] 访问倒数第2个元素

* 往list中追加元素到末尾

classmates.append(‘xxx’)

* 往list中追加元素到指定的位置

classmates.insert(1,’XXXX’) classmates[1]=’XXXX’

* 删除list末尾的元素：

classmates.pop()

* 删除指定位置的元素：

classmates.pop(i) //会返回该位置上的值。

# tuple( )的用法

* tuple一旦初始化就不能修改
* 他没有append(),insert()这样的方法
* tuple的一个陷阱：

当你定义一个tuple时，在定义的时候，tuple的元素就必须被确定下来。

* t=(1,2), t=()表示定义了一个空的元组。

但当你定义t=(1)是，会产生歧义，因为他还可以表示数学公式中的小括号。所以为了消除歧义，采用以下方法：

* t=(1,)
* 有一点很重要：

tuple所谓的“不变”是说，tuple的每个元素，指向永远不变。即指向'a'，就不能改成指向'b'，指向一个list，就不能改成指向其他对象，但指向的这个list本身是可变的！

要创建一个内容也不变的tuple怎么做？那就必须保证tuple的每一个元素本身也不能变。

# 条件判断和循环

## 判断：

* elif是else if的缩写，完全可以有多个elif，所以if语句的完整形式就是：

if <条件判断1>:

<执行1>

elif <条件判断2>:

<执行2>

elif <条件判断3>:

<执行3>

else:

<执行4>

* if判断条件还可以简写，比如写：

if x:

print 'True'

* 只要x是非零数值、非空字符串、非空list等，就判断为True，否则为False。

## 循环：

* Python的循环有两种，一种是for...in循环，依次把list或tuple中的每个元素迭代出来，看例子：

names = ['Michael', 'Bob', 'Tracy']

for name in names:

print name

执行这段代码，会依次打印names的每一个元素

### range()函数

如果要计算1-100的整数之和，从1写到100有点困难，幸好Python提供一个range()函数，可以生成一个整数序列，比如range(5)生成的序列是从0开始**小于5**的整数：

# \_\_name\_\_属性

* 由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。

>>> def now():

... print '2013-12-25'

...

>>> f = now

>>> f()

2013-12-25

* 函数对象有一个\_\_name\_\_属性，可以拿到函数的名字：

>>> now.\_\_name\_\_

'now'

>>> f.\_\_name\_\_

'now'

# 装饰器

* 现在，假设我们要增强now()函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改now()函数的定义，这种在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。本质上，decorator就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的decorator，可以定义如下：

def log(func):

def wrapper(\*args, \*\*kw):

print 'call %s():' % func.\_\_name\_\_

return func(\*args, \*\*kw)

return wrapper

* 观察上面的log，因为它是一个decorator，所以接受一个函数作为参数，并返回一个函数。我们要借助Python的@语法，把decorator置于函数的定义处：

@log

def now():

print '2013-12-25'

* 调用now()函数，不仅会运行now()函数本身，还会在运行now()函数前打印一行日志：

>>> now()

call now():

2013-12-25

* 把@log放到now()函数的定义处，相当于执行了语句：

now = log(now)

* 由于log()是一个decorator，返回一个函数，所以，原来的now()函数仍然存在，只是现在同名的now变量指向了新的函数，于是调用now()将执行新函数，即在log()函数中返回的wrapper()函数。
* wrapper()函数的参数定义是(\*args, \*\*kw)，因此，wrapper()函数可以接受任意参数的调用。在wrapper()函数内，首先打印日志，再紧接着调用原始函数。
* 一个完整的decorator的写法如下：

import functools

def log(func):

@functools.wraps(func)

def wrapper(\*args, \*\*kw):

print 'call %s():' % func.\_\_name\_\_

return func(\*args, \*\*kw)

return wrapper

* 或者针对带参数的decorator：

import functools

def log(text):

def decorator(func):

@functools.wraps(func)

def wrapper(\*args, \*\*kw):

print '%s %s():' % (text, func.\_\_name\_\_)

return func(\*args, \*\*kw)

return wrapper

return decorator

# 图形界面

# 使用模块

* 一个例子：

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

* + 第1行和第2行是标准注释，第1行注释可以让这个hello.py文件直接在Unix/Linux/Mac上运行，第2行注释表示.py文件本身使用标准UTF-8编码；

' a test module '

* + 第3行是一个字符串，表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释；

\_\_author\_\_ = 'Michael Liao'

* + 第4行使用\_\_author\_\_变量把作者写进去，这样当你公开源代码后别人就可以瞻仰你的大名；

import sys

* + 使用sys模块的第一步，就是导入该模块：
  + sys模块有一个argv变量，用list存储了命令行的所有参数。argv至少有一个元素，因为第一个参数永远是该.py文件的名称，例如：

运行python hello.py获得的sys.argv就是['hello.py']；

运行python hello.py Michael获得的sys.argv就是['hello.py', 'Michael]。

def test():

args = sys.argv

if len(args)==1:

print 'Hello, world!'

elif len(args)==2:

print 'Hello, %s!' % args[1]

else:

print 'Too many arguments!'

if \_\_name\_\_=='\_\_main\_\_':

test()

* 当我们在命令行运行hello模块文件时，Python解释器把一个特殊变量\_\_name\_\_置为\_\_main\_\_，而如果在其他地方导入该hello模块时，if判断将失败，因此，这种if测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是运行测试。
* 如果启动Python交互环境，再导入hello模块：

$ python

Python 2.7.5 (default, Aug 25 2013, 00:04:04)

[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

>>> import hello

>>>

导入时，没有打印Hello, word!，因为没有执行test()函数。

调用hello.test()时，才能打印出Hello, word!：

>>> hello.test()

Hello, world!

# 别名

* 导入模块时，还可以使用别名，这样，可以在运行时根据当前环境选择最合适的模块。比如Python标准库一般会提供StringIO和cStringIO两个库，这两个库的接口和功能是一样的，但是cStringIO是C写的，速度更快，所以，你会经常看到这样的写法：

try:

import cStringIO as StringIO

except ImportError: # 导入失败会捕获到ImportError

import StringIO

这样就可以优先导入cStringIO。如果有些平台不提供cStringIO，还可以降级使用StringIO。导入cStringIO时，用import ... as ...指定了别名StringIO，因此，后续代码引用StringIO即可正常工作。

还有类似simplejson这样的库，在Python 2.6之前是独立的第三方库，从2.6开始内置，所以，会有这样的写法：

try:

import json # python >= 2.6

except ImportError:

import simplejson as json # python <= 2.5

由于Python是动态语言，函数签名一致接口就一样，因此，无论导入哪个模块后续代码都能正常工作。

# 作用域

* 在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在Python中，是通过\_前缀来实现的。
* 正常的函数和变量名是公开的（public），可以被直接引用，比如：abc，x123，PI等；

类似\_\_xxx\_\_这样的变量是特殊变量，可以被直接引用，但是有特殊用途，比如上面的\_\_author\_\_，\_\_name\_\_就是特殊变量，hello模块定义的文档注释也可以用特殊变量\_\_doc\_\_访问，我们自己的变量一般不要用这种变量名；

类似\_xxx和\_\_xxx这样的函数或变量就是非公开的（private），不应该被直接引用，比如\_abc，\_\_abc等；

之所以我们说，private函数和变量“不应该”被直接引用，而不是“不能”被直接引用，是因为Python并没有一种方法可以完全限制访问private函数或变量，但是，从编程习惯上不应该引用private函数或变量。

private函数或变量不应该被别人引用，那它们有什么用呢？请看例子：

def \_private\_1(name):

return 'Hello, %s' % name

def \_private\_2(name):

return 'Hi, %s' % name

def greeting(name):

if len(name) > 3:

return \_private\_1(name)

else:

return \_private\_2(name)

* 我们在模块里公开greeting()函数，而把内部逻辑用private函数隐藏起来了，这样，调用greeting()函数不用关心内部的private函数细节，这也是一种非常有用的代码封装和抽象的方法，即：

外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义public。

# Glob全局变量

* global---将变量定义为全局变量。可以通过定义为全局变量，实现在函数内部改变变量值。
* 一个global语句可以同时定义多个变量，如 global x, y, z
* 示例程序：

>>> def func():

... global x

... print 'x is ', x

... x = 2

... print 'Change local x to ', x

...

>>> x = 50

>>> func()

x is 50

Change local x to 2