我的第一个Python程序

2016年8月12日

23:27

print ("Am I a cute boy? Answer yes or no.")

answer = input()

while answer == "no":

print ("Your answer was wrong. Answer again, please.")

answer = input()

if answer == "yes":

print ("You are smart, boy.")

Saturday, October 22, 2016

09:46

[**Python 的练手项目有哪些值得推荐？**](https://www.zhihu.com/question/29372574)

[小小搬运工](https://www.zhihu.com/people/zhang-peng-cheng-67-42) ....

[4453 人赞同](https://www.zhihu.com/topic/19552826/top-answers?page=2)

更新：兄弟们如此给力，小弟受宠若惊，好东西就是用来分享的。 500 line or less只是某个系列丛书中第四本，其他三本链接：

[http://www.aosabook.org/en/index.html](https://link.zhihu.com/?target=http%3A//www.aosabook.org/en/index.html)

可以在线看哦

前两本每章介绍出名开源软件的架构，你说的上名字的开源基本上都能找到，短小精悍的介绍框架

第三本介绍开源软件中实际遇到的一些performance的问题以及解决方案，同样浓缩啊！！ 每章的作者大多就是此开软软件的开发者啊！！！

不用多说，点开你就知道啊

不想看英文的怒点 [http://www.ituring.com.cn/article/13057](https://link.zhihu.com/?target=http%3A//www.ituring.com.cn/article/13057)

前两卷的中文版，不知道翻译的怎么样

----------------------------------------割了你--------------------------------------------------------------

和楼主一样在学习python， 最近找到一本python神书 500 line or less. 有兄弟提到了git的源码，这里不只是源码，而是这么一本书，目前还没出版，但是在网上有已经看到了 review版本。

强烈推荐！！！

强烈推荐！！！

强力推荐！！！

这本书共16个章节，每章均是由该领域的大牛完成，用不到500行的代码实现一个特（装）定（逼）功能。

本书链接 [http://aosabook.org/blog/](https://link.zhihu.com/?target=http%3A//aosabook.org/blog/)

目录页：[http://aosabook.org/blog/](https://link.zhihu.com/?target=http%3A//aosabook.org/blog/)

以下是章节目录，每一章都让你热血澎湃，看完介绍你就向往下读。

1. A Template Engine （[http://aosabook.org/en/500L/a-template-engine.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/a-template-engine.html)）

MVC模型中的view层如何解析html中的静态变量和简单的语句，如下：

<p>Welcome, {name}!</p>  
<p>Products:</p>  
<ul>  
{products}  
</ul>

web中的view层不只是html代码，还有支持其他的代码。比如 {products}是一个变量。 同时view层还支持{if} , {for}, {foreach}等等。django，velocity等是如何解析他们的？

大牛用不到500行代码告诉你，是如何实现的？ （不是替换，替换需要每次请求都需要解析）

2. Web Spreadsheet （[http://aosabook.org/en/500L/web-spreadsheet.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/web-spreadsheet.html)）

web的电子表格如何实现的？ 好像比较简单，但是介绍了 web storage 和 web worker，还是很值得一看的

3. A Web Crawler [http://aosabook.org/en/500L/a-web-crawler-with-asyncio-coroutines.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/a-web-crawler-with-asyncio-coroutines.html)

不多说，几百行代码实现高效的网络爬虫， 高效！

4. Static Analysis [http://aosabook.org/en/500L/static-analysis.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/static-analysis.html)

成熟的IDE都有代码检查和代码提示，怎么做的？ 看这章

5. Clustering by Consensus [http://aosabook.org/en/500L/clustering-by-consensus.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/clustering-by-consensus.html)

分布式系统 paxos原理与实现。不知道paxos说明你没接触过分布式体统，接触过分布式还不懂，说明你只会用分布式系统

6. A Simple Object Modle [http://aosabook.org/en/500L/a-simple-object-model.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/a-simple-object-model.html)

Python是面向对象语言，对象，继承，多态，怎么用代码实现的，不到500行代码，实际不到400 行， 666.。。

7. An Archaeology-Inspired Database [http://aosabook.org/en/500L/an-archaeology-inspired-database.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/an-archaeology-inspired-database.html)

如何用python实现一个数据库，支持 query，index, transaction， 2，3百行代码和对每个函数的讲解。看完你就知道知道数据库原理，太值了

8. Dog Bed Database [http://aosabook.org/en/500L/dbdb-dog-bed-database.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/dbdb-dog-bed-database.html)

类似上一章，不过这次实现的是key-value的非关系型数据库，详细的讲解和2，3百行代码

9. A 3D Modeller [http://aosabook.org/en/500L/a-3d-modeller.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/a-3d-modeller.html)

用python实现一个3D设计，显示到屏幕，可以交互。不是很懂，但不明觉厉

10. A Python Interpreter Written in Python [http://aosabook.org/en/500L/a-python-interpreter-written-in-python.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/a-python-interpreter-written-in-python.html)

手把手教你如何实现python解析器。

11. A Pedometer in the Real World [http://aosabook.org/en/500L/a-pedometer-in-the-real-world.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/a-pedometer-in-the-real-world.html)

你用过手机应用记录你每天走的步数，然后发送到朋友圈吗？ （没有？ 没关系。）这章告诉你如何实现步数记录，怎么算走一步。手机中有加速记，很容易获得你某一时刻在x,y,z三个方向的加速度，用这些参数，如何计算你走了 多少步？ 知道吗？ 不知道，看这章，讲解加实现

12. A Continuous Intergration System [http://aosabook.org/en/500L/a-continuous-integration-system.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/a-continuous-integration-system.html)

CI System是一个专门用来测试新代码的系统，根据代码提交记录，拿到新的代码，测试，生成报告。这不是关键，关键是 如果test失败，它还会 恢复，然后从失败的那个点在跑，相当于把出错环境重现了。。。

13 A Rejection Sampler [http://aosabook.org/en/500L/a-rejection-sampler.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/a-rejection-sampler.html)

不是很懂，和机器学习相关，如何 计算你赢得象棋比赛的概率，天气对飞机的影响等类似的问题

14 A visual programming toolkit [http://aosabook.org/en/500L/blockcode-a-visual-programming-toolkit.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/blockcode-a-visual-programming-toolkit.html)

不太明白

15. A Flow Shop Scheduler [http://aosabook.org/en/500L/a-flow-shop-scheduler.html](https://link.zhihu.com/?target=http%3A//aosabook.org/en/500L/a-flow-shop-scheduler.html)

flowshop调度问题，好像很出名的样子，最优化问题，如何从局部最优解找全局最优解

16 Optical Character Recognition

几百行代码使用人工神经网络实现识别手写字母。。。

github源码：[500lines/README.md at master · aosabook/500lines · GitHub](https://link.zhihu.com/?target=https%3A//github.com/aosabook/500lines/blob/master/README.md)

[编辑于 2016-03-01](https://www.zhihu.com/question/29372574/answer/88624507)

来自 <[*https://www.zhihu.com/topic/19552826/top-answers?page=2*](https://www.zhihu.com/topic/19552826/top-answers?page=2)>

书籍

2016年1月4日

11:28

**Python学习相关笔记** 来自 <<http://www.cnblogs.com/feeland/category/667460.html>>

MSDN和google的开发者中心，hack news，伯乐在线

**SICP 计算机程序的构造和解释**来自 <<https://book.douban.com/subject/1148282/>>

**编译原理** 来自 <<https://book.douban.com/subject/3296317/>>

**计算机组成：结构化方法**来自 <<https://book.douban.com/subject/1886058/>>

**深入理解计算机系统**来自 <<https://book.douban.com/subject/1230413/>>

计算机组成和编译原理、操作系统、计算机体系结构关系密切，假如想融汇贯通，必须全学完，回头看才能明白

[**http://www.runoob.com/w3cnote/google-python-styleguide.html**](http://www.runoob.com/w3cnote/google-python-styleguide.html)

**Python 简介**来自 <http://www.runoob.com/python3/python3-tutorial.html>

<<http://www.runoob.com/python/python-intro.html>>

**简明 Python 教程**来自 <<http://old.sebug.net/paper/python/index.html>>

**Python 入门教程**来自 <<http://docspy3zh.readthedocs.io/en/latest/tutorial/index.html>>

[**http://crossincode.com/home/**](http://crossincode.com/home/) **Yensan mylove2006**

[**http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000**](http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000)

[**http://www.pythonpub.com/**](http://www.pythonpub.com/)

[**http://www.pythondoc.com/pythontutorial3/**](http://www.pythondoc.com/pythontutorial3/)

[**https://docs.python.org/3/**](https://docs.python.org/3/)

入门 <http://www.imooc.com/learn/177>

进阶 <http://www.imooc.com/view/317>

<http://www.imooc.com/search/course?words=python>

**算法导论**

**“笨办法”学Python（第3版）**

Learn Python the Hard Way: A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code (3rd Edition)

来自 <[*http://book.douban.com/subject/26264642/*](http://book.douban.com/subject/26264642/)>

**Python核心编程（第二版）**

Core Python Programming, 2nd Edition

来自 <[*http://book.douban.com/subject/3112503/*](http://book.douban.com/subject/3112503/)>

**Python Cookbook**

**Python 学习手册 >>Python编程(第4版)(Programming.Python**

**python高分书单**

来自 <[*http://www.douban.com/doulist/42824743/*](http://www.douban.com/doulist/42824743/)>

Python学习技能树

2017年1月15日 星期日

上午11:20

* 1. **入门**
  2. **Python简介、环境搭建与配置、开发工具与配置**

* 1. **语法基础**

变量与数据类型

全局变量、成员变量、局部变量

数据类型

字符串str、整数型int、数字number、长整数型long、浮点数float

列表list、元组tuple、字典dict

运算

赋值：变量赋值，多变量赋值

运算符：算术运算、赋值运算、比较运算、身份运算、成员运算、逻辑运算

流程控制

赋值语句：链式赋值语句、增量赋值语句

缩进：

条件选择：if…elif…else

循环控制：while iter（迭代）

跳转语句：continue、break、return

断言

pass、del、exec

函数

基本语法、函数调用

参数：必备参数、命名参数、缺省参数、不定长参数

Return语句

异常处理

try

except

else

finally

with …

输入输出文件处理

file对象属性

open函数

read()

readlines()

write()

writelines()

close()

* 1. **简易爬虫实战**

Urllib 和 urllib2

异同点，简单实用，爬虫实例

正则表达式re

编写爬虫代码

多线程

* 1. **进阶**
  + **装饰器**
  + **面向对象**
    - 什么是面向对象
    - 继承、多态、封装
    - 类
      * 类的属性：公有属性、私有属性、受保护的属性
      * 类方法和静态方法
      * 继承和方法重载
      * 类的特殊方法
        + 特殊变量：\_\_doc\_\_ /\_\_name\_\_/\_\_model\_\_/\_\_bases\_\_/\_\_dict\_\_/\_\_class\_\_
        + 常见的魔术方法：构造函数 \_\_init\_\_()/析构函数\_\_del\_\_()/\_\_new\_\_()
        + 用于比较的，魔术方法：\_\_eq\_\_/\_\_ne\_\_/\_\_lt\_\_/\_\_gt\_\_/\_\_cmp\_\_
        + 一元操作符：\_\_pos\_\_/\_\_neg\_\_/\_\_abs\_\_/\_\_invert\_\_
        + 普通算数操作符的魔术方法
        + 控制属性访问的魔术方法
      * 访问权限
      * Super 与 self
      * 嵌套类和嵌套函数
      * 常见的工具方法
      * 元类和元编程

* + **MySQL**

SQL规范和创建

* + 安装与配置：server版，安装版
  + SQL语句规范
  + 数据类型
  + 存储引擎类型
  + 创建和修改表

约束：主键外键、unique、check、默认值

增删改查：

基本查询：表达式查询、按条件查询、范围查询、模糊查询、分组查询

排序：ASC、DESC

Limit 限制条件查询

联合查询

子查询

运算符

常用函数

* + **网络爬虫**

Scrapy 框架

创建工程

定义抓取条目

编写爬虫

提取字段

存储提取数据

* + **Web框架**
  + **Web.py基础**

Web.py介绍

安装生产环境

基本应用：静态文件访问、URL控制、跳转与重定向

高级应用：获取客服端信息、添加钩子/卸载钩子、404定义、传输大文件

Cookie&Session：cookies操作，用户认证，子应用中操作Session

templates模板

* + **Django基础**

django介绍

创建网站

模板

SQLite3数据库简介

数据库基本操作：数据基本操作、数据关联操作、数据聚合操作

admin使用

* + **Tornado基础**
    - Tornado介绍
    - 简单的Web服务
    - 表单和模板
    - 数据库操作
    - 异步Web请求
    - 编写安全应用：cookies漏洞、请求漏洞、用户验证
    - 部署Tornado：运行多个Tornado实例的原因、使用Nginx作为方向代理、监控Tornado进程

* + **Flask基础**
    - Flask介绍
    - Web服务环境搭建
    - 模板
    - 表单：配置、表单模板及视图
    - 数据库
    - 分页
    - 全局搜索

* + **项目实战**

Scrapy整站爬虫

微信系统（微信在线点餐系统）

个人博客系统（Django）

企业OA系统（Tornado）

网盘系统（Web.py）

Thursday, August 18, 2016

20:56

[**Python:eval的妙用和滥用**](http://blog.csdn.net/zhanh1218/article/details/37562167)

eval()函数十分强大，官方demo解释为：将字符串str当成有效的表达式来求值并返回计算结果。

so，结合math当成一个计算器很好用。

其他用法，可以把list,tuple,dict和string相互转化。见下例子：

**[python]** [view plain copy](http://blog.csdn.net/zhanh1218/article/details/37562167?utm_source=tuicool&utm_medium=referral)

[在CODE上查看代码片](https://code.csdn.net/snippets/421999)

* 1. a = "[[1,2], [3,4], [5,6], [7,8], [9,0]]"
  3. b = eval(a)
  5. b
  6. Out[3]: [[1, 2], [3, 4], [5, 6], [7, 8], [9, 0]]
  8. type(b)
  9. Out[4]: list
  11. a = "{1: 'a', 2: 'b'}"
  13. b = eval(a)
  15. b
  16. Out[7]: {1: 'a', 2: 'b'}
  18. type(b)
  19. Out[8]: dict
  21. a = "([1,2], [3,4], [5,6], [7,8], (9,0))"
  23. b = eval(a)
  25. b
  26. Out[11]: ([1, 2], [3, 4], [5, 6], [7, 8], (9, 0))

不可谓不强大！

ＢＵＴ！强大的函数有代价。安全性是其最大的缺点。

想一想这种使用环境：需要用户输入一个表达式，并求值。

如果用户恶意输入，例如：

**[plain]** [view plain copy](http://blog.csdn.net/zhanh1218/article/details/37562167?utm_source=tuicool&utm_medium=referral)

[在CODE上查看代码片](https://code.csdn.net/snippets/421999)

* 1. \_\_import\_\_('os').system('dir')

那么eval()之后，你会发现，当前目录文件都会展现在用户前面。

那么继续输入：

**[plain]** [view plain copy](http://blog.csdn.net/zhanh1218/article/details/37562167?utm_source=tuicool&utm_medium=referral)

* 1. open('文件名').read()

代码都给人看了。获取完毕，一条删除命令，文件消失。哭吧！

怎么避免安全问题？

１、自行写检查函数；

２、使用ast.literal\_eval：[自行查看DOCUMENT](https://docs.python.org/2/library/ast.html)

3、更多好文：[Restricted "safe" eval(Python recipe)](http://code.activestate.com/recipes/496746-restricted-safe-/)

本文由@The\_Third\_Wave（Blog地址：<http://blog.csdn.net/zhanh1218>）原创。还有未涉及的，会不定期更新，有错误请指正。

如果你看到这篇博文时发现不完整，那是我为防止爬虫先发布一半的原因，请看原作者Blog。

如果这篇博文对您有帮助，为了好的网络环境，不建议转载，建议收藏！如果您一定要转载，请带上后缀和本文地址。

来自 <[*http://blog.csdn.net/zhanh1218/article/details/37562167?utm\_source=tuicool&utm\_medium=referral*](http://blog.csdn.net/zhanh1218/article/details/37562167?utm_source=tuicool&utm_medium=referral)>

Py2 与 Py3的主要差异

2016年8月6日

1:40

因这学期负责Python课程的助教，刚开始上机试验的几节课，有很多同学用 Python3.4 的编译器编译 Python 2.7 的程序而导致不通过。Python 2.7.x 和 Python 3.x 版本并非完全兼容。

许多 Python 初学者想知道他们应该从 Python 的哪个版本开始学习。对于这个问题我的答案是 “你学习你喜欢的教程的版本，然后检查他们之间的不同。” 但如果你并未了解过两个版本之间的差异，个人推荐使用 Python 2.7.x 版本，毕竟大部分教材等资料还是用Python 2.7.x来写的。

但是如果你开始一个新项目，并且有选择权？我想说的是目前没有对错，只要 你计划使用的库 Python 2.7.x 和 Python 3.x 双方都支持的话。尽管如此，当在编写它们中的任何一个的代码，或者是你计划移植你的项目的时候，是非常值得看看这两个主要流行的 Python 版本之间的差别的，以便避免常见的陷阱。

本文翻译自：[《Key differences between Python 2.7.x and Python 3.x》](http://nbviewer.ipython.org/github/rasbt/python_reference/blob/master/tutorials/key_differences_between_python_2_and_3.ipynb?create=1)

**\_\_future\_\_模块**

Python 3.x 介绍的 一些Python 2 不兼容的关键字和特性可以通过在 Python 2 的内置 \_\_future\_\_ 模块导入。如果你计划让你的代码支持 Python 3.x，建议你使用 \_\_future\_\_ 模块导入。例如，如果我想要 在Python 2 中表现 Python 3.x 中的整除，我们可以通过如下导入

|  |  |
| --- | --- |
| 1 | from \_\_future\_\_ import division |

更多的 \_\_future\_\_ 模块可被导入的特性被列在下表中：

|  |  |  |  |
| --- | --- | --- | --- |
| **feature** | **optional in** | **mandatory in** | **effect** |
| nested\_scopes | 2.1.0b1 | 2.2 | [PEP 227](http://legacy.python.org/dev/peps/pep-0227/): Statically Nested Scopes |
| generators | 2.2.0a1 | 2.3 | [PEP 255](http://legacy.python.org/dev/peps/pep-0255/): Simple Generators |
| division | 2.2.0a2 | 3.0 | [PEP 238](http://legacy.python.org/dev/peps/pep-0238/): Changing the Division Operator |
| absolute\_import | 2.5.0a1 | 3.0 | [PEP 328](http://legacy.python.org/dev/peps/pep-0328/): Imports: Multi-Line and Absolute/Relative |
| with\_statement | 2.5.0a1 | 2.6 | [PEP 343](http://www.python.org/dev/peps/pep-0343): The “with” Statement |
| print\_function | 2.5.0a2 | 3.0 | [PEP 3105](http://legacy.python.org/dev/peps/pep-3105/): Make print a function |
| unicode\_literals | 2.5.0a2 | 3.0 | [PEP 3112](http://legacy.python.org/dev/peps/pep-3112/): Bytes literals in Python 3000 |

(Source: <https://docs.python.org/2/library/future.html>)

|  |  |
| --- | --- |
| 1 | from platform import python\_version |

**print函数**

很琐碎，而 print 语法的变化可能是最广为人知的了，但是仍值得一提的是： Python 2 的 print 声明已经被 print() 函数取代了，这意味着我们必须包装我们想打印在小括号中的对象。

Python 2 不具有额外的小括号问题。但对比一下，如果我们按照 Python 2 的方式不使用小括号调用 print 函数，Python 3 将抛出一个语法异常（SyntaxError）。

***Python 2***

|  |  |
| --- | --- |
| 1  2  3  4 | print 'Python', python\_version()  print 'Hello, World!'  print('Hello, World!')  print "text", ; print 'print more text on the same line' |

*run result:*

Python 2.7.6

Hello, World!

Hello, World!

text print more text on the same line

***Python 3***

|  |  |
| --- | --- |
| 1  2  3  4 | print('Python', python\_version())  print('Hello, World!')  print("some text,", end="")  print(' print more text on the same line') |

*run result:*

Python 3.4.1

Hello, World!

some text, print more text on the same line

|  |  |
| --- | --- |
| 1 | print 'Hello, World!' |

*run result:*

File ““, line 1

print ‘Hello, World!’

^

SyntaxError: invalid syntax

**Note:**

以上通过 Python 2 使用 Printing "Hello, World" 是非常正常的，尽管如此，如果你有多个对象在小括号中，我们将创建一个元组，因为 print 在 Python 2 中是一个声明，而不是一个函数调用。

|  |  |
| --- | --- |
| 1  2  3 | print 'Python', python\_version()  print('a', 'b')  print 'a', 'b' |

*run result:*

Python 2.7.7

(‘a’, ‘b’)

a b

**整除**

如果你正在移植代码，这个变化是特别危险的。或者你在 Python 2 上执行 Python 3 的代码。因为这个整除的变化表现在它会被忽视（即它不会抛出语法异常）。

因此，我还是倾向于使用一个 float(3)/2 或 3/2.0 代替在我的 Python 3 脚本保存在 Python 2 中的 3/2 的一些麻烦（并且反而过来也一样，我建议在你的 Python 2 脚本中使用 from \_\_future\_\_ import division）

***Python 2***

|  |  |
| --- | --- |
| 1  2  3  4  5 | print 'Python', python\_version()  print '3 / 2 =', 3 / 2  print '3 // 2 =', 3 // 2  print '3 / 2.0 =', 3 / 2.0  print '3 // 2.0 =', 3 // 2.0 |

*run result:*

Python 2.7.6

3 / 2 = 1

3 // 2 = 1

3 / 2.0 = 1.5

3 // 2.0 = 1.0

***Python 3***

|  |  |
| --- | --- |
| 1  2  3  4  5 | print('Python', python\_version())  print('3 / 2 =', 3 / 2)  print('3 // 2 =', 3 // 2)  print('3 / 2.0 =', 3 / 2.0)  print('3 // 2.0 =', 3 // 2.0) |

*run result:*

Python 3.4.1

3 / 2 = 1.5

3 // 2 = 1

3 / 2.0 = 1.5

3 // 2.0 = 1.0

**Unicode**

Python 2 有 ASCII str() 类型，unicode() 是单独的，不是 byte 类型。

现在， 在 Python 3，我们最终有了 Unicode (utf-8) 字符串，以及一个字节类：byte 和 bytearrays。

***Python 2***

|  |  |
| --- | --- |
| 1 | print 'Python', python\_version() |

*run result:*

Python 2.7.6

|  |  |
| --- | --- |
| 1 | print type(unicode('this is like a python3 str type')) |

*run result:*

< type ‘unicode’ >

|  |  |
| --- | --- |
| 1 | print type(b'byte type does not exist') |

*run result:*

< type ‘str’ >

|  |  |
| --- | --- |
| 1 | print 'they are really' + b' the same' |

*run result:*

they are really the same

|  |  |
| --- | --- |
| 1 | print type(bytearray(b'bytearray oddly does exist though')) |

*run result:*

< type ‘bytearray’ >

***Python 3***

|  |  |
| --- | --- |
| 1  2 | print('Python', python\_version())  print('strings are now utf-8 \u03BCnico\u0394é!') |

*run result:*

Python 3.4.1

strings are now utf-8 μnicoΔé!

|  |  |
| --- | --- |
| 1  2 | print('Python', python\_version(), end="")  print(' has', type(b' bytes for storing data')) |

*run result:*

Python 3.4.1 has < class ‘bytes’ >

|  |  |
| --- | --- |
| 1  2 | print('and Python', python\_version(), end="")  print(' also has', type(bytearray(b'bytearrays'))) |

*run result:*

and Python 3.4.1 also has < class ‘bytearray’>

|  |  |
| --- | --- |
| 1 | 'note that we cannot add a string' + b'bytes for data' |

*run result:*

-—————————————————————————————————————

TypeError Traceback (most recent call last)

< ipython-input-13-d3e8942ccf81> in < module>()

——> 1 ‘note that we cannot add a string’ + b’bytes for data’

TypeError: Can’t convert ‘bytes’ object to str implicitly

**xrange模块**

在 Python 2 中 xrange() 创建迭代对象的用法是非常流行的。比如： for 循环或者是列表/集合/字典推导式。

这个表现十分像生成器（比如。“惰性求值”）。但是这个 xrange-iterable 是无穷的，意味着你可以无限遍历。

由于它的惰性求值，如果你不得仅仅不遍历它一次，xrange() 函数 比 range() 更快（比如 for 循环）。尽管如此，对比迭代一次，不建议你重复迭代多次，因为生成器每次都从头开始。

在 Python 3 中，range() 是像 xrange() 那样实现以至于一个专门的 xrange() 函数都不再存在（在 Python 3 中 xrange() 会抛出命名异常）。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | import timeit  n = 10000  def test\_range(n):  return for i in range(n):  pass  def test\_xrange(n):  for i in xrange(n):  pass |

***Python 2***

|  |  |
| --- | --- |
| 1  2  3  4  5 | print 'Python', python\_version()  print '\ntiming range()'  %timeit test\_range(n)  print '\n\ntiming xrange()'  %timeit test\_xrange(n) |

*run result:*

Python 2.7.6

timing range()

1000 loops, best of 3: 433 µs per loop

timing xrange()

1000 loops, best of 3: 350 µs per loop

***Python 3***

|  |  |
| --- | --- |
| 1  2  3 | print('Python', python\_version())  print('\ntiming range()')  %timeit test\_range(n) |

*run result:*

Python 3.4.1

timing range()

1000 loops, best of 3: 520 µs per loop

|  |  |
| --- | --- |
| 1 | print(xrange(10)) |

*run result:*

-—————————————————————————————————————

NameError Traceback (most recent call last)

in ()

——> 1 print(xrange(10))

NameError: name ‘xrange’ is not defined

**Python3中的range对象的\_\_contains\_\_方法**

另外一件值得一提的事情就是在 Python 3 中 range 有一个新的 \_\_contains\_\_ 方法（感谢 Yuchen Ying 指出了这个），\_\_contains\_\_ 方法可以加速 “查找” 在 Python 3.x 中显著的整数和布尔类型。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | x = 10000000  def val\_in\_range(x, val):  return val in range(x)  def val\_in\_xrange(x, val):  return val in xrange(x)  print('Python', python\_version())  assert(val\_in\_range(x, x/2) == True)  assert(val\_in\_range(x, x//2) == True)  %timeit val\_in\_range(x, x/2)  %timeit val\_in\_range(x, x//2) |

*run result:*

Python 3.4.1

1 loops, best of 3: 742 ms per loop

1000000 loops, best of 3: 1.19 µs per loop

基于以上的 timeit 的结果，当它使一个整数类型，而不是浮点类型的时候，你可以看到执行查找的速度是 60000 倍快。尽管如此，因为 Python 2.x 的 range 或者是 xrange 没有一个 \_\_contains\_\_ 方法，这个整数类型或者是浮点类型的查询速度不会相差太大。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | print 'Python', python\_version()  assert(val\_in\_xrange(x, x/2.0) == True)  assert(val\_in\_xrange(x, x/2) == True)  assert(val\_in\_range(x, x/2) == True)  assert(val\_in\_range(x, x//2) == True)  %timeit val\_in\_xrange(x, x/2.0)  %timeit val\_in\_xrange(x, x/2)  %timeit val\_in\_range(x, x/2.0)  %timeit val\_in\_range(x, x/2) |

*run result:*

Python 2.7.7

1 loops, best of 3: 285 ms per loop

1 loops, best of 3: 179 ms per loop

1 loops, best of 3: 658 ms per loop

1 loops, best of 3: 556 ms per loop

下面说下 \_\_contain\_\_方法并没有加入到 Python 2.x 中的证据：

|  |  |
| --- | --- |
| 1  2 | print('Python', python\_version())  range.\_\_contains\_\_ |

*run result:*

Python 3.4.1

< slot wrapper ‘**contains**‘ of ‘range’ objects >

|  |  |
| --- | --- |
| 1  2 | print 'Python', python\_version()  range.\_\_contains\_\_ |

*run result:*

Python 2.7.7

-—————————————————————————————————————

AttributeError Traceback (most recent call last)

< ipython-input-7-05327350dafb> in < module>()

1 print ‘Python’, python*version()*

*——> 2 range.`\_contains*`

AttributeError: ‘builtin*functionor\_method’ object has no attribute `’\_\_contains*‘`

|  |  |
| --- | --- |
| 1  2 | print 'Python', python\_version()  xrange.\_\_contains\_\_ |

*run result:*

Python 2.7.7

-—————————————————————————————————————

AttributeError Traceback (most recent call last)

< ipython-input-8-7d1a71bfee8e> in < module>()

1 print ‘Python’, python*version()*

*——> 2 xrange.`\_contains*`

AttributeError: type object ‘xrange’ has no attribute '\_\_contains\_\_'

**注意在 Python 2 和 Python 3 中速度的不同**

有些人指出了 Python 3 的 range() 和 Python 2 的 xrange() 之间的速度不同。因为他们是用相同的方法实现的，因此期望相同的速度。尽管如此，这事实在于 Python 3 倾向于比 Python 2 运行的慢一点。

|  |  |
| --- | --- |
| 1  2  3  4  5 | def test\_while():  i = 0  while i < 20000:  i += 1  return |

***Python 3***

|  |  |
| --- | --- |
| 1  2 | print('Python', python\_version())  %timeit test\_while() |

*run result:*

Python 3.4.1

100 loops, best of 3: 2.68 ms per loop

***Python 2***

|  |  |
| --- | --- |
| 1  2 | print 'Python', python\_version()  %timeit test\_while() |

*run result:*

Python 2.7.6

1000 loops, best of 3: 1.72 ms per loop

**Raising exceptions**

Python 2 接受新旧两种语法标记，在 Python 3 中如果我不用小括号把异常参数括起来就会阻塞（并且反过来引发一个语法异常）。

***Python 2***

|  |  |
| --- | --- |
| 1 | print 'Python', python\_version() |

*run result:*

Python 2.7.6

|  |  |
| --- | --- |
| 1 | raise IOError, "file error" |

*run result:*

-—————————————————————————————————————

IOError Traceback (most recent call last)

< ipython-input-8-25f049caebb0> in < module>()

——> 1 raise IOError, “file error”

IOError: file error

|  |  |
| --- | --- |
| 1 | raise IOError("file error") |

*run result:*

-—————————————————————————————————————

IOError Traceback (most recent call last)

< ipython-input-9-6f1c43f525b2> in < module>()

——> 1 raise IOError(“file error”)

IOError: file error

***Python 3***

|  |  |
| --- | --- |
| 1 | print 'Python', python\_version() |

*run result:*

Python 3.4.1

|  |  |
| --- | --- |
| 1 | raise IOError, "file error" |

*run result:*

File ““, line 1

raise IOError, “file error”

^

SyntaxError: invalid syntax

在 Python 3 中，可以这样抛出异常：

|  |  |
| --- | --- |
| 1  2 | print('Python', python\_version())  raise IOError("file error") |

*run result:*

Python 3.4.1

-—————————————————————————————————————

OSError Traceback (most recent call last)

< ipython-input-11-c350544d15da> in < module>()

1 print(‘Python’, python\_version())

——> 2 raise IOError(“file error”)

OSError: file error

**Handling exceptions**

在 Python 3 中处理异常也轻微的改变了，在 Python 3 中我们现在使用 as 作为关键词。

***Python 2***

|  |  |
| --- | --- |
| 1  2  3  4  5 | print 'Python', python\_version()  try:  let\_us\_cause\_a\_NameError  except NameError, err:  print err, '--> our error message' |

*run result:*

Python 2.7.6

name ‘let\_us\_cause\_a\_NameError’ is not defined —> our error message

***Python 3***

|  |  |
| --- | --- |
| 1  2  3  4  5 | print('Python', python\_version())  try:  let\_us\_cause\_a\_NameError  except NameError as err:  print(err, '--> our error message') |

*run result:*

Python 3.4.1

name ‘let\_us\_cause\_a\_NameError’ is not defined —> our error message

**next()函数 and .next()方法**

因为 next() (.next()) 是一个如此普通的使用函数（方法），这里有另外一个语法改变（或者是实现上改变了），值得一提的是：在 Python 2.7.5 中函数和方法你都可以使用，next() 函数在 Python 3 中一直保留着（调用 .next() 抛出属性异常）。

***Python 2***

|  |  |
| --- | --- |
| 1  2  3  4 | print 'Python', python\_version()  my\_generator = (letter for letter in 'abcdefg')  next(my\_generator)  my\_generator.next() |

*run result:*

Python 2.7.6

‘b

***Python 3***

|  |  |
| --- | --- |
| 1  2  3 | print('Python', python\_version())  my\_generator = (letter for letter in 'abcdefg')  next(my\_generator) |

*run result:*

Python 3.4.1

‘a’

|  |  |
| --- | --- |
| 1 | my\_generator.next() |

*run result:*

-—————————————————————————————————————

AttributeError Traceback (most recent call last)

< ipython-input-14-125f388bb61b> in < module>()

——> 1 my\_generator.next()

AttributeError: ‘generator’ object has no attribute ‘next’

**For循环变量和全局命名空间泄漏**

好消息：在 Python 3.x 中 for 循环变量不会再导致命名空间泄漏。

在 Python 3.x 中做了一个改变，在 What’s New In Python 3.0 中有如下描述：

“列表推导不再支持 [... for var in item1, item2, ...] 这样的语法。使用 [... for var in (item1, item2, ...)] 代替。也需要提醒的是列表推导有不同的语义： 他们关闭了在 list() 构造器中的生成器表达式的语法糖, 并且特别是循环控制变量不再泄漏进周围的作用范围域.”

***Python 2***

|  |  |
| --- | --- |
| 1  2  3  4  5 | print 'Python', python\_version()  i = 1  print 'before: i =', i  print 'comprehension: ', [i for i in range(5)]  print 'after: i =', i |

*run result:*

Python 2.7.6

before: i = 1

comprehension: [0, 1, 2, 3, 4]

after: i = 4

***Python 3***

|  |  |
| --- | --- |
| 1  2  3  4  5 | print('Python', python\_version())  i = 1  print('before: i =', i)  print('comprehension:', [i for i in range(5)])  print('after: i =', i) |

*run result:*

Python 3.4.1

before: i = 1

comprehension: [0, 1, 2, 3, 4]

after: i = 1

**比较不可排序类型**

在 Python 3 中的另外一个变化就是当对不可排序类型做比较的时候，会抛出一个类型错误。

***Python 2***

|  |  |
| --- | --- |
| 1  2  3  4 | print 'Python', python\_version()  print "[1, 2] > 'foo' = ", [1, 2] > 'foo'  print "(1, 2) > 'foo' = ", (1, 2) > 'foo'  print "[1, 2] > (1, 2) = ", [1, 2] > (1, 2) |

*run result:*

Python 2.7.6

[1, 2] > ‘foo’ = False

(1, 2) > ‘foo’ = True

[1, 2] > (1, 2) = False

***Python 3***

|  |  |
| --- | --- |
| 1  2  3  4 | print('Python', python\_version())  print("[1, 2] > 'foo' = ", [1, 2] > 'foo')  print("(1, 2) > 'foo' = ", (1, 2) > 'foo')  print("[1, 2] > (1, 2) = ", [1, 2] > (1, 2)) |

*run result:*

Python 3.4.1

-—————————————————————————————————————

TypeError Traceback (most recent call last)

< ipython-input-16-a9031729f4a0> in < module>()

1 print(‘Python’, python\_version())

——> 2 print(“[1, 2] > ‘foo’ = “, [1, 2] > ‘foo’)

3 print(“(1, 2) > ‘foo’ = “, (1, 2) > ‘foo’)

4 print(“[1, 2] > (1, 2) = “, [1, 2] > (1, 2))

TypeError: unorderable types: list() > str()

**通过input()解析用户的输入**

幸运的是，在 Python 3 中已经解决了把用户的输入存储为一个 str 对象的问题。为了避免在 Python 2 中的读取非字符串类型的危险行为，我们不得不使用 raw\_input() 代替。

***Python 2***

Python 2.7.6

[GCC 4.0.1 (Apple Inc. build 5493)] on darwin

Type “help”, “copyright”, “credits” or “license” for more information.

>>> my\_input = input('enter a number: ')

enter a number: 123

>>> type(my\_input)  
<type 'int'>

>>> my\_input = raw\_input('enter a number: ')

enter a number: 123

>>> type(my\_input)  
<type 'str'>

***Python 3***

Python 3.4.1

[GCC 4.2.1 (Apple Inc. build 5577)] on darwin

Type “help”, “copyright”, “credits” or “license” for more information.

>>> my\_input = input('enter a number: ')

enter a number: 123

>>> type(my\_input)  
<class 'str'>

**返回可迭代对象，而不是列表**

如果在 xrange 章节看到的，现在在 Python 3 中一些方法和函数返回迭代对象 — 代替 Python 2 中的列表

因为我们通常那些遍历只有一次，我认为这个改变对节约内存很有意义。尽管如此，它也是可能的，相对于生成器 —- 如需要遍历多次。它是不那么高效的。

而对于那些情况下，我们真正需要的是列表对象，我们可以通过 list() 函数简单的把迭代对象转换成一个列表。

***Python 2***

|  |  |
| --- | --- |
| 1  2  3 | print 'Python', python\_version()  print range(3)  print type(range(3)) |

*run result:*

Python 2.7.6

[0, 1, 2]

< type ‘list’>

***Python 3***

|  |  |
| --- | --- |
| 1  2  3  4 | print('Python', python\_version())  print(range(3))  print(type(range(3)))  print(list(range(3))) |

*run result:*

Python 3.4.1

range(0, 3)

< class ‘range’>

[0, 1, 2]

在 Python 3 中一些经常使用到的不再返回列表的函数和方法：

* zip()
* map()
* filter()
* dictionary’s .keys() method
* dictionary’s .values() method
* dictionary’s .items() method

来自 <[*http://chenqx.github.io/2014/11/10/Key-differences-between-Python-2-7-x-and-Python-3-x/*](http://chenqx.github.io/2014/11/10/Key-differences-between-Python-2-7-x-and-Python-3-x/)>

编程规范PEP8

2016年8月2日

19:59

**PEP8 Python 编码规范整理**

一 代码编排

1 缩进。4个空格的缩进（编辑器都可以完成此功能），不使用Tap，更不能混合使用Tap和空格。

2 每行最大长度79，换行可以使用反斜杠，最好使用圆括号。换行点要在操作符的后边敲回车。

3 类和top-level函数定义之间空两行；类中的方法定义之间空一行；函数内逻辑无关段落之间空一行；其他地方尽量不要再空行。

二 文档编排

1 模块内容的顺序：模块说明和docstring—import—globals&constants—其他定义。其中import部分，又按标准、三方和自己编写顺序依次排放，之间空一行。

2 不要在一句import中多个库，比如import os, sys不推荐。

3 如果采用from XX import XX引用库，可以省略‘module.’，都是可能出现命名冲突，这时就要采用import XX。

三 空格的使用

        总体原则，避免不必要的空格。

1 各种右括号前不要加空格。

2 逗号、冒号、分号前不要加空格。

3 函数的左括号前不要加空格。如Func(1)。

4 序列的左括号前不要加空格。如list[2]。

5 操作符左右各加一个空格，不要为了对齐增加空格。

6 函数默认参数使用的赋值符左右省略空格。

7 不要将多句语句写在同一行，尽管使用‘；’允许。

8 if/for/while语句中，即使执行语句只有一句，也必须另起一行。

四 注释

        总体原则，错误的注释不如没有注释。所以当一段代码发生变化时，第一件事就是要修改注释！

        注释必须使用英文，最好是完整的句子，首字母大写，句后要有结束符，结束符后跟两个空格，开始下一句。如果是短语，可以省略结束符。

1 块注释，在一段代码前增加的注释。在‘#’后加一空格。段落之间以只有‘#’的行间隔。比如：

# Description : Module config.

#

# Input : None

#

# Output : None

2 行注释，在一句代码后加注释。比如：x = x + 1 # Increment x

但是这种方式尽量少使用。

3 避免无谓的注释。

五 文档描述

1 为所有的共有模块、函数、类、方法写docstrings；非共有的没有必要，但是可以写注释（在def的下一行）。

2 如果docstring要换行，参考如下例子,详见PEP 257

"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.

"""

六 命名规范

module\_name, packagename, ClassName, method\_name, ExceptionName, function\_name, GLOBAL\_VAR\_NAME, instance\_var\_name, function\_parameter\_name, local\_var\_name.

        总体原则，新编代码必须按下面命名风格进行，现有库的编码尽量保持风格。

1 尽量单独使用小写字母‘l’，大写字母‘O’等容易混淆的字母。

2 模块命名尽量短小，使用全部小写的方式，可以使用下划线。

3 包命名尽量短小，使用全部小写的方式，不可以使用下划线。

4 类的命名使用CapWords的方式，模块内部使用的类采用\_CapWords的方式。

5 异常命名使用CapWords+Error后缀的方式。

6 全局变量尽量只在模块内有效，类似C语言中的static。实现方法有两种，一是\_\_all\_\_机制;二是前缀一个下划线。

7 函数命名使用全部小写的方式，可以使用下划线。

8 常量命名使用全部大写的方式，可以使用下划线。

9 类的属性（方法和变量）命名使用全部小写的方式，可以使用下划线。

9 类的属性有3种作用域public、non-public和subclass API，可以理解成C++中的public、private、protected，non-public属性前，前缀一条下划线。

11 类的属性若与关键字名字冲突，后缀一下划线，尽量不要使用缩略等其他方式。

12 为避免与子类属性命名冲突，在类的一些属性前，前缀两条下划线。比如：类Foo中声明\_\_a,访问时，只能通过Foo.\_Foo\_\_a，避免歧义。如果子类也叫Foo，那就无能为力了。

13 类的方法第一个参数必须是self，而静态方法第一个参数必须是cls。

七 编码建议

1 编码中考虑到其他python实现的效率等问题，比如运算符‘+’在CPython（Python）中效率很高，都是Jython中却非常低，所以应该采用.join()的方式。

2 尽可能使用‘is’‘is not’取代‘==’，比如if x is not None 要优于if x。

3 使用基于类的异常，每个模块或包都有自己的异常类，此异常类继承自Exception。

4 异常中不要使用裸露的except，except后跟具体的exceptions。

5 异常中try的代码尽可能少。比如：

try:

value = collection[key]

except KeyError:

return key\_not\_found(key)

else:

return handle\_value(value)

要优于

try:

# Too broad!

return handle\_value(collection[key])

except KeyError:

# Will also catch KeyError raised by handle\_value()

return key\_not\_found(key)

6 使用startswith() and endswith()代替切片进行序列前缀或后缀的检查。比如：

Yes: if foo.startswith('bar'):优于

No: if foo[:3] == 'bar':

7 使用isinstance()比较对象的类型。比如

Yes: if isinstance(obj, int): 优于

No: if type(obj) is type(1):

8 判断序列空或不空，有如下规则

Yes: if not seq:

if seq:

优于

No: if len(seq)

if not len(seq)

9 字符串不要以空格收尾。

10 二进制数据判断使用 if boolvalue的方式。

来自 <[*https://www.douban.com/note/134971609/*](https://www.douban.com/note/134971609/)>

编程规范Google

Tuesday, September 20, 2016

17:13

**Python 风格规范(Google)** 。本项目并非 Google 官方项目, 而是由国内程序员凭热情创建和维护。如果你关注的是 Google 官方英文版, 请移步 [Google Style Guide](https://google-styleguide.googlecode.com/svn/trunk/pyguide.html)以下代码中 **Yes** 表示推荐，**No** 表示不推荐。

**分号**

不要在行尾加分号, 也不要用分号将两条命令放在同一行。

**行长度**

每行不超过80个字符。以下情况除外：

1. 长的导入模块语句
2. 注释里的URL

不要使用反斜杠连接行。

Python会将 [圆括号, 中括号和花括号中的行隐式的连接起来](http://docs.python.org/2/reference/lexical_analysis.html#implicit-line-joining) , 你可以利用这个特点. 如果需要, 你可以在表达式外围增加一对额外的圆括号。

推荐: foo\_bar(self, width, height, color='black', design=None, x='foo',  
 emphasis=None, highlight=0)

if (width == 0 and height == 0 and  
 color == 'red' and emphasis == 'strong'):

如果一个文本字符串在一行放不下, 可以使用圆括号来实现隐式行连接:

x = ('这是一个非常长非常长非常长非常长 '  
 '非常长非常长非常长非常长非常长非常长的字符串')

在注释中，如果必要，将长的URL放在一行上。

Yes: # See details at  
 # <http://www.example.com/us/developer/documentation/api/content/v2.0/csv_file_name_extension_full_specification.html>

No: # See details at  
 # [http://www.example.com/us/developer/documentation/api/content/\](http://www.example.com/us/developer/documentation/api/content/)  
 # v2.0/csv\_file\_name\_extension\_full\_specification.html

注意上面例子中的元素缩进; 你可以在本文的 :ref:`缩进 <indentation>`部分找到解释.

**括号**

宁缺毋滥的使用括号

除非是用于实现行连接, 否则不要在返回语句或条件语句中使用括号. 不过在元组两边使用括号是可以的.

Yes: if foo:  
 bar()  
 while x:  
 x = bar()  
 if x and y:  
 bar()  
 if not x:  
 bar()  
 return foo  
 for (x, y) in dict.items(): ...

No: if (x):  
 bar()  
 if not(x):  
 bar()  
 return (foo)

**缩进**

用4个空格来缩进代码。绝对不要用tab, 也不要tab和空格混用. 对于行连接的情况, 你应该要么垂直对齐换行的元素(见 :ref:`行长度 <line\_length>` 部分的示例), 或者使用4空格的悬挂式缩进(这时第一行不应该有参数):

Yes: # 与起始变量对齐  
 foo = long\_function\_name(var\_one, var\_two,  
 var\_three, var\_four)

# 字典中与起始值对齐  
 foo = {  
 long\_dictionary\_key: value1 +  
 value2,  
 ...  
 }

# 4 个空格缩进，第一行不需要  
 foo = long\_function\_name(  
 var\_one, var\_two, var\_three,  
 var\_four)

# 字典中 4 个空格缩进  
 foo = {  
 long\_dictionary\_key:  
 long\_dictionary\_value,  
 ...  
 }

No: # 第一行有空格是禁止的  
 foo = long\_function\_name(var\_one, var\_two,  
 var\_three, var\_four)

# 2 个空格是禁止的  
 foo = long\_function\_name(  
 var\_one, var\_two, var\_three,  
 var\_four)

# 字典中没有处理缩进  
 foo = {  
 long\_dictionary\_key:  
 long\_dictionary\_value,  
 ...  
 }

**空行**

顶级定义之间空两行, 方法定义之间空一行。顶级定义之间空两行, 比如函数或者类定义. 方法定义, 类定义与第一个方法之间, 都应该空一行. 函数或方法中, 某些地方要是你觉得合适, 就空一行.

**空格**

按照标准的排版规范来使用标点两边的空格

括号内不要有空格.

按照标准的排版规范来使用标点两边的空格

Yes: spam(ham[1], {eggs: 2}, [])

No: spam( ham[ 1 ], { eggs: 2 }, [ ] )

不要在逗号, 分号, 冒号前面加空格, 但应该在它们后面加(除了在行尾).

Yes: if x == 4:  
 print x, y  
 x, y = y, x

No: if x == 4 :  
 print x , y  
 x , y = y , x

参数列表, 索引或切片的左括号前不应加空格.

Yes: spam(1)

no: spam (1)

Yes: dict['key'] = list[index]

No: dict ['key'] = list [index]

在 二元操作符两边都加上一个空格, 比如赋值(=), 比较(==, <, >, !=, <>, <=, >=, in, not in, is, is not), 布尔(and, or, not). 至于算术操作符两边的空格该如何使用, 需要你自己好好判断. 不过两侧务必要保持一致.

Yes: x == 1

No: x<1

当'='用于指示关键字参数或默认参数值时, 不要在其两侧使用空格.

Yes: def complex(real, imag=0.0): return magic(r=real, i=imag)

No: def complex(real, imag = 0.0): return magic(r = real, i = imag)

不要用空格来垂直对齐多行间的标记, 因为这会成为维护的负担(适用于:, #, =等):

Yes:  
 foo = 1000 # 注释  
 long\_name = 2 # 注释不需要对齐

dictionary = {  
 "foo": 1,  
 "long\_name": 2,  
 }

No:  
 foo = 1000 # 注释  
 long\_name = 2 # 注释不需要对齐

dictionary = {  
 "foo" : 1,  
 "long\_name": 2,  
 }

**Shebang**

大部分.py文件不必以#!作为文件的开始. 根据 [PEP-394](http://www.python.org/dev/peps/pep-0394/) , 程序的main文件应该以 #!/usr/bin/python2或者 #!/usr/bin/python3开始.

(译者注: 在计算机科学中, [Shebang](http://en.wikipedia.org/wiki/Shebang_%28Unix%29) (也称为Hashbang)是一个由井号和叹号构成的字符串行(#!), 其出现在文本文件的第一行的前两个字符. 在文件中存在Shebang的情况下, 类Unix操作系统的程序载入器会分析Shebang后的内容, 将这些内容作为解释器指令, 并调用该指令, 并将载有Shebang的文件路径作为该解释器的参数. 例如, 以指令#!/bin/sh开头的文件在执行时会实际调用/bin/sh程序.)

#!先用于帮助内核找到Python解释器, 但是在导入模块时, 将会被忽略. 因此只有被直接执行的文件中才有必要加入#!.

**注释**

确保对模块, 函数, 方法和行内注释使用正确的风格

**文档字符串**

Python有一种独一无二的的注释方式: 使用文档字符串. 文档字符串是包, 模块, 类或函数里的第一个语句. 这些字符串可以通过对象的\_\_doc\_\_成员被自动提取, 并且被pydoc所用. (你可以在你的模块上运行pydoc试一把, 看看它长什么样). 我们对文档字符串的惯例是使用三重双引号"""( [PEP-257](http://www.python.org/dev/peps/pep-0257/) ). 一个文档字符串应该这样组织: 首先是一行以句号, 问号或惊叹号结尾的概述(或者该文档字符串单纯只有一行). 接着是一个空行. 接着是文档字符串剩下的部分, 它应该与文档字符串的第一行的第一个引号对齐. 下面有更多文档字符串的格式化规范.

**模块**

每个文件应该包含一个许可样板. 根据项目使用的许可(例如, Apache 2.0, BSD, LGPL, GPL), 选择合适的样板.

**函数和方法**

下文所指的函数,包括函数, 方法, 以及生成器.

一个函数必须要有文档字符串, 除非它满足以下条件:

1. 外部不可见
2. 非常短小
3. 简单明了

文档字符串应该包含函数做什么, 以及输入和输出的详细描述. 通常, 不应该描述"怎么做", 除非是一些复杂的算法. 文档字符串应该提供足够的信息, 当别人编写代码调用该函数时, 他不需要看一行代码, 只要看文档字符串就可以了. 对于复杂的代码, 在代码旁边加注释会比使用文档字符串更有意义.

关于函数的几个方面应该在特定的小节中进行描述记录， 这几个方面如下文所述. 每节应该以一个标题行开始. 标题行以冒号结尾. 除标题行外, 节的其他内容应被缩进2个空格.

Args:

列出每个参数的名字, 并在名字后使用一个冒号和一个空格, 分隔对该参数的描述.如果描述太长超过了单行80字符,使用2或者4个空格的悬挂缩进(与文件其他部分保持一致). 描述应该包括所需的类型和含义. 如果一个函数接受\*foo(可变长度参数列表)或者\*\*bar (任意关键字参数), 应该详细列出\*foo和\*\*bar.

Returns: (或者 Yields: 用于生成器)

描述返回值的类型和语义. 如果函数返回None, 这一部分可以省略.

Raises:

列出与接口有关的所有异常.

def fetch\_bigtable\_rows(big\_table, keys, other\_silly\_variable=None):  
 """Fetches rows from a Bigtable.

Retrieves rows pertaining to the given keys from the Table instance  
 represented by big\_table. Silly things may happen if  
 other\_silly\_variable is not None.

Args:  
 big\_table: An open Bigtable Table instance.  
 keys: A sequence of strings representing the key of each table row  
 to fetch.  
 other\_silly\_variable: Another optional variable, that has a much  
 longer name than the other args, and which does nothing.

Returns:  
 A dict mapping keys to the corresponding table row data  
 fetched. Each row is represented as a tuple of strings. For  
 example:

{'Serak': ('Rigel VII', 'Preparer'),  
 'Zim': ('Irk', 'Invader'),  
 'Lrrr': ('Omicron Persei 8', 'Emperor')}

If a key from the keys argument is missing from the dictionary,  
 then that row was not found in the table.

Raises:  
 IOError: An error occurred accessing the bigtable.Table object.  
 """  
 pass

**类**

类应该在其定义下有一个用于描述该类的文档字符串. 如果你的类有公共属性(Attributes), 那么文档中应该有一个属性(Attributes)段. 并且应该遵守和函数参数相同的格式.

class SampleClass(object):  
 """Summary of class here.

Longer class information....  
 Longer class information....

Attributes:  
 likes\_spam: A boolean indicating if we like SPAM or not.  
 eggs: An integer count of the eggs we have laid.  
 """

def \_\_init\_\_(self, likes\_spam=False):  
 """Inits SampleClass with blah."""  
 self.likes\_spam = likes\_spam  
 self.eggs = 0

def public\_method(self):  
 """Performs operation blah."""

**块注释和行注释**

最需要写注释的是代码中那些技巧性的部分. 如果你在下次 [代码审查](http://en.wikipedia.org/wiki/Code_review) 的时候必须解释一下, 那么你应该现在就给它写注释. 对于复杂的操作, 应该在其操作开始前写上若干行注释. 对于不是一目了然的代码, 应在其行尾添加注释.

# We use a weighted dictionary search to find out where i is in  
# the array. We extrapolate position based on the largest num  
# in the array and the array size and then do binary search to  
# get the exact number.

if i & (i-1) == 0: # true iff i is a power of 2

为了提高可读性, 注释应该至少离开代码2个空格.

另一方面, 绝不要描述代码. 假设阅读代码的人比你更懂Python, 他只是不知道你的代码要做什么.

# BAD COMMENT: Now go through the b array and make sure whenever i occurs  
# the next element is i+1

**类**

如果一个类不继承自其它类, 就显式的从object继承. 嵌套类也一样.

Yes: class SampleClass(object):  
 pass

class OuterClass(object):

class InnerClass(object):  
 pass

class ChildClass(ParentClass):  
 """Explicitly inherits from another class already."""

No: class SampleClass:  
 pass

class OuterClass:

class InnerClass:  
 pass

继承自 object 是为了使属性(properties)正常工作, 并且这样可以保护你的代码, 使其不受Python 3000的一个特殊的潜在不兼容性影响. 这样做也定义了一些特殊的方法, 这些方法实现了对象的默认语义, 包括 \_\_new\_\_, \_\_init\_\_, \_\_delattr\_\_, \_\_getattribute\_\_, \_\_setattr\_\_, \_\_hash\_\_, \_\_repr\_\_, and \_\_str\_\_ .

**字符串**

Yes: x = a + b  
 x = '%s, %s!' % (imperative, expletive)  
 x = '{}, {}!'.format(imperative, expletive)  
 x = 'name: %s; score: %d' % (name, n)  
 x = 'name: {}; score: {}'.format(name, n)

No: x = '%s%s' % (a, b) # use + in this case  
 x = '{}{}'.format(a, b) # use + in this case  
 x = imperative + ', ' + expletive + '!'  
 x = 'name: ' + name + '; score: ' + str(n)

避免在循环中用+和+=操作符来累加字符串. 由于字符串是不可变的, 这样做会创建不必要的临时对象, 并且导致二次方而不是线性的运行时间. 作为替代方案, 你可以将每个子串加入列表, 然后在循环结束后用 .join 连接列表. (也可以将每个子串写入一个 cStringIO.StringIO 缓存中.)

Yes: items = ['<table>']  
 for last\_name, first\_name in employee\_list:  
 items.append('<tr><td>%s, %s</td></tr>' % (last\_name, first\_name))  
 items.append('</table>')  
 employee\_table = ''.join(items)

No: employee\_table = '<table>'  
 for last\_name, first\_name in employee\_list:  
 employee\_table += '<tr><td>%s, %s</td></tr>' % (last\_name, first\_name)  
 employee\_table += '</table>'

在同一个文件中, 保持使用字符串引号的一致性. 使用单引号'或者双引号"之一用以引用字符串, 并在同一文件中沿用. 在字符串内可以使用另外一种引号, 以避免在字符串中使用. PyLint已经加入了这一检查.

Yes:  
 Python('Why are you hiding your eyes?')  
 Gollum("I'm scared of lint errors.")  
 Narrator('"Good!" thought a happy Python reviewer.')

No:  
 Python("Why are you hiding your eyes?")  
 Gollum('The lint. It burns. It burns us.')  
 Gollum("Always the great lint. Watching. Watching.")

为多行字符串使用三重双引号"""而非三重单引号'''. 当且仅当项目中使用单引号'来引用字符串时, 才可能会使用三重'''为非文档字符串的多行字符串来标识引用. 文档字符串必须使用三重双引号""". 不过要注意, 通常用隐式行连接更清晰, 因为多行字符串与程序其他部分的缩进方式不一致.

Yes:  
 print ("This is much nicer.\n"  
 "Do it this way.\n")

No:  
 print """This is pretty ugly.  
 Don't do this.  
 """

**文件和sockets**

在文件和sockets结束时, 显式的关闭它.

除文件外, sockets或其他类似文件的对象在没有必要的情况下打开, 会有许多副作用, 例如:

1. 它们可能会消耗有限的系统资源, 如文件描述符. 如果这些资源在使用后没有及时归还系统, 那么用于处理这些对象的代码会将资源消耗殆尽.
2. 持有文件将会阻止对于文件的其他诸如移动、删除之类的操作.
3. 仅仅是从逻辑上关闭文件和sockets, 那么它们仍然可能会被其共享的程序在无意中进行读或者写操作. 只有当它们真正被关闭后, 对于它们尝试进行读或者写操作将会跑出异常, 并使得问题快速显现出来.

而且, 幻想当文件对象析构时, 文件和sockets会自动关闭, 试图将文件对象的生命周期和文件的状态绑定在一起的想法, 都是不现实的. 因为有如下原因:

1. 没有任何方法可以确保运行环境会真正的执行文件的析构. 不同的Python实现采用不同的内存管理技术, 比如延时垃圾处理机制. 延时垃圾处理机制可能会导致对象生命周期被任意无限制的延长.
2. 对于文件意外的引用,会导致对于文件的持有时间超出预期(比如对于异常的跟踪, 包含有全局变量等).

推荐使用 ["with"语句](http://docs.python.org/reference/compound_stmts.html#the-with-statement) 以管理文件:

with open("hello.txt") as hello\_file:  
 for line in hello\_file:  
 print line

对于不支持使用"with"语句的类似文件的对象,使用 contextlib.closing():

import contextlib

with contextlib.closing(urllib.urlopen("http://www.python.org/")) as front\_page:  
 for line in front\_page:  
 print line

Legacy AppEngine 中Python 2.5的代码如使用"with"语句, 需要添加 "from \_\_future\_\_ import with\_statement".

**TODO注释**

为临时代码使用TODO注释, 它是一种短期解决方案. 不算完美, 但够好了.

TODO注释应该在 所有开头处包含"TODO"字符串, 紧跟着是用括号括起来的你的名字, email地址或其它标识符. 然后是一个可选的冒号. 接着必须有一行注释, 解释要做什么. 主要目的是为了有一个统一的TODO格式, 这样添加注释的人就可以搜索到(并可以按需提供更多细节). 写了TODO注释并不保证写的人会亲自解决问题. 当你写了一个TODO, 请注上你的名字.

# TODO(kl@gmail.com): Use a "\*" here for string repetition.  
# TODO(Zeke) Change this to use relations.

如果你的TODO是"将来做某事"的形式, 那么请确保你包含了一个指定的日期("2009年11月解决")或者一个特定的事件("等到所有的客户都可以处理XML请求就移除这些代码").

**导入格式**

每个导入应该独占一行

Yes: import os  
 import sys

No: import os, sys

导入总应该放在文件顶部, 位于模块注释和文档字符串之后, 模块全局变量和常量之前. 导入应该按照从最通用到最不通用的顺序分组:

1. 标准库导入
2. 第三方库导入
3. 应用程序指定导入

每种分组中, 应该根据每个模块的完整包路径按字典序排序, 忽略大小写.

import foo  
from foo import bar  
from foo.bar import baz  
from foo.bar import Quux  
from Foob import ar

**语句**

通常每个语句应该独占一行

不过, 如果测试结果与测试语句在一行放得下, 你也可以将它们放在同一行. 如果是if语句, 只有在没有else时才能这样做. 特别地, 绝不要对 try/except 这样做, 因为try和except不能放在同一行.

Yes:

if foo: bar(foo)

No:

if foo: bar(foo)  
 else: baz(foo)

try: bar(foo)  
 except ValueError: baz(foo)

try:  
 bar(foo)  
 except ValueError: baz(foo)

**访问控制**

在Python中, 对于琐碎又不太重要的访问函数, 你应该直接使用公有变量来取代它们, 这样可以避免额外的函数调用开销. 当添加更多功能时, 你可以用属性(property)来保持语法的一致性.

(译者注: 重视封装的面向对象程序员看到这个可能会很反感, 因为他们一直被教育: 所有成员变量都必须是私有的! 其实, 那真的是有点麻烦啊. 试着去接受Pythonic哲学吧)

另一方面, 如果访问更复杂, 或者变量的访问开销很显著, 那么你应该使用像 get\_foo() 和 set\_foo() 这样的函数调用. 如果之前的代码行为允许通过属性(property)访问 , 那么就不要将新的访问函数与属性绑定. 这样, 任何试图通过老方法访问变量的代码就没法运行, 使用者也就会意识到复杂性发生了变化.

**命名**

module\_name, package\_name, ClassName, method\_name, ExceptionName, function\_name, GLOBAL\_VAR\_NAME, instance\_var\_name, function\_parameter\_name, local\_var\_name.

**应该避免的名称**

1. 单字符名称, 除了计数器和迭代器.
2. 包/模块名中的连字符(-)
3. 双下划线开头并结尾的名称(Python保留, 例如\_\_init\_\_)

**命名约定**

1. 所谓"内部(Internal)"表示仅模块内可用, 或者, 在类内是保护或私有的.
2. 用单下划线(\_)开头表示模块变量或函数是protected的(使用import \* from时不会包含).
3. 用双下划线(\_\_)开头的实例变量或方法表示类内私有.
4. 将相关的类和顶级函数放在同一个模块里. 不像Java, 没必要限制一个类一个模块.
5. 对类名使用大写字母开头的单词(如CapWords, 即Pascal风格), 但是模块名应该用小写加下划线的方式(如lower\_with\_under.py). 尽管已经有很多现存的模块使用类似于CapWords.py这样的命名, 但现在已经不鼓励这样做, 因为如果模块名碰巧和类名一致, 这会让人困扰.

**Python之父Guido推荐的规范**

|  |  |  |
| --- | --- | --- |
| **Type** | **Public** | **Internal** |
| Modules | lower\_with\_under | \_lower\_with\_under |
| Packages | lower\_with\_under |  |
| Classes | CapWords | \_CapWords |
| Exceptions | CapWords |  |
| Functions | lower\_with\_under() | \_lower\_with\_under() |
| Global/Class Constants | CAPS\_WITH\_UNDER | \_CAPS\_WITH\_UNDER |
| Global/Class Variables | lower\_with\_under | \_lower\_with\_under |
| Instance Variables | lower\_with\_under | \_lower\_with\_under (protected) or \_\_lower\_with\_under (private) |
| Method Names | lower\_with\_under() | \_lower\_with\_under() (protected) or \_\_lower\_with\_under() (private) |
| Function/Method Parameters | lower\_with\_under |  |
| Local Variables | lower\_with\_under |  |

**Main**

即使是一个打算被用作脚本的文件, 也应该是可导入的. 并且简单的导入不应该导致这个脚本的主功能(main functionality)被执行, 这是一种副作用. 主功能应该放在一个main()函数中.

在Python中, pydoc以及单元测试要求模块必须是可导入的. 你的代码应该在执行主程序前总是检查 if \_\_name\_\_ == '\_\_main\_\_' , 这样当模块被导入时主程序就不会被执行.

def main():  
 ...

if \_\_name\_\_ == '\_\_main\_\_':  
 main()

所有的顶级代码在模块导入时都会被执行. 要小心不要去调用函数, 创建对象, 或者执行那些不应该在使用pydoc时执行的操作.

来自 <[*http://www.runoob.com/w3cnote/google-python-styleguide.html*](http://www.runoob.com/w3cnote/google-python-styleguide.html)>

慕课/开发环境

2016年8月31日

9:36

学会编程，最重要的不是天赋，而是决心。

很多技术活，是一门手艺。手艺，最重要是技术熟练和精湛。

• 注重能力培养和实践。读和写、注重细节、发现不同、不要复制粘贴而要手打代码，足够的代码练习，让程序运行起来。自己专研问题和寻求答案的能力。

• 注重好习惯的养成。怎样写出好的代码、好的注释、好的项目。

坚持练习。“万事开头难”，对于有价值的事情尤其如 此。要学好一样东西，每天练习必不可少。也许你害怕失败，一碰到困难就想放弃；也许你缺乏自律，一碰到“无聊” 的事情就不想上手；也许你被人夸“有天分”而自视甚高，不愿意做些看上去很笨拙的事情， 怕有负你”神童”的称号；也许你太过激进，把自己跟编程老手相比，让自己失去了信 心。如果这天自己的状态很差、碰到做不出来的、甚至看不懂的习题， 你可以暂时跳过去，过一阵子回来再看。一开始你可能什么都看不懂，这会让你感觉很不舒服。如果你坚持练习下去，坚持去上下求索，你最终会学会这些东西的。

Python创立者是数学科班出身，所以Python具有高度统一性，而Perl的创立者是语言学科班出身，所以Perl表达自由而微妙，对环境敏感。 但是，工程与艺术不同，在工程世界，我们需要的是最小化功能集合与可预测性，自由表达是一种噩梦。代码是为读者(维护或者重用)写的。

* 1. **简介**

Python（英音：/ˈpaɪθən/ 美音：/ˈpaɪθɑːn/）,意思“巨蟒”， 是一种面向对象的解释型计算机程序设计语言，由荷兰人Guido van Rossum于1989年发明，第一个公开发行版发行于1991年。读作“派森”，。Python的特点：优雅、明确、简单。 缺点：运行速度慢（并没有太大的影响），跨平台，不能加密，优点：代码少 ，易懂。Python与其他语言对比

|  |  |  |  |
| --- | --- | --- | --- |
|  | 类型 | 运行速度 | 代码量 |
| C | 编译为机器码 | 非常快 | 非常多 |
| Java | 编译为字节码 | 快 | 多 |
| Python | 解释执行 | 慢 | 少 |

用 Python程序通常比 C、C++ 或 Java 更短小，是因为: 高级数据结构使你可以在一条语句中表达复杂的操作； 语句组使用缩进代替开始和结束大括号来组织； 变量或参数无需声明。

Python适合的领域：Web网站和各种网络服务、系统工具和脚本、作为胶水语言把其他语言开发的模块包装起来方便使用。不适合的领域：贴近硬件的代码（首选C）；移动开发（ios用oc，swift；安卓Java）；游戏开发（C和 C++）。实际应用：Youtube、豆瓣、搜狐邮箱、openstack……google、yahoo、NASA

Python 是 可扩展 的：如果你会 C 语言编程便可以轻易地为解释器添加内置函数或模块，或者为了对性能瓶颈作优化，或者将 Python 程序与只有二进制形式的库（比如某个专业的商业图形库）连接起来。一旦你真正掌握了它，你可以将 Python 解释器集成进某个 C 应用程序，并把它当作那个程序的扩展或命令行语言。

### The Zen of Python

**>>> import this**

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than \*right\* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

* 1. Python语法更简洁

Python的优势是入门快、开发效率高，写法灵活，代码易于理解（本来代码就是给人看的嘛，计算机才不关心你写的漂不漂亮呢），再也不需要纠结花括号是另起一行还是紧接在方法后面了。

* 1. Python是一门动态强类型语言

很多初学者认为Python是弱类型语言，因为定义变量的时候不需要指定类型，而且变量随时可以改变其类型，事实上Python是“强”类型语言，变量名只是一个名字而已，而该变量所绑定的对象在对象创建初期就确定好了类型的，你永远也不可能再改变其类型，因此Python是强类型语言。而Java是一门静态强类型语言。

* 1. Python中一切皆是对象

在Python中，不论是数值（整型、浮点型），字符串，字典，元组对象，还是他们所对应的类型，以及函数，模块等你所能看到的都是对象，他们的祖先是PyObject。而Java中至少函数，基本数据类型都不算对象。

* 1. Python的GIL（缺点）

Python中的GIL一直被不少开发者所诟病，GIL是指全局解释锁，Python的多线程在多CPU条件下并不能并行的运行，而只能是每个线程运行的时候首先需要获得解释器的访问权限才可以执行，其他线程只能处于等待的过程，不过GIL并不是Python的致命缺点，否则现在没人会用的，因为有很多方案是协程，或者是用多进程机制来弥补。而Java支持真正的多线程并发操作，能很好的控制资源的的共享。

* 1. Python的版本

Python社区一直维护两个差异比较大的版本，Python2.x和Python3.x，目前觉得绝大部分公司的生产环境还是运行在Python2中，而Python3.x不提供像后兼容，但是Python3.x从性能上很做了很多改善，所以目前Python处在一个比较尴尬的位置，不过好消息是绝大部分第三方库开始兼容Python3。希望大家早日迁移到Python3。而JDK从1.0到1.9都是逐步的改进，升级对开发者比较友好。

* 1. 关于垃圾收集

Python的垃圾收集机制主要使用的引用计数方式

* 1. 文化

Java是一门商业化味道很重的语言.James Gosling 早就不关心Java了.或者他觉得现在的Java已经不是当年他想要的Java吧。Python是一门有情怀的语言, Guido van Rossum 一直小心翼翼地维护着Python的发展，人称Guido为仁慈的独裁者

* 1. 开发环境搭建，第一个Python程序
  + **安装**

<https://www.python.org/> 安装Python，**2.X版**，3.X版，两个语法不兼容。winXP只能运行3.4版。安装的时候，**勾选“Add Python to environment variables”**

在win10上安装python官网上下载的msi安装包出现错误: there is a problem with widows installer package,a program run as part of setup …如果以管理员权限运行msi安装文件, 就能安装成功. 我的电脑 -> 文件 -> 打开命令行提示符 -> 以管理员权限运行命令行… 然后在命令行中进入安装文件所在目录, 运行之。比如 D:\Downloads>Python-2.7.11.amd.msi

下载之后，就和装其他软件一样，双击，一路Next，想换安装路径的同学可以换个位置。但不管换不换，请把这个路径复制下来，比如我的是“C：\python27\”，后面要用到它。

安装结束还没完，我们还差最后一步：设置环境变量。这是什么东西我暂时先不解释，大家照着做就好。右键单击我的电脑（不，是你的电脑），依次点击"属性"->"高级"->"环境变量"，在“系统变量”表单中点击叫做Path的变量，然后编辑这个变量，把“;C:\Python27\”，也就是你刚才复制的安装路径，加到它的结尾。注意！要用英文分号和前面已有的内容隔开。

当前的版本安装中将会默认已经安装了setuptools和pip这两个Python的基本工具。如果使用了比较旧的Python版本的话，需要自行安装这两个工具。

setuptools：Python的基础工具包，用来构建、安装卸载Python程序

pip：Python软件包的安装和管理工具。通过pip可以简单的安装Python的任意类库

**软件包，依赖环境** <http://www.lfd.uci.edu/~gohlke/pythonlibs> <https://pypi.python.org>

* + **Python程序执行的过程：**

安装好了后，IDLE (Python GUI)是命令行开发环境，可以直接写语句并且运行。

也可以用文件的形式，file-new file，写好后，保存到桌面。运行已经保存的.py文件file-open, Run-Run Module。

Python文件**A.py**——Python编译器编译成字节码（**A.pyc/A.pyo**）——Python Virtue Machine（PVM，Python解释器，虚拟机）变成二进制指令，让内存/CPU运行，运行结果

Python没有 build 和 make的步骤。

.pyc加载更快，运行无差别；.pyo，优化后的文件，运行更快，然而并不是二进制文件，所以与平台无关。这些都是通过源代码编译字节码，然后在虚拟机运行。上面是CPython，也有其他的实现方式。Jython，可以让py文件变成Java程序一样，可与Java代码集成；IronPython，可以让py文件变得像.net程序一样，可与C#代码集成

* + **交互模式**

* + **模块模式：第一个Python程序：**

$ python ex/ex1.py #在命令行终端输入命令来运行 ex1.py 脚本

File "ex/ex1.py", line 3 # ex1.py 文件的第 3 行有一个错误

print "I like typing this. #这一行的内容被打印出来。 然后 Python 打印出一个 ^ (井号，caret) 符号，用来指示出错的位置。少了一个 " (双引号，double-quote)

SyntaxError: EOL while scanning string literal #“语法错误(SyntaxError)”

* + **IDE/Integrated Development Environment**

IDLE、Eclipse、Pycharm

**Python3** 数值

Friday, September 23, 2016

12:42

* 1. 数值类型简介

python中数有四种类型：长整数**int**、浮点数**float**、布尔值**bool**和复数**complex**。**数值类型实例**

|  |  |  |
| --- | --- | --- |
| **int** | **float** | **complex** |
| 10 | 0.0 | 3.14j |
| -786 | -21.9 | 9.322e-36j |
| 080 | 32.3+e18 | .876j |
| -0490 | -90. | -.6545+0J |
| -0x260 | -32.54e100 | 3e+26J |
| 0x69 | 70.2-E12 | 4.53e-7j |

* 1. Python3 布尔值(Bool)
  + **布尔值与逻辑判断**

**任何程序都使用顺序、选择、循环三种基本控制结构构造，而控制的根本在于逻辑判断。**大量的复杂程序在根本上都是建立在“真”与“假”的基本逻辑之上。计算机用布尔运算来做条件判断，bool值所表示的就是这种最单纯最本质的True/Flase，真与假，是与非，计算机就以此自动执行不同的后续代码。

通过逻辑运算，我们就得到了一个bool值。常用的逻辑运算符包括：

>大于，<小于，>=大于等于，<=小于等于，==等于，!=不等与，

not：逻辑“非”运算。把True变为False，如果x为True，则not x为False，反之则反；

and：逻辑“与”运算。只有两个布尔值都为 True 时，计算结果才为 True；

or：逻辑“或”运算。只要有一个布尔值为 True，计算结果就是 True。

在Python中，布尔类型还可以与其他数据类型做 and、or和not运算，请看下面的代码：

a = True  
print a and 'a=T' or 'a=F'

计算结果不是布尔类型，而是字符串 'a=T'，这是为什么呢？

因为Python把0、空字符串''和None看成 False，其他数值和非空字符串都看成 True，所以：True and 'a=T' 计算结果是 'a=T' 而继续计算 'a=T' or 'a=F' 计算结果还是 'a=T' 为什么？

**要解释上述结果，又涉及到 and 和 or 运算的一条重要法则：短路计算。**

1. 在计算 a and b 时，如果 a 是 False，则根据与运算法则，整个结果必定为 False，因此返回 a；如果 a 是 True，则整个计算结果必定取决与 b，因此返回 b。

2. 在计算 a or b 时，如果 a 是 True，则根据或运算法则，整个计算结果必定为 True，因此返回 a；如果 a 是 False，则整个计算结果必定取决于 b，因此返回 b。

所以Python解释器在做布尔运算时，只要能提前确定计算结果，它就不会往后算了，直接返回结果。

试试把bool加到我们的小游戏里：

num = 10

print 'Guess what I think?'

answer = input()

result = answer<num

print 'too small?'

print result

result = answer>num

print 'too big?'

print result

result = answer==num

print 'equal?'

print result

代码比之前稍微多了一点，解释一下。

第一段代码：先创建一个值为10的变量num，输出一句提示，然后再输入一个值给变量answer。

第二段代码：计算answer<num的结果，记录在result里，输出提示，再输出结果。

第三段、第四段都与第二段类似，只是比较的内容不一样。

来自 <[*http://www.imooc.com/code/3273*](http://www.imooc.com/code/3273)>

* + Python3 数字(Number)

Python 数字数据类型用于存储数值。

数据类型是不允许改变的,这就意味着如果改变数字数据类型得值，将重新分配内存空间。

以下实例在变量赋值时 Number 对象将被创建：

var1 = 1  
var2 = 10

您也可以使用del语句删除一些数字对象的引用。

del语句的语法是：

del var1[,var2[,var3[....,varN]]]]

您可以通过使用del语句删除单个或多个对象的引用，例如：

del var  
del var\_a, var\_b

Python 支持三种不同的数值类型：

* + **整型(Int)** - 通常被称为是整型或整数，是正或负整数，不带小数点。Python3 整型是没有限制大小的，可以当作 Long 类型使用，所以 Python3 没有 Python2 的 Long 类型。
  + **浮点型(float)** - 浮点型由整数部分与小数部分组成，浮点型也可以使用科学计数法表示（2.5e2 = 2.5 x 102 = 250）
  + **复数( (complex))** - 复数由实数部分和虚数部分构成，可以用a + bj,或者complex(a,b)表示， 复数的实部a和虚部b都是浮点型。

我们可以使用十六进制和八进制来代表整数：

>>> number = 0xA0F # 十六进制  
>>> number  
2575

>>> number=0o37 # 八进制  
>>> number  
31

|  |  |  |
| --- | --- | --- |
| **int** | **float** | **complex** |
| 10 | 0.0 | 3.14j |
| 100 | 15.20 | 45.j |
| -786 | -21.9 | 9.322e-36j |
| 080 | 32.3+e18 | .876j |
| -0490 | -90. | -.6545+0J |
| -0x260 | -32.54e100 | 3e+26J |
| 0x69 | 70.2-E12 | 4.53e-7j |

* + Python支持复数，复数由实数部分和虚数部分构成，可以用a + bj,或者complex(a,b)表示， 复数的实部a和虚部b都是浮点型。

* + **Python 数字类型转换**

有时候，我们需要对数据内置的类型进行转换，数据类型的转换，你只需要将数据类型作为函数名即可。

* + **int(x)** 将x转换为一个整数。
  + **float(x)** 将x转换到一个浮点数。
  + **complex(x)** 将x转换到一个复数，实数部分为 x，虚数部分为 0。
  + **complex(x, y)** 将 x 和 y 转换到一个复数，实数部分为 x，虚数部分为 y。x 和 y 是数字表达式。

以下实例将浮点数变量 a 转换为整数：

>>> a = 1.0  
>>> int(a)  
1

* + **Python 数字运算**

和数学运算不同的地方是，Python的整数运算结果仍然是整数，浮点数运算结果仍然是浮点数：

1 + 2 # ==> 整数 3  
1.0 + 2.0 # ==> 浮点数 3.0

但是整数和浮点数混合运算的结果就变成浮点数了：

1 + 2.0 # ==> 浮点数 3.0

为什么要区分整数运算和浮点数运算呢？这是因为整数运算的结果永远是精确的，而浮点数运算的结果不一定精确，因为计算机内存再大，也无法精确表示出无限循环小数，比如 0.1 换成二进制表示就是无限循环小数。

那整数的除法运算遇到除不尽的时候，Python的整数除法，即使除不尽，结果仍然是整数，余数直接被扔掉。不过，Python提供了一个求余的运算 % 可以计算余数：

11 % 4 # ==> 3

如果我们要计算 11 / 4 的精确结果，按照“整数和浮点数混合运算的结果是浮点数”的法则，把两个数中的一个变成浮点数再运算就没问题了：

11.0 / 4 # ==> 2.75

来自 <[*http://www.imooc.com/code/3272*](http://www.imooc.com/code/3272)>

Python 解释器可以作为一个简单的计算器，您可以在解释器里输入一个表达式，它将输出表达式的值。

表达式的语法很直白： +, -, \* 和 / 和其它语言（如Pascal或C）里一样。例如：

>>> 2 + 2  
4  
>>> 50 - 5\*6  
20  
>>> (50 - 5\*6) / 4  
5.0  
>>> 8 / 5 # 总是返回一个浮点数  
1.6

**注意：**在不同的机器上浮点运算的结果可能会不一样。

在整数除法中，除法（/）总是返回一个浮点数，如果只想得到整数的结果，丢弃可能的分数部分，可以使用运算符 **//** ：

>>> 17 / 3 # 整数除法返回浮点型  
5.666666666666667  
>>>  
>>> 17 // 3 # 整数除法返回向下取整后的结果  
5  
>>> 17 % 3 # ％操作符返回除法的余数  
2  
>>> 5 \* 3 + 2   
17

等号（=）用于给变量赋值。赋值之后，除了下一个提示符，解释器不会显示任何结果。

>>> width = 20  
>>> height = 5\*9  
>>> width \* height  
900

Python 可以使用 **\*\*** 操作来进行幂运算：

>>> 5 \*\* 2 # 5 的平方  
25  
>>> 2 \*\* 7 # 2的7次方  
128

变量在使用前必须先"定义"（即赋予变量一个值），否则会出现错误：

>>> n # 尝试访问一个未定义的变量  
Traceback (most recent call last):  
 File "<stdin>", line 1, in <module>  
NameError: name 'n' is not defined

不同类型的数混合运算时会将整数转换为浮点数：

>>> 3 \* 3.75 / 1.5  
7.5  
>>> 7.0 / 2  
3.5

在交互模式中，最后被输出的表达式结果被赋值给变量 **\_** 。例如：

>>> tax = 12.5 / 100  
>>> price = 100.50  
>>> price \* tax  
12.5625  
>>> price + \_  
113.0625  
>>> round(\_, 2)  
113.06

此处， **\_** 变量应被用户视为只读变量。

* + **相关**

**数学函数**

|  |  |
| --- | --- |
| **函数** | **返回值 ( 描述 )** |
| [abs(x)](http://www.runoob.com/python3/python3-func-number-abs.html) | 返回数字的绝对值，如abs(-10) 返回 10 |
| [ceil(x)](http://www.runoob.com/python3/python3-func-number-ceil.html) | 返回数字的上入整数，如math.ceil(4.1) 返回 5 |
| cmp(x, y) | 如果 x < y 返回 -1, 如果 x == y 返回 0, 如果 x > y 返回 1。 **Python 3 已废弃** 。使用 **使用 (x>y)-(x<y)** 替换。 |
| [exp(x)](http://www.runoob.com/python3/python3-func-number-exp.html) | 返回e的x次幂(ex),如math.exp(1) 返回2.718281828459045 |
| [fabs(x)](http://www.runoob.com/python3/python3-func-number-fabs.html) | 返回数字的绝对值，如math.fabs(-10) 返回10.0 |
| [floor(x)](http://www.runoob.com/python3/python3-func-number-floor.html) | 返回数字的下舍整数，如math.floor(4.9)返回 4 |
| [log(x)](http://www.runoob.com/python3/python3-func-number-log.html) | 如math.log(math.e)返回1.0,math.log(100,10)返回2.0 |
| [log10(x)](http://www.runoob.com/python3/python3-func-number-log10.html) | 返回以10为基数的x的对数，如math.log10(100)返回 2.0 |
| [max(x1, x2,...)](http://www.runoob.com/python3/python3-func-number-max.html) | 返回给定参数的最大值，参数可以为序列。 |
| [min(x1, x2,...)](http://www.runoob.com/python3/python3-func-number-min.html) | 返回给定参数的最小值，参数可以为序列。 |
| [modf(x)](http://www.runoob.com/python3/python3-func-number-modf.html) | 返回x的整数部分与小数部分，两部分的数值符号与x相同，整数部分以浮点型表示。 |
| [pow(x, y)](http://www.runoob.com/python3/python3-func-number-pow.html) | x\*\*y 运算后的值。 |
| [round(x [,n])](http://www.runoob.com/python3/python3-func-number-round.html) | 返回浮点数x的四舍五入值，如给出n值，则代表舍入到小数点后的位数。 |
| [sqrt(x)](http://www.runoob.com/python3/python3-func-number-sqrt.html) | 返回数字x的平方根，数字可以为负数，返回类型为实数，如math.sqrt(4)返回 2+0j |

**随机数函数**

随机数可以用于数学，游戏，安全等领域中，还经常被嵌入到算法中，用以提高算法效率，并提高程序的安全性。

Python包含以下常用随机数函数：

|  |  |
| --- | --- |
| **函数** | **描述** |
| [choice(seq)](http://www.runoob.com/python3/python3-func-number-choice.html) | 从序列的元素中随机挑选一个元素，比如random.choice(range(10))，从0到9中随机挑选一个整数。 |
| [randrange ([start,] stop [,step])](http://www.runoob.com/python3/python3-func-number-randrange.html) | 从指定范围内，按指定基数递增的集合中获取一个随机数，基数缺省值为1 |
| [random()](http://www.runoob.com/python3/python3-func-number-random.html) | 随机生成下一个实数，它在[0,1)范围内。 |
| [seed([x])](http://www.runoob.com/python3/python3-func-number-seed.html) | 改变随机数生成器的种子seed。如果你不了解其原理，你不必特别去设定seed，Python会帮你选择seed。 |
| [shuffle(lst)](http://www.runoob.com/python3/python3-func-number-shuffle.html) | 将序列的所有元素随机排序 |
| [uniform(x, y)](http://www.runoob.com/python3/python3-func-number-uniform.html) | 随机生成下一个实数，它在[x,y]范围内。 |

**三角函数**

Python包括以下三角函数：

|  |  |
| --- | --- |
| **函数** | **描述** |
| [acos(x)](http://www.runoob.com/python3/python3-func-number-acos.html) | 返回x的反余弦弧度值。 |
| [asin(x)](http://www.runoob.com/python3/python3-func-number-asin.html) | 返回x的反正弦弧度值。 |
| [atan(x)](http://www.runoob.com/python3/python3-func-number-atan.html) | 返回x的反正切弧度值。 |
| [atan2(y, x)](http://www.runoob.com/python3/python3-func-number-atan2.html) | 返回给定的 X 及 Y 坐标值的反正切值。 |
| [cos(x)](http://www.runoob.com/python3/python3-func-number-cos.html) | 返回x的弧度的余弦值。 |
| [hypot(x, y)](http://www.runoob.com/python3/python3-func-number-hypot.html) | 返回欧几里德范数 sqrt(x\*x + y\*y)。 |
| [sin(x)](http://www.runoob.com/python3/python3-func-number-sin.html) | 返回的x弧度的正弦值。 |
| [tan(x)](http://www.runoob.com/python3/python3-func-number-tan.html) | 返回x弧度的正切值。 |
| [degrees(x)](http://www.runoob.com/python3/python3-func-number-degrees.html) | 将弧度转换为角度,如degrees(math.pi/2) ， 返回90.0 |
| [radians(x)](http://www.runoob.com/python3/python3-func-number-radians.html) | 将角度转换为弧度 |

**数学常量**

|  |  |
| --- | --- |
| **常量** | **描述** |
| pi | 数学常量 pi（圆周率，一般以π来表示） |
| e | 数学常量 e，e即自然常数（自然常数）。 |

来自 <[*http://www.runoob.com/python3/python3-number.html*](http://www.runoob.com/python3/python3-number.html)>

**Python3 字符串**

Tuesday, August 16, 2016

16:40

* + **字符串**

python中单引号和双引号使用完全相同。

使用三引号('''或""")可以指定一个多行字符串。

转义符 '\'

自然字符串， 通过在字符串前加r或R。 如 r"this is a line with \n" 则\n会显示，并不是换行。

python允许处理unicode字符串，加前缀u或U， 如 u"this is an unicode string"。

字符串是不可变的。

按字面意义级联字符串，如"this " "is " "string"会被自动转换为this is string。

word = '字符串'

sentence = "这是一个句子。"

paragraph = """这是一个段落，

可以由多行组成"""

* + 定义字符串

字符串就是一组字符的序列。单行字符串可以用英文单引号（‘’）和双引号（""）括起来表示。注意，Python 没有单独的字符类型（就像C语言的Char），一个字符就是长度为1的字符串。

print ('good') # 可以直接输出一个字符串，单引号双引号均可以

print ("I'm OK") # 双引号，输出单引号，I'm OK

print ('Learn "Python" in imooc') # 单引号，输出双引号，Learn "Python" in imooc

str = ‘bad’ # 也可以用一个变量来保存字符串，然后输出

print str

使用反斜杠(\)转义特殊字符。转义字符 \ 不计入字符串的内容中，可以用来输出特殊字符。\\ 表示字符串中的\ ，\n表示字符串中的换行，\t 制表符，\还有个用处，就是用来在代码中换行，分割很长的代码：

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \(行尾) | \\ | \' | \" | \a | \b | \e | \000 | \n | \v | \t | \r | \f |
| 续行符 | \ | ' | " | 响铃 | Backspace | 转义 | 空 | 换行 | 纵向制表符 | 横向制表符 | 回车 | 换页 |

print ('Bob said \"I\'m OK\".') # 用\进行转义，输出单引号与双引号， Bob said "I'm OK".

"this is the\

same line" # 和"this is thesame line"是一样的。

* + **for in遍历**

word = 'helloworld'

for c in word:

print c

* + **索引、切片**

字符串有两种索引方式，从左往右以0开始值，从右往左以-1开始值。

加号 (+) 是字符串的连接符， 星号 (\*) 表示复制当前字符串，紧跟的数字为复制的次数。字符串的切片的语法格式：变量[头下标:尾下标]。实例如下：

str = 'Runoob'

print (str) # 输出字符串。结果：Runoob

print (str[0]) # 索引，输出字符串第一个字符。结果：R

print(str[:]) # 切片，输出整个字符串

print (str[0:-1]) # 输出第一个个到倒数第一个的所有字符（但不包括倒数第一）。结果：Runoo  
print (str[2:5]) # 输出从第三个开始到第五个的字符。结果：noo  
print (str[2:]) # 输出从第三个开始的后的所有字符。结果：noob  
print (str \* 2) # 输出字符串两次。结果：RunoobRunoob  
print (str + "TEST") # 连接字符串。结果：RunoobTEST

与list不同的是，字符串不能通过索引访问去更改其中的字符。

word[1] = 'a'

这样的赋值是错误的

* + **字符串运算**

下表实例变量a值为字符串 "Hello"，b变量值为 "Python"：

|  |  |  |
| --- | --- | --- |
| **操作符** | **描述** | **实例** |
| + | 字符串连接 | a + b 输出结果： HelloPython |
| \* | 重复输出字符串 | a\*2 输出结果：HelloHello |
| [] | 通过索引获取字符串中字符 | a[1] 输出结果 **e** |
| [ : ] | 切片，截取字符串中的一部分 | a[1:4] 输出结果 **ell** |
| in | 如果字符串中包含给定的字符返回 True | **H in a** 输出结果 1 |
| not in | 如果字符串中不包含给定的字符返回 True | **M not in a** 输出结果 1 |

与 C 字符串不同的是，Python 字符串不能被改变。向一个索引位置赋值，比如word[0] = 'm'会导致错误。可以用切片 拼接的方式解决这个问题

a, b = 'Hello，John', 'Python'

var1 = a + ' ' + b # 结果是'Hello，John Python'

var3 = var1[:6] + 'World' # 这样，字符串就被改变了。

* + **字符串格式化**

我们在输出字符串的时候，如果想对输出的内容进行一些整理，比如把几段字符拼接起来（切片与拼接），或者把一段字符插入到另一段字符中间，就需要用到字符串的格式化输出。格式化字符串的输出可能会用到非常复杂的表达式，但最基本的用法是将一个值插入到一个有字符串格式符 %s 的字符串中。在 Python 中，字符串格式化使用与 C 中 sprintf 函数一样的语法。把一个数字加到文字后面输出，用str()函数，或者%d

注意区分：有引号的表示一段字符，没有引号的就是一个变量，这个变量可能是字符，也可能是数字，但一定要和%所表示的格式相一致。无论你有多少个值需要代入字符串中进行格式化，只需要在字符串中的合适位置用对应格式的%表示，然后在后面的括号中按顺序提供代入的值就可以了。占位的%和括号中的值在数量上必须相等，类型也要匹配。

str1 = 'My age is '

str2 = 'eighteen'

num = 18

print 'My age is' + num #**错误！！**字符和数字不能直接用+相加

print(str1+str2) # 字符串连接

print(str1+str(num)) # 用str()函数转成字符串

print('My age is %d' % num) # 字符串格式化，输出时%d会被%后面的值替换。这里，%d只能用来替换整数。如果你想格式化的数值是小数，要用%f

print ‘Price is %f’ % 4.99 # 输出Price is 4.990000

print ‘Price is %.2f’ % 4.99 # 保留两位小数%.2f，输出Price is 4.99

name, score = 'Lily', 95

print("%s's score is %d" %('Mike', score)) #字符串格式化。变量score。元组('Mike', score)

print("%s's score is %d" %(name, score)) # 字符串格式化。有变量。有元组

**('Mike', 87)**这种用()表示的一组数据在python中被称为元组（tuple），是python的一种基本数据结构。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| %c | %s | %d | %u | %o | %x和%X | %f | %e  和%E | %g和%G | %p | %r |
| 字符及其ASCII码 | 字符串 | 整数 | 无符号整型 | 无符号八进制数 | 无符号十六进制数（%X是大写） | 浮点数字，可指定小数点后的精度 | 用科学计数法格式化浮点数 | %f和%e的简写 | 用十六进制数格式化变量的地址 | 变量的原始数据raw data |

格式化操作符辅助指令:

|  |  |
| --- | --- |
| **符号** | **功能** |
| \* | 定义宽度或者小数点精度 |
| - | 用做左对齐 |
| + | 在正数前面显示加号( + ) |
| <sp> | 在正数前面显示空格 |
| # | 在八进制数前面显示零('0')，在十六进制前面显示'0x'或者'0X'(取决于用的是'x'还是'X') |
| 0 | 显示的数字前面填充'0'而不是默认的空格 |
| % | '%%'输出一个单一的'%' |
| (var) | 映射变量(字典参数) |
| m.n. | m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话) |

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000065&itemidx=1&sign=e9d904f034e4c73e5bca6af1ce269117*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000065&itemidx=1&sign=e9d904f034e4c73e5bca6af1ce269117)>

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000072&itemidx=1&sign=72151ce617b0c1003dbf9f499d66b946*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000072&itemidx=1&sign=72151ce617b0c1003dbf9f499d66b946)>

* + 多行字符串、raw字符串、编码问题

**多行字符串**：三个引号（‘’‘）或者（"""），可以方便地使用单引号和双引号，并且可以直接换行。多行字符串可以使用制表符TAB ( \t )，也可以使用换行符 [ \n ]。是所谓的WYSIWYG（所见即所得）格式。

para\_str = '''

"What's your name?" I asked.

"I'm Han Meimei."

'''

**raw 字符串**：如果一个字符串包含很多转义字符，会很麻烦。为了避免这种情况，我们可以在字符串前面加个前缀 r ，里面的字符就不需要转义了。

r'\(~\_~)/ \(~\_~)/'

**raw多行字符串：**但是r'...'表示法不能表示多行字符串，也不能表示包含'和 "的字符串。在多行字符串前面添加 r ，把这个多行字符串也变成一个raw多行字符串：

r'''Python is created by "Guido".  
It is free and easy to learn.  
Let's start learn Python in imooc!'''

来自 <[*http://www.imooc.com/code/3270*](http://www.imooc.com/code/3270)>

**Unicode字符串**

在Python3中，所有的字符串都是Unicode字符串。

如果中文字符串在Python环境下遇到 UnicodeDecodeError，这是因为.py文件保存的格式有问题。可以在第一行添加注释， 并选择UTF-8格式保存。

**# -\*- coding: utf-8 -\*- 告诉Python解释器，用UTF-8编码读取源代码**

Python2字符串的编码问题。计算机里，所有的字符都是二进制保存的。ASCII编码里，1byte=8bit=28=256=｛0，255｝，0 - 255映射成大小写英文字母、数字和一些符号。不用拉丁字母的那些语言，有各自的编码（中国GB2312），Unicode统一了所有语言文字的编码。

在Python2中，普通字符串是以8位ASCII码进行存储的，而Unicode字符串则存储为16位unicode字符串，这样能够表示更多的字符集。使用的语法是在字符串前面加上前缀 **u**，不加 u ，中文就不能正常显示。Unicode字符串除了多了一个 u 之外，与普通字符串没啥区别，转义字符和多行表示法仍然有效：

u'中文\n日文\n韩文' # 单行转义字符

u'''第一行 # 多行字符  
 第二行'''

ur'''Python的Unicode字符串支持"中文", # raw+多行  
 "日文",  
 "韩文"等多种语言'''

来自 <[*http://www.imooc.com/code/3271*](http://www.imooc.com/code/3271)>

* 1. Python 的字符串内建函数

Python 的字符串常用内建函数如下：

|  |  |
| --- | --- |
| **序号** | **方法及描述** |
| 1 | [capitalize()](http://www.runoob.com/python3/python3-string-capitalize.html)将字符串的第一个字符转换为大写 |
| 2 | [center(width, fillchar)](http://www.runoob.com/python3/python3-string-center.html)返回一个指定的宽度 width 居中的字符串，fillchar 为填充的字符，默认为空格。 |
| 3 | [count(str, beg= 0,end=len(string))](http://www.runoob.com/python3/python3-string-count.html)返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数 |
| 4 | [decode(encoding='UTF-8',errors='strict')](http://www.runoob.com/python3/python3-string-decode.html)使用指定编码来解码字符串。默认编码为字符串编码。 |
| 5 | [encode(encoding='UTF-8',errors='strict')](http://www.runoob.com/python3/python3-string-encode.html)以 encoding 指定的编码格式编码字符串，如果出错默认报一个ValueError 的异常，除非 errors 指定的是'ignore'或者'replace' |
| 6 | [endswith(suffix, beg=0, end=len(string))](http://www.runoob.com/python3/python3-string-endswith.html)检查字符串是否以 obj 结束，如果beg 或者 end 指定则检查指定的范围内是否以 obj 结束，如果是，返回 True,否则返回 False. |
| 7 | [expandtabs(tabsize=8)](http://www.runoob.com/python3/python3-string-expandtabs.html)把字符串 string 中的 tab 符号转为空格，tab 符号默认的空格数是 8 。 |
| 8 | [find(str, beg=0 end=len(string))](http://www.runoob.com/python3/python3-string-find.html)检测 str 是否包含在字符串中 中，如果 beg 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回-1 |
| 9 | [index(str, beg=0, end=len(string))](http://www.runoob.com/python3/python3-string-index.html)跟find()方法一样，只不过如果str不在字符串中会报一个异常. |
| 10 | [isalnum()](http://www.runoob.com/python3/python3-string-isalnum.html)如果字符串至少有一个字符并且所有字符都是字母或数字则返 回 True,否则返回 False |
| 11 | [isalpha()](http://www.runoob.com/python3/python3-string-isalpha.html)如果字符串至少有一个字符并且所有字符都是字母则返回 True, 否则返回 False |
| 12 | [isdigit()](http://www.runoob.com/python3/python3-string-isdigit.html)如果字符串只包含数字则返回 True 否则返回 False.. |
| 13 | [islower()](http://www.runoob.com/python3/python3-string-islower.html)如果字符串中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False |
| 14 | [isnumeric()](http://www.runoob.com/python3/python3-string-isnumeric.html)如果字符串中只包含数字字符，则返回 True，否则返回 False |
| 15 | [isspace()](http://www.runoob.com/python3/python3-string-isspace.html)如果字符串中只包含空格，则返回 True，否则返回 False. |
| 16 | [istitle()](http://www.runoob.com/python3/python3-string-istitle.html)如果字符串是标题化的(见 title())则返回 True，否则返回 False |
| 17 | [isupper()](http://www.runoob.com/python3/python3-string-isupper.html)如果字符串中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False |
| 18 | [join(seq)](http://www.runoob.com/python3/python3-string-join.html)以指定字符串作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串 |
| 19 | [len(string)](http://www.runoob.com/python3/python3-string-len.html)返回字符串长度 |
| 20 | [ljust(width[, fillchar])](http://www.runoob.com/python3/python3-string-ljust.html)  返回一个原字符串左对齐,并使用 fillchar 填充至长度 width 的新字符串，fillchar 默认为空格。 |
| 21 | [lower()](http://www.runoob.com/python3/python3-string-lower.html)  转换字符串中所有大写字符为小写. |
| 22 | [lstrip()](http://www.runoob.com/python3/python3-string-lstrip.html)  截掉字符串左边的空格 |
| 23 | [maketrans()](http://www.runoob.com/python3/python3-string-maketrans.html)  创建字符映射的转换表，对于接受两个参数的最简单的调用方式，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示转换的目标。 |
| 24 | [max(str)](http://www.runoob.com/python3/python3-string-max.html)  返回字符串 str 中最大的字母。 |
| 25 | [min(str)](http://www.runoob.com/python3/python3-string-min.html)  返回字符串 str 中最小的字母。 |
| 26 | [replace(old, new [, max])](http://www.runoob.com/python3/python3-string-replace.html)  把 将字符串中的 str1 替换成 str2,如果 max 指定，则替换不超过 max 次。 |
| 27 | [rfind(str, beg=0,end=len(string))](http://www.runoob.com/python3/python3-string-rfind.html)  类似于 find()函数，不过是从右边开始查找. |
| 28 | [rindex( str, beg=0, end=len(string))](http://www.runoob.com/python3/python3-string-rindex.html)  类似于 index()，不过是从右边开始. |
| 29 | [rjust(width,[, fillchar])](http://www.runoob.com/python3/python3-string-rjust.html)  返回一个原字符串右对齐,并使用fillchar(默认空格）填充至长度 width 的新字符串 |
| 30 | [rstrip()](http://www.runoob.com/python3/python3-string-rstrip.html)  删除字符串字符串末尾的空格. |
| 31 | [split(str="", num=string.count(str))](http://www.runoob.com/python3/python3-string-split.html)  num=string.count(str)) 以 str 为分隔符截取字符串，如果 num 有指定值，则仅截取 num 个子字符串 |
| 32 | [splitlines( num=string.count('\n'))](http://www.runoob.com/python3/python3-string-splitlines.html)  按照行分隔，返回一个包含各行作为元素的列表，如果 num 指定则仅切片 num 个行. |
| 33 | [startswith(str, beg=0,end=len(string))](http://www.runoob.com/python3/python3-string-startswith.html)  检查字符串是否是以 obj 开头，是则返回 True，否则返回 False。如果beg 和 end 指定值，则在指定范围内检查。 |
| 34 | [strip([chars])](http://www.runoob.com/python3/python3-string-strip.html)  在字符串上执行 lstrip()和 rstrip() |
| 35 | [swapcase()](http://www.runoob.com/python3/python3-string-swapcase.html)  将字符串中大写转换为小写，小写转换为大写 |
| 36 | [title()](http://www.runoob.com/python3/python3-string-title.html)  返回"标题化"的字符串,就是说所有单词都是以大写开始，其余字母均为小写(见 istitle()) |
| 37 | [translate(table, deletechars="")](http://www.runoob.com/python3/python3-string-translate.html)  根据 str 给出的表(包含 256 个字符)转换 string 的字符, 要过滤掉的字符放到 deletechars 参数中 |
| 38 | [upper()](http://www.runoob.com/python3/python3-string-upper.html)  转换字符串中的小写字母为大写 |
| 39 | [zfill (width)](http://www.runoob.com/python3/python3-string-zfill.html)  返回长度为 width 的字符串，原字符串右对齐，前面填充0 |
| 40 | [isdecimal()](http://www.runoob.com/python3/python3-string-isdecimal.html)  检查字符串是否只包含十进制字符，如果是返回 true，否则返回 false。 |

来自 <[*http://www.runoob.com/python3/python3-string.html*](http://www.runoob.com/python3/python3-string.html)>

字符串Split&Join

Thursday, August 18, 2016

20:44

* 1. 字符串的分割
  2. **默认分割**

sentence = 'I am an Englist sentence'

sentence.split()

split()默认是按照空白字符进行分割（空格、换行符\n，制表符\t），分割后的每一段都是一个新的字符串，最终返回这些字符串组成一个list。于是得到

['I', 'am', 'an', 'Englist', 'sentence']

原来字符串中的空白字符不再存在。

* 1. **指定分割符号**

split还可以指定分割的符号。

section = 'Hi. I am the one. Bye.'

print（section.split('.')）

通过指定分割符号为'.'，逗号作为分割符被去掉了，而空格仍然保留在它的位置上，得到['Hi', ' I am the one', ' Bye', '']

注意最后那个空字符串。每个'.'都会被作为分割符，即使它的后面没有其他字符，也会有一个空串被分割出来。例如

'aaa'.split('a')

将会得到['', '', '', '']，由四个空串组成的list。

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000127&itemidx=1&sign=4687ea41941ea68021df365f53a7ae86*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000127&itemidx=1&sign=4687ea41941ea68021df365f53a7ae86)>

* 1. list 连接成字符串

join把一个list中的所有字符串连接成一个字符串。

join不是list的方法，而是字符串的方法。首先你需要有一个字符串作为list中所有元素的连接符，然后再调用这个连接符的join方法，join的参数是被连接的list：

li = ['apple', 'pear', 'orange']

s = ';'

fruit = s.join(li)

print fruit # 得到结果'apple;pear;orange'分号把list中的几个字符串都连接了起来

指定连接符、join过程可以合为一步，如下：

';'.join(['apple', 'pear', 'orange'])

得到同样的结果。

连接符可以是多个字符，也可以用一个空字符串，**无缝连接**：

''.join(['hello', 'world'])

得到'helloworld'

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000130&itemidx=1&sign=cfa22615698a2d02b8233f7e9d002950*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000130&itemidx=1&sign=cfa22615698a2d02b8233f7e9d002950)>

**Python3列表 list**

Friday, September 23, 2016

12:43

实例：

list = [ 'abcd', 786 , 2.23, 'runoob', 70.2 ] #元素的类型可以不相同  
tinylist = [123, 'runoob']

print (list) # 输出完整列表：['abcd', 786, 2.23, 'runoob', 70.2]  
print (list[0]) # 输出列表第一个元素 ：abcd  
print (list[1:3]) # 从第二个开始输出到第三个元素：[786, 2.23]  
print (list[2:]) # 输出从第三个元素开始的所有元素：[2.23, 'runoob', 70.2]  
print (tinylist \* 2) # 输出两次列表：[123, 'runoob', 123, 'runoob']  
print (list + tinylist) # 连接列表：['abcd', 786, 2.23, 'runoob', 70.2, 123, 'runoob']

* 1. 简介

**包含Collection，容器数据类型containers**也是Python中基本的数据结构，包括字典 dict, 列表list, 集合set, 元组tuple。

**有序Sequence，序列数据类型**是Python中最基本的数据结构，包括字符串String，列表List，元组Tuple，等差数列Range。序列中的每个元素都分配一个数字 - 它的位置，或索引，第一个索引是0，第二个索引是1，依此类推。序列都可以进行的操作包括索引，切片，加，乘，检查成员。此外，Python已经内置确定序列的长度以及确定最大和最小的元素的方法。

Python有6个序列的内置类型，但最常见的是列表和元组。

* 1. **List**

List（列表） 是 Python 中使用最频繁的数据类型。list是一种有序的集合，list中的元素是按照顺序排列的，可以随时添加和删除其中的元素。列表可以完成大多数集合类的数据结构实现。由于Python是动态语言，所以**列表中元素的类型可以不相同**，它支持数字，字符串甚至可以包含列表（所谓嵌套）。

列表是最常用的Python数据类型。

l = ['meat', 'egg', 'fish', 'milk'] # 都是String类型

l = [365, 'everyday', 0.618, True] # 有int String double Boolean数据类型

列表用来处理一组有序项目的数据结构。想象一下你的购物清单、待办工作、手机通讯录等等，它们都可以看作是一个列表。

列表是写在方括号([])之间、用逗号分隔开的元素列表。List中的元素是可以改变的。

和字符串一样，列表索引从0开始。列表可以进行截取、组合等。列表被截取后返回一个包含所需元素的新列表。 列表截取的语法格式：变量[头下标:尾下标]，索引值以 0 为开始值，-1 为从末尾的开始位置。

加号（+）是列表连接运算符，星号（\*）是重复操作。

与Python字符串不一样的是，列表中的元素是可以改变的。

* 1. 语法
  + **创建list**

创建一个列表，只要把逗号分隔的不同的数据项使用方括号括起来即可。通常，我们会把list赋值给一个变量，这样，就可以通过变量来引用list：

classmates = ['Michael', 'Bob', 'Tracy']  
print( classmates) # 打印classmates变量的内容['Michael', 'Bob', 'Tracy']

empty\_list = [] # 创建空list，一个元素也没有的list

list2=list(range(5)) # 创建列表 [1, 2, 3, 4]

**嵌套List** 使用嵌套列表即在列表里创建其它列表，例如：

a = ['a', 'b', 'c']  
 n = [1, 2, 3]  
 x = [a, n] # [['a', 'b', 'c'], [1, 2, 3]]  
print( "x[0],",x[0] ,"x[0][1]", x[0][1] ) #结果 x[0] , ['a', 'b', 'c'] x[0][1] ，'b'

**转成List**

list()方法，将元组转换为列表。元组与列表是非常类似的，区别在于元组的元素值不能修改，元组是放在括号中，列表是放于方括号中。

aTuple = (123, 'Google', 'Runoob', 'Taobao')

list1 = list(aTuple)

print ("列表元素 : ", list1) # 列表元素 : [123, 'Google', 'Runoob', 'Taobao']

str="Hello World"

list2=list(str)

print ("列表元素 : ", list2) # 列表元素 : ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']

* + **索引**

使用下标索引来访问列表中的值，同样你也可以使用方括号的形式截取字符。list中的每个元素都对应一个递增的序号，与日常习惯不同，计算机中的计数通常都是从0开始。注意不要越界。

**用for in遍历**

l = [365, 'everyday', 0.618, True]

for i in l: # for in遍历

print(i, end=' ') # 只有空格，而不换行

print()

print l[1] # 索引。要访问l中的第1个元素365，只要用l[0]就可以了。输出'365'。注意，你不能访问一个不存在的元素，比如l[10]，程序就会报错，提示你index越界了。

print l[-1] # 索引。负数表示倒数第几位。

list2 = [1, 2, 3, 4, 5, 6, 7 ];

print ("list2[1:5]: ", list2[1:5]) # 结果：list2[1:5]: [2, 3, 4, 5]

**倒序访问list**

用 -1 这个索引来表示最后一个元素。类似地，倒数第二用 -2 表示，倒数第三用 -3 表示…

来自 <[*http://www.imooc.com/code/3357*](http://www.imooc.com/code/3357)>

* + **切片**

切片操作符是在[]内提供一对可选数字，用:分割。冒号前的数表示切片的开始位置，冒号后的数字表示切片到哪里结束。注意，开始位置包含在切片中，而结束位置不包括。

print l[1:3] # 取得第二个、第三个值，去除第三个，得到的结果是['everyday', 0.618]

print l[:3] # 如果不指定第一个数，切片就从列表第一个元素开始，[365, 'everyday', 0.618]

print l[1:] # 如果不指定第二个数，就一直到最后一个元素结束，[ 'everyday', 0.618, True]。

print l[:] # 都不指定，则返回整个列表，[365, 'everyday', 0.618, True]

print l[1:-1] # 也可以使用负数。得到['everyday', 0.618]

l[0] = 123 # 修改list中的某一个元素，只需要直接给那个元素赋值就可以了：

l.append(1024) # 用append方法增加元素。1024被添加到了l，成为最后一个元素。

del l[0] # 用del删除list中的某一个元素。其他元素的序号相应提前。

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000119&itemidx=1&sign=428cc1aece74a4e44d4884da28d9c285*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000119&itemidx=1&sign=428cc1aece74a4e44d4884da28d9c285)>

**list.copy()** 函数用于**复制列表**，类似于 a[:]。返回复制后的新列表。

list1 = ['Google', 'Runoob', 'Taobao', 'Baidu']

list2 = list1.copy()

print ("list2 列表: ", list2) # list2 列表: ['Google', 'Runoob', 'Taobao', 'Baidu']

* + **替换List元素**

你可以对列表的数据项进行修改或更新：

list = ['Google', 'Runoob', 1997, 2000]

list[2] = 2001  
print ("更新后的第三个元素为 : ", list[2]) # 结果：更新后的第三个元素为 : 2001

来自 <[*http://www.imooc.com/code/3360*](http://www.imooc.com/code/3360)>

* + **增加List元素**

**append()**默认把新的元素添加到 list 的尾部。

L = ['Adam', 'Lisa', 'Bart']  
 L.append('Paul')  
 print L # 结果 ['Adam', 'Lisa', 'Bart', 'Paul']

**insert()方法**，插入新元素，顺序后延。两个参数，第一个参数是索引号，第二个参数是待添加的新元素：

L = ['Adam', 'Lisa', 'Bart']  
 L.insert(0, 'Paul') #'Paul'将被添加到索引为 0 的位置上（也就是第一个），而原来的元素都向后移动一位。结果['Paul', 'Adam', 'Lisa', 'Bart']

来自 <[*http://www.imooc.com/code/3358*](http://www.imooc.com/code/3358)>

**+组合、拼接 \*重复**

squares = [1, 4, 9, 16, 25]  
list = squares + [36, 49, 64, 81, 100] # 结果：[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

list2 = squares \* 2 # 结果： [1, 4, 9, 16, 25, 1, 4, 9, 16, 25]

**list.extend()** 函数用于在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）。该方法没有返回值，但会在已存在的列表中添加新的列表内容。

list1 = ['Google', 'Runoob', 'Taobao']

list2=list(range(5)) # 创建列表 [1, 2, 3, 4]

list1.extend(list2) # 扩展列表

print ("扩展后的列表：", list1) # 扩展后的列表： ['Google', 'Runoob', 'Taobao', 0, 1, 2, 3, 4]

* + **删除List元素**

**pop()**方法，默认删掉list的最后一个元素，也可以用索引指定删除哪个元素，并且它还返回这个元素，打印出来。Paul同学刚来几天又要转走了。如果Paul同学排在最后一个，我们可以用list的pop()方法删除：

>>> L = ['Adam', 'Lisa', 'Bart', 'Paul']  
>>> L.pop(2)  
'Bart'  
>>> print L

['Adam', 'Lisa', 'Paul']

还可以使用 **del 语句**来删除列表的的元素，如下实例：

list = ['Google', 'Runoob', 1997, 2000]

del list[2]  
print (list) # 结果： ['Google', 'Runoob', 2000]

**list.remove()** 函数用于移除列表中某个值的第一个匹配项。该方法没有返回值但是会移除两种中的某个值的第一个匹配项。

list1 = ['Google', 'Runoob', 'Taobao', 'Baidu']

list1.remove('Taobao')

print ("列表现在为 : ", list1) # 列表现在为 : ['Google', 'Runoob', 'Baidu']

list1.remove('Baidu')

print ("列表现在为 : ", list1) # 列表现在为 : ['Google', 'Runoob']

**list.clear()** 函数用于清空列表，类似于 del a[:]

list1 = ['Google', 'Runoob', 'Taobao', 'Baidu']

list1.clear()

print ("列表清空后 : ", list1) # 列表清空后 : []

* + **排序**

**list.reverse()** 函数。该方法没有参数、返回值，但是会对列表的元素进行**反向排列**。

list1 = ['Google', 'Runoob', 'Taobao', 'Baidu']

list1.reverse()

print ("列表反转后: ", list1) # 列表反转后: ['Baidu', 'Taobao', 'Runoob', 'Google']

**list.sort( [func] )** 函数用于对原列表进行**排序**，如果指定参数，则使用比较函数指定的比较函数。func 可选参数, 如果指定了该参数会使用该参数的方法进行排序。该方法没有返回值，但是会对列表的对象进行排序。

list1 = ['Google', 'Runoob', 'Taobao', 'Baidu']

list1.sort()

print ("列表排序后 : ", list1) # 列表排序后 : ['Baidu', 'Google', 'Runoob', 'Taobao']

* + **运算**

**in运算**

3 in [1, 2, 3] #元素是否存在于列表中。结果： True

**list.index()** 函数用于从列表中找出某个值第一个匹配项的索引位置。该方法返回查找对象的索引位置，如果没有找到对象则抛出异常。

list1 = ['Google', 'Runoob', 'Taobao']

print ('Runoob 索引值为', list1.index('Runoob')) # Runoob 索引值为 1

print ('Taobao 索引值为', list1.index('Taobao')) # Taobao 索引值为 2

**len()方法**，返回列表元素个数（长度）

list1 = ['Google', 'Runoob', 'Taobao']

print (len(list1))

**list.count()** 函数用于统计某个元素在列表中出现的次数。返回元素在列表中出现的次数。

aList = [123, 'Google', 'Runoob', 'Taobao', 123];

print ("123 元素个数 : ", aList.count(123)) # 123 元素个数 : 2

print ("Runoob 元素个数 : ", aList.count('Runoob')) # Runoob 元素个数 : 1

**max()方法**，返回列表元素最大值；**min()方法**，返回列表元素最小值

list1, list2 = ['Google', 'Runoob', 'Taobao'], [456, 700, 200]

print ("list1 最大元素值 : ", max(list1)) # list1 最大元素值 : Taobao

print ("list2 最大元素值 : ", max(list2)) # list2 最大元素值 : 700

print ("list1 最小元素值 : ", min(list1)) # list1 最小元素值 : Google

print ("list2 最小元素值 : ", min(list2)) # list2 最小元素值 : 200

来自 <[*http://www.runoob.com/python3/python3-list.html*](http://www.runoob.com/python3/python3-list.html)>

Python3 元组 Tuple

Friday, September 23, 2016

12:44

tuple = ( 'abcd', 786 , 2.23, 'runoob', 70.2 )  
tinytuple = (123, 'runoob')

print (tuple) # 输出完整元组 ('abcd', 786, 2.23, 'runoob', 70.2)  
print (tuple[0]) # 输出元组的第一个元素 abcd  
print (tuple[1:3]) # 输出从第二个元素开始到第三个元素 (786, 2.23)  
print (tuple[2:]) # 输出从第三个元素开始的所有元素 (2.23, 'runoob', 70.2)  
print (tinytuple \* 2) # 输出两次元组 (123, 'runoob', 123, 'runoob')  
print (tuple + tinytuple) # 连接元组 ('abcd', 786, 2.23, 'runoob', 70.2, 123, 'runoob')

* 1. 简介

String、list和tuple都属于sequence（序列）。

Python 的元组（tuple）与列表类似，元组中的元素类型也可以不相同，元素可以索引、截取、组合，不同之处在于元组使用小括号。与字符串一样，元组的元素不能修改，没有 append()、insert()、pop()等方法。元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。

元组、字符串、列表类似，可以被索引且下标索引从0开始，-1 为从末尾开始的位置。也可以进行截取（看上面，这里不再赘述）、组合。其实，可以把字符串看作一种特殊的元组。

* 1. **创建tuple**

tup1 = ('Google', 'Runoob', 1997, 2000); # 可以是任何数据类型，一个元组里面可以混合数据类型  
构造包含 0 个或 1 个元素的元组比较特殊，所以有一些额外的语法规则：

tup1 = () # 空元组  
tup2 = (20,) # 元组中只包含一个元素时，需要在元素后添加逗号

**为什么**呢？因为用()定义单元素的tuple有歧义，()既可以表示tuple，又可以作为括号表示运算时的优先级，结果 tup2 = (20) 被Python解释器计算出结果 20，等于赋值语句 tup2 = 20。Python在打印单元素tuple时，也自动添加了一个“,”，为了更明确地告诉你这是一个tuple。多元素 tuple 加不加这个额外的“,”效果是一样的。

来自 <[*http://www.imooc.com/code/3361*](http://www.imooc.com/code/3361)> 来自 <[*http://www.imooc.com/code/3362*](http://www.imooc.com/code/3362)>

**“可变”的tuple**

虽然tuple的元素不可改变、一旦创建就不能修改，但它可以包含可变的对象，比如list列表。 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，但这个list本身是可变的！那么，如果要创建一个内容也不变的tuple，就必须保证tuple的每一个元素本身也不能变，比如t=('a','b',('c','d'))。

t = ('a', 'b', ['A', 'B']) #t 有 3 个元素：**'a'，'b'**和一个list：**['A', 'B']**。

# 通过 t[2] 拿到list对象，然后，我们把list的两个元素改一改，然后tuple的内容就变了

L = t[2]

L[0] = 'X'  
L[1] = 'Y' #结果t就变为 ('a', 'b', ['X', 'Y'])

**再议不可变对象**

上面我们讲了，str是不变对象，而list是可变对象。

虽然字符串有个replace()方法，也确实变出了'Abc'，但变量a最后仍是'abc'，应该怎么理解呢？

我们先把代码改成下面这样：

a = 'abc'  
b = a.replace('a', 'A') #  
print（ a，b） #'Abc' 'abc'  
a是变量，而'abc'才是字符串对象！我们经常说，对象a的内容是'abc'，但其实是，a本身是一个变量，它指向的对象的内容才是'abc'：

当我们调用a.replace('a', 'A')时，实际上调用方法replace是作用在字符串对象'abc'上的，而这个方法虽然名字叫replace，但却没有改变字符串'abc'的内容。相反，replace方法创建了一个新字符串'Abc'并返回，如果我们用变量b指向该新字符串，就容易理解了，变量a仍指向原有的字符串'abc'，但变量b却指向新字符串'Abc'了：

所以，对于不变对象来说，调用对象自身的任意方法，也不会改变该对象自身的内容。相反，这些方法会创建新的对象并返回，这样，就保证了不可变对象本身永远是不可变的。

来自 <[*http://www.liaoxuefeng.com*](http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/00143167793538255adf33371774853a0ef943280573f4d000)> 来自 <[*http://www.imooc.com/code/3363*](http://www.imooc.com/code/3363)>

**索引访问**

元组可以使用下标索引来访问元组中的值，如下实例:

#!/usr/bin/python3

tup1 = ('Google', 'Runoob', 1997, 2000)  
tup2 = (1, 2, 3, 4, 5, 6, 7 )

print ("tup1[0]: ", tup1[0])  
print ("tup2[1:5]: ", tup2[1:5])

以上实例输出结果：

tup1[0]: Google  
tup2[1:5]: (2, 3, 4, 5)

|  |  |  |
| --- | --- | --- |
| for x in (1, 2, 3): print x, | 1 2 3 | 迭代 |

**索引截取（切片）**

因为元组也是一个序列，所以我们可以访问元组中的指定位置的元素，也可以截取索引中的一段元素，如下所示：

元组：

L = ('Google', 'Taobao', 'Runoob')

|  |  |  |
| --- | --- | --- |
| **Python 表达式** | **结果** | **描述** |
| L[2] | 'Runoob!' | 读取第三个元素 |
| L[-2] | 'Taobao' | 反向读取；读取倒数第二个元素 |
| L[1:] | ('Taobao', 'Runoob!') | 截取元素，从第二个开始后的所有元素。 |

运行实例如下：

>>> L = ('Google', 'Taobao', 'Runoob')  
>>> L[2]  
'Runoob'  
>>> L[-2]  
'Taobao'  
>>> L[1:]  
('Taobao', 'Runoob')

**元组拼接**

元组中的元素值是不允许修改的，但我们可以对元组进行连接组合，如下实例:

#!/usr/bin/python3

tup1 = (12, 34.56);  
tup2 = ('abc', 'xyz')

# 以下修改元组元素操作是非法的。  
# tup1[0] = 100

# 创建一个新的元组  
tup3 = tup1 + tup2;  
print (tup3)

以上实例输出结果：

(12, 34.56, 'abc', 'xyz')

**元组运算符**

与字符串一样，元组之间可以使用 + 号和 \* 号进行运算。这就意味着他们可以组合和复制，运算后会生成一个新的元组。

|  |  |  |
| --- | --- | --- |
| ['Hi!'] \* 4 | ['Hi!', 'Hi!', 'Hi!', 'Hi!'] | 复制 |

**删除整个元组**

元组中的元素值是不允许删除的，但我们可以使用del语句来删除整个元组，如下实例:

#!/usr/bin/python3

tup = ('Google', 'Runoob', 1997, 2000)

print (tup)  
del tup;  
print ("删除后的元组 tup : ")  
print (tup)

以上实例元组被删除后，输出变量会有异常信息，输出如下所示：

删除后的元组 tup :   
Traceback (most recent call last):  
 File "test.py", line 8, in <module>  
 print (tup)  
NameError: name 'tup' is not defined

|  |  |  |
| --- | --- | --- |
| **Python 表达式** | **结果** | **描述** |
| len((1, 2, 3)) | 3 | 计算元素个数 |
| 3 in (1, 2, 3) | True | 元素是否存在 |

**元组内置函数**

Python元组包含了以下内置函数

|  |  |  |
| --- | --- | --- |
| 2 | max(tuple)  返回元组中元素最大值。 | >>> tuple2 = ('5', '4', '8') >>> max(tuple2) '8' >>> |
| 3 | min(tuple)  返回元组中元素最小值。 | >>> tuple2 = ('5', '4', '8') >>> min(tuple2) '4' >>> |
| 4 | tuple(seq)  将列表转换为元组。 | >>> list1= ['Google', 'Taobao', 'Runoob', 'Baidu'] >>> tuple1=tuple(list1) >>> tuple1 ('Google', 'Taobao', 'Runoob', 'Baidu') |

来自 <[*http://www.runoob.com/python3/python3-tuple.html*](http://www.runoob.com/python3/python3-tuple.html)>

**字典Dictionary与集合Set**

Friday, September 23, 2016

12:44

字典（dictionary）是Python中另一个非常有用的内置数据类型。

列表是有序的对象结合，字典是无序的对象集合。两者都可变，且可以存储任意类型的对象。两者之间的区别在于：字典当中的元素是通过键来存取的，而不是通过偏移存取。

字典是一种**映射数据类型(mapping)**，用"{ }"标识，是一个无序的**键(key) : 值(value)**对集合。每个键值(key=>value)对用冒号(**:**)分割，每个对之间用逗号(**,**)分割，整个字典包括在花括号(**{})**中 ,格式如下所示：

d = {key1 : value1, key2 : value2 }

键(key)必须使用不可变类型。在同一个字典中，键(key)必须是唯一、不重复的。最常用的key是字符串。

**创建字典**

dict = {}  
dict['one'] = "1 - 菜鸟教程"  
dict[2] = "2 - 菜鸟工具"

tinydict = {'name': 'runoob','code':1, 'site': '[www.runoob.com](http://www.runoob.com)'}

print (dict['one']) # 输出键为 'one' 的值 1 - 菜鸟教程  
print (dict[2]) # 输出键为 2 的值 2 - 菜鸟工具  
print (tinydict) # 输出完整的字典{'name': 'runoob', 'site': '[www.runoob.com](http://www.runoob.com)', 'code': 1}  
print (tinydict.keys()) # 输出所有键 dict\_keys(['name', 'site', 'code'])  
print (tinydict.values()) # 输出所有值 dict\_values(['runoob', '[www.runoob.com](http://www.runoob.com)', 1])

构造函数 dict() 可以直接从键值对序列中构建字典如下：

>>> dict([('Runoob', 1), ('Google', 2), ('Taobao', 3)])  
{'Taobao': 3, 'Runoob': 1, 'Google': 2}

>>> {x: x\*\*2 for x in (2, 4, 6)}  
{2: 4, 4: 16, 6: 36}

>>> dict(Runoob=1, Google=2, Taobao=3)  
{'Taobao': 3, 'Runoob': 1, 'Google': 2}

创建空字典

nulldict = {}

一个简单的字典实例：

dict = {'Alice': '2341', 'Beth': '9102', 984: '3258'}

print(dict.keys()) 查看所有的键

print(dict.values()) 查看所有的值

print(dict.items()) 查看所有的键值对

for k, v in dic.items(): 遍历所有的键值对

print("dict key:value" %(k,v))

用zip可以将两个list合并为dictionary

keys = ["a","b","c","d","e"]

values = [1,2,3,4,5]

for k, v in zip(keys, values):

print((k,v))

**访问字典里的值**

把相应的键放入熟悉的方括弧，如下实例:

dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}

print ("dict['Name']: ", dict['Name'])  
print ("dict['Age']: ", dict['Age'])

以上实例输出结果：

dict['Name']: Runoob  
dict['Age']: 7

不能越界，如果用字典里没有的键访问数据，会出现错误。

**修改字典**

向字典添加新内容的方法是增加新的键/值对，修改或删除已有键/值对如下实例:

#!/usr/bin/python3

dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}

dict['Age'] = 8; # 更新 Age  
dict['School'] = "菜鸟教程" # 添加信息

print ("dict['Age']: ", dict['Age'])  
print ("dict['School']: ", dict['School'])

以上实例输出结果：

dict['Age']: 8  
dict['School']: 菜鸟教程

**删除字典元素**

能删单一的元素也能清空字典，清空只需一项操作。

显示删除一个字典用del命令，如下实例：

dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}

del dict['Name'] # 删除键 'Name'  
dict.clear() # 删除字典  
del dict # 删除字典

print ("dict['Age']: ", dict['Age'])  
print ("dict['School']: ", dict['School'])

但这会引发一个异常，因为用执行 del 操作后字典不再存在：

Traceback (most recent call last):  
 File "test.py", line 9, in <module>  
 print ("dict['Age']: ", dict['Age'])  
TypeError: 'type' object is not subscriptable

**注：**del() 方法后面也会讨论。

**字典键的特性**

字典值可以没有限制地取任何python对象，既可以是标准的对象，也可以是用户定义的，但键不行。

两个重要的点需要记住：

1）不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住，如下实例：

#!/usr/bin/python3

dict = {'Name': 'Runoob', 'Age': 7, 'Name': '小菜鸟'}

print ("dict['Name']: ", dict['Name'])

以上实例输出结果：

dict['Name']: 小菜鸟

2）键必须不可变，所以可以用数字，字符串或元组充当，而用列表就不行，如下实例：

#!/usr/bin/python3

dict = {['Name']: 'Runoob', 'Age': 7}

print ("dict['Name']: ", dict['Name'])

以上实例输出结果：

Traceback (most recent call last):  
 File "test.py", line 3, in <module>  
 dict = {['Name']: 'Runoob', 'Age': 7}  
TypeError: unhashable type: 'list'

**字典内置函数&方法**

Python字典包含了以下内置函数：

|  |  |  |
| --- | --- | --- |
| **序号** | **函数及描述** | **实例** |
| 1 | len(dict)  计算字典元素个数，即键的总数。 | >>> dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} >>> len(dict) 3 |
| 2 | str(dict)  输出字典以可打印的字符串表示。 | >>> dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} >>> str(dict) "{'Name': 'Runoob', 'Class': 'First', 'Age': 7}" |
| 3 | type(variable)  返回输入的变量类型，如果变量是字典就返回字典类型。 | >>> dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} >>> type(dict) <class 'dict'> |

Python字典包含了以下内置方法：

|  |  |
| --- | --- |
| **序号** | **函数及描述** |
| 1 | [radiansdict.clear()](http://www.runoob.com/python3/python3-att-dictionary-clear.html)  删除字典内所有元素 |
| 2 | [radiansdict.copy()](http://www.runoob.com/python3/python3-att-dictionary-copy.html)  返回一个字典的浅复制 |
| 3 | [radiansdict.fromkeys()](http://www.runoob.com/python3/python3-att-dictionary-fromkeys.html)  创建一个新字典，以序列seq中元素做字典的键，val为字典所有键对应的初始值 |
| 4 | [radiansdict.get(key, default=None)](http://www.runoob.com/python3/python3-att-dictionary-get.html)  返回指定键的值，如果值不在字典中返回default值 |
| 5 | [key in dict](http://www.runoob.com/python3/python3-att-dictionary-in.html)  如果键在字典dict里返回true，否则返回false |
| 6 | [radiansdict.items()](http://www.runoob.com/python3/python3-att-dictionary-items.html)  以列表返回可遍历的(键, 值) 元组数组 |
| 7 | [radiansdict.keys()](http://www.runoob.com/python3/python3-att-dictionary-keys.html)  以列表返回一个字典所有的键 |
| 8 | [radiansdict.setdefault(key, default=None)](http://www.runoob.com/python3/python3-att-dictionary-setdefault.html)  和get()类似, 但如果键不存在于字典中，将会添加键并将值设为default |
| 9 | [radiansdict.update(dict2)](http://www.runoob.com/python3/python3-att-dictionary-update.html)  把字典dict2的键/值对更新到dict里 |
| 10 | [radiansdict.values()](http://www.runoob.com/python3/python3-att-dictionary-values.html)  以列表返回字典中的所有值 |

来自 <[*http://www.runoob.com/python3/python3-dictionary.html*](http://www.runoob.com/python3/python3-dictionary.html)>

* + **Set（集合）**

集合（set）是一个**无序不重复**元素的集合。基本功能是进行成员关系测试和删除重复元素。set和dict类似，也是一组key的集合，但不存储value。set的key，也是不可以放入可变对象（比如list），因为无法判断两个可变对象是否相等，也就无法保证set内部“不会有重复元素”。

可以使用大括号{} 创建集合。

student = {'Tom', 'Jim', 'Mary', 'Tom', 'Jack', 'Rose'} #重复元素在set中自动被过滤

print(student) # 输出集合 {'Jack', 'Rose', 'Mary', 'Jim', 'Tom'}

创建空集合：必须用 set() 而不是 { }，因为 { } 是用来创建一个空字典

nullset = set()

# 成员测试  
if('Rose' in student) :  
 print('Rose 在集合中') #Rose 在集合中

set()函数创建集合，**里面是一个序列，或者是一个字符串**

s = set([1, 2, 3]) # {1, 2, 3}

s.add(4) # 通过add(key)方法可以添加元素到set中，同一个元素可以重复添加，但不会有效果：{1, 2, 3, 4 }

s.remove(4) #通过remove(key)方法可以删除元素： {1, 2, 3}

正因为set可以变化，所以不可散列（被固定引用），不能作为字典dictionary的key，也不能作为别的集合set的key

构造不可改变的Frozenset

fset = frozenset([1, 2, 4])

a = frozenset('Jack', 'Rose', 'Mary', 'Jim', 'Tom')

# set可以进行集合运算  
a = set('abracadabra')  
b = set('alacazam')

print(a) # {'r', 'b', 'a', 'c', 'd'}

print(a - b) # a和b的差集 {'r', 'b', 'd'}

print(a | b) # a和b的并集 {'a', 'l', 'z', 'b', 'm', 'd', 'r', 'c'}

print(a & b) # a和b的交集 {'a', 'c'}

print(a ^ b) # a和b中不同时存在的元素 {'l', 'z', 'b', 'm', 'd', 'r'}

Tuesday, September 20, 2016

18:00

**数据类型转换**

python是动态语言，定义变量时不需要限定类型。变量会根据赋给它的值，自动决定它的类型。如果重新赋值时，改变值的类型，于是也就改变了变量的类型。

虽然类型可以随意改变，但当你对一个特定类型的变量进行操作时，如果这个操作与它的数据类型不匹配，就会产生错误。这种情况下，python提供了一些方法对数值进行类型转换。并不是所有的值都能做类型转换，比如int('abc')同样会报错，python没办法把它转成一个整数。

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000076&itemidx=1&sign=1a3aba84dc60daf796b13c72c8c8b5b4*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000076&itemidx=1&sign=1a3aba84dc60daf796b13c72c8c8b5b4)>

* + **数据类型转换**

有时候，我们需要对数据内置的类型进行转换，数据类型的转换，你只需要将数据类型作为函数名即可。

以下几个内置的函数可以执行数据类型之间的转换。这些函数返回一个新的对象，表示转换的值。

|  |  |
| --- | --- |
| **函数** | **描述** |
| int(x [,base]) | 将x转换为一个整数 |
| float(x) | 将x转换到一个浮点数 |
| complex(real [,imag]) | 创建一个复数 |
| str(x) | 将对象 x 转换为字符串 |
| repr(x) | 将对象 x 转换为表达式字符串 |
| eval(str) | 用来计算在字符串中的有效Python表达式,并返回一个对象 |
| tuple(s) | 将序列 s 转换为一个元组 |
| list(s) | 将序列 s 转换为一个列表 |
| set(s) | 转换为可变集合 |
| dict(d) | 创建一个字典。d 必须是一个序列 (key,value)元组。 |
| frozenset(s) | 转换为不可变集合 |
| chr(x) | 将一个整数转换为一个字符 |
| unichr(x) | 将一个整数转换为Unicode字符 |
| ord(x) | 将一个字符转换为它的整数值 |
| hex(x) | 将一个整数转换为一个十六进制字符串 |
| oct(x) | 将一个整数转换为一个八进制字符串 |

来自 <[*http://www.runoob.com/python3/python3-data-type.html*](http://www.runoob.com/python3/python3-data-type.html)>

**bool转换**

因为在python中，以下数值会被认为是False：

为0的数字，包括0，0.0

空字符串，包括''，""

None空值

空集合，包括()，[]，{}

None是python中的一个特殊值，表示什么都没有，它和0、空字符、False、空集合都不一样。关于集合，我们后面的课程再说。

其他的值都认为是True。

在if、while等条件判断语句里，判断条件会自动进行一次bool的转换。比如

   a = '123'

   if a:

       print 'this is not a blank string'

这在编程中是很常见的一种写法。效果等同于

if bool(a)

或者

if a != ''

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000079&itemidx=1&sign=4946af6e5e78180bcb4648610ac67ccc*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000079&itemidx=1&sign=4946af6e5e78180bcb4648610ac67ccc)>

**Python3 运算符**

Thursday, September 22, 2016

09:41

**4 +5 = 9** 。 例子中，**4** 和 **5** 被称为**操作数**，"**+**" 称为运算符。

|  |  |  |
| --- | --- | --- |
| 算术运算符 | +加  -减  \*乘  /除  //取整除。返回商的整数部分  %取模。返回除法的余数  \*\*幂。返回x的y次幂 | a,b=10,21 a+b=31  -a=10 a-b=-11  a\*b=210  b/a=2.1    b//a=2  b%a=1  a\*\*b=1021 |
| 比较运算符 | == !=  > <  >= <= | 得到的是布尔值True/False |
| 赋值运算符 | = 简单的赋值运算符  +=加法赋值运算符  -=减法赋值运算符  \*=乘法赋值运算符  /=除法赋值运算符  %=取模赋值运算符  \*\*=幂赋值运算符  //=取整除赋值运算符 | c = a + b 将 a + b 的运算结果赋值为 c  c += a 等效于 c = c + a  c -= a 等效于 c = c - a  c \*= a 等效于 c = c \* a  c /= a 等效于 c = c / a  c %= a 等效于 c = c % a  c \*\*= a 等效于 c = c \*\* a  c //= a 等效于 c = c // a |
| 逻辑运算符 | and 布尔"与" - x and y，如果 x 为 False，x and y 返回 False，否则它返回 y 的计算值。  or 布尔"或" - x or y，如果 x 是 True，它返回 True，否则它返回 y 的计算值。  not 布尔"非" - not x，如果 x 为 True，返回 False 。如果 x 为 False，它返回 True。 | a,b=10,20  (a and b) 返回 20。布尔值True  (a or b) 返回 10。布尔值True  not(a and b) 返回 False |
| 位运算符 | |  | | --- | | & 按位与运算符：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0 | | | 按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1。 | | ^ 按位异或运算符：当两对应的二进位相异时，结果为1 | | ~ 按位取反运算符：对数据的每个二进制位取反,即把1变为0,把0变为1 | | << 左移动运算符：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。 | | >> 右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数 | | a,b=60,13；(a & b) 输出结果 12 ，二进制解释： 0000 1100  (a | b) 输出结果 61 ，二进制解释： 0011 1101  (a ^ b) 输出结果 49 ，二进制解释： 0011 0001  (~a ) 输出结果 -61 ，二进制解释： 1100 0011， 在一个有符号二进制数的补码形式。  a << 2 输出结果 240 ，二进制解释： 1111 0000  a >> 2 输出结果 15 ，二进制解释： 0000 1111 |
| 成员运算符 | in 如果在指定的序列中找到值返回 True，否则返回 False。序列包括字符串，列表或元组。  not in 如果在指定的序列中没有找到值返回 True，否则返回 False。 | a,b=10,20  mlist=[1,2,3,4,5]  a in mlist 这个布尔值是False |
| 身份运算符 | is是判断两个标识符是不是引用自一个对象    is not是判断两个标识符是不是引用自不同对象 | x is y, 如果 id(x) 等于 id(y) , **is** 返回结果 1  x is not y, 如果 id(x) 不等于 id(y). **is not** 返回结果 1 |
| 优先级 | 不需记忆，一般用括号辅助。以下表格列出了从最高到最低优先级的所有运算符：   |  |  | | --- | --- | | **运算符** | **描述** | | \*\* | 指数 (最高优先级) | | ~ + - | 按位翻转, 一元加号和减号 (最后两个的方法名为 +@ 和 -@) | | \* / % // | 乘，除，取模和取整除 | | + - | 加法减法 | | >> << | 右移，左移运算符 | | & | 位 'AND' | | ^ | | 位运算符 | | <= < > >= | 比较运算符 | | <> == != | 等于运算符 | | = %= /= //= -= += \*= \*\*= | 赋值运算符 | | is is not | 身份运算符 | | in not in | 成员运算符 | | not or and | 逻辑运算符 | |  |

控制与循环

2016年8月11日

20:33

* 1. 变量

变量3个属性：**数据类型、变量名**以及**变量值**。根据所需要保存的数据的格式，将其保存在指定类型的变量空

* 1. **Java标识符**

int age; //声明变量的语法：数据类型 变量名;

age=18; //变量赋值的语法：变量名=值；

* 1. if——分支语句，依据条件选择执行

程序顺序往下执行遇到if语句的时候，会去判断它所带条件的真假。如果为True，就会去执行接下来的内容；如果为False，就跳过。

if在编程语言中被称为“控制流语句”，用来控制程序的执行顺序。还有其他的控制流语句，后面会学。

计算机生成了可选文字:
False 
True 
if 
task 

语法为：

   if 条件:

       选择执行的语句

例句：

age = 20  
if age >= 18: # 冒号不能少，必须是英文字符，表示代码块开始  
 print 'your age is', age # 统一的缩进，一般用4个空格  
 print 'adult'  
print 'END'

计算机生成了可选文字:
Ture 
False 
els 
END NEXT 

计算机生成了可选文字:
Ture 
False 
True 
False 
—n elif... 
True 
(else) 
False 
END/NEXT 

if, elif, else可组成一个整体的条件语句。

if，是必须有的；

elif，意为else if。elif后面需要有一个逻辑判断语句。可以没有，也可以有很多个，每个elif条件不满足时会进入下一个elif判断；

else，可以没有，如果有的话只能有一个，必须在条件语句的最后。

例1，if…else

if a == 1:

   print 'right'

else # 当if后面的条件语句不满足时，与之相对应的else中的代码块将被执行

   print 'wrong'

 例2，if…elif

if a == 1:

   print 'one'

elif a == 2: #当if条件不满足时，再去判断elif的条件，如果满足则执行其中的代码块。

   print 'two'

例3，if…elif…else

if a == 1:

   print 'one'

elif a == 2:

   print 'two'

elif a == 3:

   print 'three'

else:

   print 'too many'

**注意:**Python代码的缩进规则，，否则很容易造成因为缩进引起的语法错误。具有相同缩进的代码被视为代码块，上面的3，4行 print 语句就构成一个代码块（但不包括第5行的print）。如果 if 语句判断为 True，就会执行这个代码块。缩进请严格按照Python的习惯写法：特别留意缩进，4个空格，不要使用Tab，更不要混合Tab和空格，并且退出缩进需要多敲一行回车。

来自 <[*http://www.imooc.com/code/3421*](http://www.imooc.com/code/3421)>

**if的嵌套**

和for循环一样，if也可以嵌套使用，即在一个if/elif/else的内部，再使用if。这有点类似于电路的串联。 假设需要这样一个程序：我们先向程序输入一个值x，再输入一个值y。(x,y)表示一个点的坐标。程序要告诉我们这个点处在坐标系的哪一个象限。你可以分别写4个if，也可以用if的嵌套：

if y >= 0: #条件1。

   if x >= 0: #条件2

       print 1 # 条件1为True，条件2为True时，执行语句1；

   else:

       print 2 # 条件1为True，条件2为False时，执行语句2；

else:

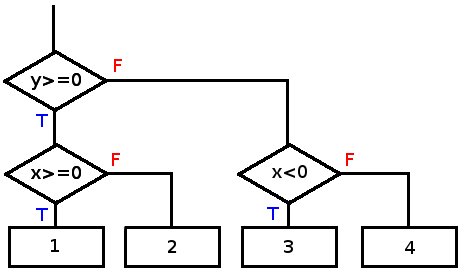
   if x < 0: #条件2

       print 3 # 条件1为False，条件2为True时，执行语句3；

   else:

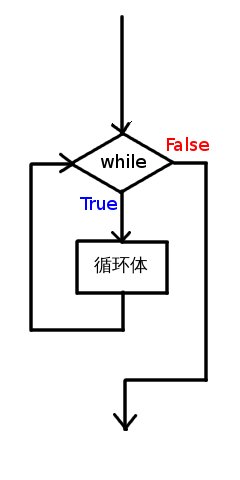
       print 4 # 条件1为False，条件2为False时，执行语句4。

从流程图上来看，应该是这样。



* 1. while循环

程序执行到while处，当条件为True时，就去执行while内部的代码，当条件为False时，就跳过。



语法为：

   while 条件:

       循环执行的语句

同if一样，注意冒号，注意缩进。例子：

   a = 1            #变量赋值，将a设为1

   while a != 0:  #a不等于0就一直做

       print "please input"

       a = input()

   print "over"

注意，这里出现了两层缩进，要保持每层缩进的空格数相同。

* 1. for ... in ...循环

for i in range(a, b)

从a循环至b-1

for循环的本质是对一个序列中的元素进行递归。当你需要一个循环n次的循环，你就只需要写：

for i in range(1, n+1)或者for i in range(0, n)

区别在于前者i是从1到n，后者i是从0到n-1。当然，你也可以不用i这个变量名。

用while循环输出1到100，需要有一个值来记录已经做了多少次，还需要在while后面判断是不是到了100。如果用for循环，则可以这么写：

for i in range(1, 101): # range(1, 101)表示从1开始，到101为止（不包括101），取其中所有的整数。for i in range(1, 101)，把这些数依次赋值给变量i，相当于一个一个循环过去，第一次i = 1，第二次i = 2，……，直到i = 100。当i = 101时跳出循环

print i

**循环的嵌套**

要输出5个\*，或者矩阵

|  |  |
| --- | --- |
| for i in range(0, 5):     print '\*' | **\***  **\***  **\***  **\***  **\*** |

想让这5个\*在同一行，就在print语句后面加上逗号

for i in range(0, 5):

   print '\*', **# \* \* \* \* \***

循环的嵌套

两个嵌套在一起的循环，第二个for循环在第一个for循环的内部，表示每一次外层的循环中，都要进行一遍内层的循环。注意，第二个print的缩进和内层的for是一样的，这表明它是外层for循环中的语句，每次i的循环中，它会执行一次。print后面没有写任何东西，是起到换行的作用，这样，每输出5个\*，就会换行。

i从0到4循环了5次。对应于每一个i的值，j又做了从0到4五次循环。所以内层循环中的print语句一共被执行了5\*5一共25次。

|  |  |  |  |
| --- | --- | --- | --- |
| for i in range(0, 5):     for j in range(0, 5):         print i, j | 0 0  0 1  0 2  0 3  0 4  1 0  ...  4 4 | 输出矩阵  for i in range(0, 5):     print '\*\*\*\*\*'  用循环的方法输出矩阵  for i in range(0, 5):     for j in range(0, 5):         print '\*',     print | **\* \* \* \* \***  **\* \* \* \* \***  **\* \* \* \* \***  **\* \* \* \* \***  **\* \* \* \* \*** |

关联递进的循环

根据当前外层循环的序数，设置内层循环应当执行的次数。内层的j每次从0到i+1进行循环。这样，当第一次i=0时，j就是range(0,1)，只输出1个\*。而当最后一次i=4时，j就是range(0,5)，输出5个\*。

|  |  |
| --- | --- |
| for i in range(0, 5):     for j in range(0, i+1):         print '\*',     print | **\***  **\*\***  **\*\*\***  **\*\*\*\***  **\*\*\*\*\*** |

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000067&itemidx=1&sign=e21788fa0137c6dde3f22b88ff3d7fd6*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000067&itemidx=1&sign=e21788fa0137c6dde3f22b88ff3d7fd6)>

Friday, September 23, 2016

12:45

**条件控制 if 语句**

Python条件语句是通过一条或多条语句的执行结果（True或者False）来决定执行的代码块。

Python中if语句的一般形式如下所示：

if condition\_1: #如果 "condition\_1" 为 True 将执行 "statement\_block\_1" 块语句  
 statement\_block\_1  
elif condition\_2: #如果 "condition\_1" 为False，将判断 "condition\_2"   
 statement\_block\_2 #如果"condition\_2" 为 True 将执行 "statement\_block\_2" 块语句  
else: #如果 "condition\_2" 为False，将执行"statement\_block\_3"块语句  
 statement\_block\_3

**注意：**

* 1、每个条件后面要使用冒号（:），表示接下来是满足条件后要执行的语句块。
* 2、使用缩进来划分语句块，相同缩进数的语句在一起组成一个语句块。
* 3、Python 中，if语句的关键字为**if-elif-else，用elif** 代替了**else if**。没有switch – case语句。

**if 嵌套**

在嵌套 if 语句中，可以把 if...elif...else 结构放在另外一个 if...elif...else 结构中。

来自 <[*http://www.runoob.com/python3/python3-conditional-statements.html*](http://www.runoob.com/python3/python3-conditional-statements.html)>

Friday, September 23, 2016

12:46

**循环语句**  for 和 while

**while 循环**

Python中while语句的一般形式：

while 判断条件：  
 语句

同样需要注意冒号和缩进。另外，在Python中没有do..while循环。

以下实例使用了 while 来计算 1 到 100 的总和：

#!/usr/bin/env python3

n = 100

sum = 0  
counter = 1  
while counter <= n:  
 sum = sum + counter  
 counter += 1

print("1 到 %d 之和为: %d" % (n,sum))

执行结果如下：

1 到 100 之和为: 5050

**无限循环**

我们可以通过设置条件表达式永远不为 false 来实现无限循环，实例如下：

#!/usr/bin/python3

var = 1  
while var == 1 : # 表达式永远为 true  
 num = int(input("输入一个数字 :"))  
 print ("你输入的数字是: ", num)

print ("Good bye!")

执行以上脚本，输出结果如下：

输入一个数字 :5  
你输入的数字是: 5  
输入一个数字 :

你可以使用 **CTRL+C** 来退出当前的无限循环。

无限循环在服务器上客户端的实时请求非常有用。

**while 循环使用 else 语句**

在 while … else 在条件语句为 false 时执行 else 的语句块：

#!/usr/bin/python3

count = 0  
while count < 5:  
 print (count, " 小于 5")  
 count = count + 1  
else:  
 print (count, " 大于或等于 5")

执行以上脚本，输出结果如下：

0 小于 5  
1 小于 5  
2 小于 5  
3 小于 5  
4 小于 5  
5 大于或等于 5

**简单语句组**

类似if语句的语法，如果你的while循环体中只有一条语句，你可以将该语句与while写在同一行中， 如下所示：

#!/usr/bin/python

flag = 1

while (flag): print ('欢迎访问菜鸟教程!')

print ("Good bye!")

**注意：**以上的无限循环你可以使用 CTRL+C 来中断循环。

执行以上脚本，输出结果如下：

**for 语句**

Python for循环可以遍历任何序列的项目，如一个列表或者一个字符串。

for循环的一般格式如下：

for <variable> in <sequence>:  
 <statements>  
else:  
 <statements>

Python loop循环实例：

>>> languages = ["C", "C++", "Perl", "Python"]   
>>> for x in languages:  
... print (x)  
...   
C  
C++  
Perl  
Python  
>>>

以下 for 实例中使用了 break 语句，break 语句用于跳出当前循环体：

#!/usr/bin/python3

sites = ["Baidu", "Google","Runoob","Taobao"]  
for site in sites:  
 if site == "Runoob":  
 print("菜鸟教程!")  
 break  
 print("循环数据 " + site)  
else:  
 print("没有循环数据!")  
print("完成循环!")

执行脚本后，在循环到 "Runoob"时会跳出循环体：

循环数据 Baidu  
循环数据 Google  
菜鸟教程!  
完成循环!

**range()函数**

如果你需要遍历数字序列，可以使用内置range()函数。它会生成数列，例如:

>>> for i in range(5):  
... print(i)  
...  
0  
1  
2  
3  
4

你也可以使用range指定区间的值：

>>> for i in range(5,9) :  
 print(i)

5  
6  
7  
8  
>>>

也可以使range以指定数字开始并指定不同的增量(甚至可以是负数，有时这也叫做'步长'):

>>> for i in range(0, 10, 3) :  
 print(i)

0  
3  
6  
9  
>>>

负数：

>>> for i in range(-10, -100, -30) :  
 print(i)

-10  
-40  
-70  
>>>

您可以结合range()和len()函数以遍历一个序列的索引,如下所示:

>>> a = ['Google', 'Baidu', 'Runoob', 'Taobao', 'QQ']  
>>> for i in range(len(a)):  
... print(i, a[i])  
...   
0 Google  
1 Baidu  
2 Runoob  
3 Taobao  
4 QQ  
>>>

还可以使用range()函数来创建一个列表：

>>> list(range(5))  
[0, 1, 2, 3, 4]  
>>>

**函数原型：range（start， end， scan):**

参数含义：start:计数从start开始。默认是从0开始。例如range（5）等价于range（0， 5）;

end:技术到end结束，但不包括end.例如：range（0， 5） 是[0, 1, 2, 3, 4]没有5

scan：每次跳跃的间距，默认为1。例如：range（0， 5） 等价于 range(0, 5, 1)

示例

>>> range(5) #代表从0到5(不包含5)

[0, 1, 2, 3, 4]

>>> range(0,5) #代表从0到5(不包含5)

[0, 1, 2, 3, 4]

>>> range(1,5) #代表从1到5(不包含5)

[1, 2, 3, 4]

>>> range(1,5,2) #代表从1到5，间隔2(不包含5)

[1, 3]

来自 <[*http://www.cnblogs.com/wangwp/p/4535299.html*](http://www.cnblogs.com/wangwp/p/4535299.html)>

range(1,10)就是一个列表，意思是

I=[1, 2, 3, 4, 5, 6, 7, 8, 9] # 定义列表，格式就是用中括号包围、逗号隔开的一组数值

for循环做的事情其实就是遍历一个列表中的每一项，每次循环都把当前项赋值给一个变量（这里是i），直到列表结束。下面三种语句，意义一样。

|  |  |  |
| --- | --- | --- |
| for i in range(1, 10) ，     print i, | l = range(1, 10)  for i in l:     print i, | l = [1, 2, 3, 4, 5, 6, 7, 8, 9]  for i in l:     print i, |

**break和continue语句及循环中的else子句**

break 语句可以跳出 for 和 while 的循环体。如果你从 for 或 while 循环中终止，任何对应的循环 else 块将不执行。 实例如下：

#!/usr/bin/python3

for letter in 'Runoob': # 第一个实例  
 if letter == 'b':  
 break  
 print ('当前字母为 :', letter)  
   
var = 10 # 第二个实例  
while var > 0:   
 print ('当期变量值为 :', var)  
 var = var -1  
 if var == 5:  
 break

print ("Good bye!")

执行以上脚本输出结果为：

当前字母为 : R  
当前字母为 : u  
当前字母为 : n  
当前字母为 : o  
当前字母为 : o  
当期变量值为 : 10  
当期变量值为 : 9  
当期变量值为 : 8  
当期变量值为 : 7  
当期变量值为 : 6  
Good bye!

continue语句被用来告诉Python跳过当前循环块中的剩余语句，然后继续进行下一轮循环。

#!/usr/bin/python3

for letter in 'Runoob': # 第一个实例  
 if letter == 'o': # 字母为 o 时跳过输出  
 continue  
 print ('当前字母 :', letter)

var = 10 # 第二个实例  
while var > 0:   
 var = var -1  
 if var == 5: # 变量为 5 时跳过输出  
 continue  
 print ('当前变量值 :', var)  
print ("Good bye!")

执行以上脚本输出结果为：

当前字母 : R  
当前字母 : u  
当前字母 : n  
当前字母 : b  
当前变量值 : 9  
当前变量值 : 8  
当前变量值 : 7  
当前变量值 : 6  
当前变量值 : 4  
当前变量值 : 3  
当前变量值 : 2  
当前变量值 : 1  
当前变量值 : 0  
Good bye!

循环语句可以有 else 子句，它在穷尽列表(以for循环)或条件变为 false (以while循环)导致循环终止时被执行,但循环被break终止时不执行。

如下实例用于查询质数的循环例子:

#!/usr/bin/python3

for n in range(2, 10):  
 for x in range(2, n):  
 if n % x == 0:  
 print(n, '等于', x, '\*', n//x)  
 break  
 else:  
 # 循环中没有找到元素  
 print(n, ' 是质数')

执行以上脚本输出结果为：

2 是质数  
3 是质数  
4 等于 2 \* 2  
5 是质数  
6 等于 2 \* 3  
7 是质数  
8 等于 2 \* 4  
9 等于 3 \* 3

**pass 语句**

Python pass是空语句，是为了保持程序结构的完整性。

pass 不做任何事情，一般用做占位语句，如下实例

>>> while True:  
... pass # 等待键盘中断 (Ctrl+C)

最小的类:

>>> class MyEmptyClass:  
... pass

以下实例在字母为 o 时 执行 pass 语句块:

#!/usr/bin/python3

for letter in 'Runoob':   
 if letter == 'o':  
 pass  
 print ('执行 pass 块')  
 print ('当前字母 :', letter)

print ("Good bye!")

执行以上脚本输出结果为：

当前字母 : R  
当前字母 : u  
当前字母 : n  
执行 pass 块  
当前字母 : o  
执行 pass 块  
当前字母 : o  
当前字母 : b  
Good bye!

来自 <[*http://www.runoob.com/python3/python3-loop.html*](http://www.runoob.com/python3/python3-loop.html)>

Friday, September 23, 2016

12:47

**Python3 迭代器与生成器**

**迭代器**

迭代是Python最强大的功能之一，是访问集合元素的一种方式。。

迭代器是一个可以记住遍历的位置的对象。

迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

迭代器有两个基本的方法：**iter()** 和 **next()**。

字符串，列表或元组对象都可用于创建迭代器：

>>> list=[1,2,3,4]  
>>> it = iter(list) # 创建迭代器对象  
>>> print (next(it)) # 输出迭代器的下一个元素  
1  
>>> print (next(it))  
2  
>>>

迭代器对象可以使用常规for语句进行遍历：

#!/usr/bin/python3

list=[1,2,3,4]  
it = iter(list) # 创建迭代器对象  
for x in it:  
 print (x, end=" ")

执行以上程序，输出结果如下：

1 2 3 4

也可以使用 next() 函数：

#!/usr/bin/python3

import sys # 引入 sys 模块

list=[1,2,3,4]  
it = iter(list) # 创建迭代器对象

while True:  
 try:  
 print (next(it))  
 except StopIteration:  
 sys.exit()

执行以上程序，输出结果如下：

1  
2  
3  
4

**生成器**

在 Python 中，使用了 yield 的函数被称为生成器（generator）。

跟普通函数不同的是，生成器是一个返回迭代器的函数，只能用于迭代操作，更简单点理解生成器就是一个迭代器。

在调用生成器运行的过程中，每次遇到 yield 时函数会暂停并保存当前所有的运行信息，返回yield的值。并在下一次执行 next()方法时从当前位置继续运行。

以下实例使用 yield 实现斐波那契数列：

#!/usr/bin/python3

import sys

def fibonacci(n): # 生成器函数 - 斐波那契  
 a, b, counter = 0, 1, 0  
 while True:  
 if (counter > n):   
 return  
 yield a  
 a, b = b, a + b  
 counter += 1  
f = fibonacci(10) # f 是一个迭代器，由生成器返回生成

while True:  
 try:  
 print (next(f), end=" ")  
 except StopIteration:  
 sys.exit()

执行以上程序，输出结果如下：

0 1 1 2 3 5 8 13 21 34 55

来自 <[*http://www.runoob.com/python3/python3-iterator-generator.html*](http://www.runoob.com/python3/python3-iterator-generator.html)>

Friday, September 23, 2016

12:47

**Python3 函数**

函数是组织好的，可重复使用的，用来实现单一，或关联功能的代码段。能提高应用的模块性，和代码的重复利用率。

Python提供了许多内建函数，比如print()。用户也可以自定义函数。

**定义一个函数**

你可以定义一个由自己想要功能的函数，以下是简单的规则：

**语法**

**def function(arg1=10,arg2…): #def后接函数名。**传入参数放在圆括号中间。默认情况下，参数值和参数名称是按函数声明中定义的的顺序匹配起来的。

"打印任何传入的字符串" **#**函数的第一行，可以写函数说明

**pass #**函数内容以冒号起始，并且缩进。

**return sth #**结束函数，返回一个值给调用方。return后没有语句，就返回 None。

**按值传递参数和按引用传递参数**

在 Python 中，所有参数（变量）都是按引用传递。如果你在函数里修改了参数，那么在调用这个函数的函数里，原始的参数也被改变了。例如：

def changeme( mylist ):  
 "修改传入的列表"  
 mylist.append([1,2,3,4]);  
 print ("函数内取值: ", mylist)  
 return   
# 调用changeme函数  
mylist = [10,20,30];  
changeme( mylist );  
print ("函数外取值: ", mylist)

传入函数的和在末尾添加新内容的对象用的是同一个引用。故输出结果如下：

函数内取值: [10, 20, 30, [1, 2, 3, 4]]  
函数外取值: [10, 20, 30, [1, 2, 3, 4]]

**参数**

以下是调用函数时可使用的正式参数类型：

* 必需参数
* 关键字参数
* 默认参数
* 不定长参数

**必需参数**

必需参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样。

调用printme()函数，你必须传入一个参数，不然会出现语法错误：

#!/usr/bin/python3  
   
#可写函数说明  
def printme( str ):  
 "打印任何传入的字符串"  
 print (str);  
 return;  
   
#调用printme函数  
printme();

以上实例输出结果：

Traceback (most recent call last):  
 File "test.py", line 10, in <module>  
 printme();  
TypeError: printme() missing 1 required positional argument: 'str'

**关键字参数**

关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值。

使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值。

以下实例在函数 printme() 调用时使用参数名：

#!/usr/bin/python3  
   
#可写函数说明  
def printme( str ):  
 "打印任何传入的字符串"  
 print (str);  
 return;  
   
#调用printme函数  
printme( str = "菜鸟教程");

以上实例输出结果：

菜鸟教程

以下实例中演示了函数参数的使用不需要使用指定顺序：

#!/usr/bin/python3  
   
#可写函数说明  
def printinfo( name, age ):  
 "打印任何传入的字符串"  
 print ("名字: ", name);  
 print ("年龄: ", age);  
 return;  
   
#调用printinfo函数  
printinfo( age=50, name="runoob" );

以上实例输出结果：

名字: runoob  
年龄: 50

**默认参数**

调用函数时，如果没有传递参数，则会使用默认参数。以下实例中如果没有传入 age 参数，则使用默认值：

#!/usr/bin/python3  
  
#可写函数说明  
def printinfo( name, age = 35 ):  
 "打印任何传入的字符串"  
 print ("名字: ", name);  
 print ("年龄: ", age);  
 return;  
   
#调用printinfo函数  
printinfo( age=50, name="runoob" );  
print ("------------------------")  
printinfo( name="runoob" );

以上实例输出结果：

名字: runoob  
年龄: 50  
------------------------  
名字: runoob  
年龄: 35

**不定长参数**

你可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，和上述2种参数不同，声明时不会命名。基本语法如下：

def functionname([formal\_args,] \*var\_args\_tuple ):  
 "函数\_文档字符串"  
 function\_suite  
 return [expression]

加了星号（\*）的变量名会存放所有未命名的变量参数。如果在函数调用时没有指定参数，它就是一个空元组。我们也可以不向函数传递未命名的变量。如下实例：

#!/usr/bin/python3  
   
# 可写函数说明  
def printinfo( arg1, \*vartuple ):  
 "打印任何传入的参数"  
 print ("输出: ")  
 print (arg1)  
 for var in vartuple:  
 print (var)  
 return;  
   
# 调用printinfo 函数  
printinfo( 10 );  
printinfo( 70, 60, 50 );

以上实例输出结果：

输出:  
10  
输出:  
70  
60  
50

**匿名函数**

python 使用 lambda 来创建匿名函数。

所谓匿名，意即不再使用 def 语句这样标准的形式定义一个函数。

* lambda 只是一个表达式，函数体比 def 简单很多。
* lambda的主体是一个表达式，而不是一个代码块。仅仅能在lambda表达式中封装有限的逻辑进去。
* lambda 函数拥有自己的命名空间，且不能访问自有参数列表之外或全局命名空间里的参数。
* 虽然lambda函数看起来只能写一行，却不等同于C或C++的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。

**语法**

lambda 函数的语法只包含一个语句，如下：

lambda [arg1 [,arg2,.....argn]]:expression

如下实例：

#!/usr/bin/python3  
   
# 可写函数说明  
sum = lambda arg1, arg2: arg1 + arg2;  
   
# 调用sum函数  
print ("相加后的值为 : ", sum( 10, 20 ))  
print ("相加后的值为 : ", sum( 20, 20 ))

以上实例输出结果：

相加后的值为 : 30  
相加后的值为 : 40

**return语句**

**return [表达式]** 语句用于退出函数，选择性地向调用方返回一个表达式。不带参数值的return语句返回None。之前的例子都没有示范如何返回数值，以下实例演示了 return 语句的用法：

#!/usr/bin/python3

# 可写函数说明  
def sum( arg1, arg2 ):  
 # 返回2个参数的和."  
 total = arg1 + arg2  
 print ("函数内 : ", total)  
 return total;

# 调用sum函数  
total = sum( 10, 20 );  
print ("函数外 : ", total)

以上实例输出结果：

函数内 : 30  
函数外 : 30

**变量作用域**

Pyhton 中，程序的变量并不是在哪个位置都可以访问的，访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序可以访问哪个特定的变量名称。两种最基本的变量作用域如下：

* 全局变量
* 局部变量

**全局变量和局部变量**

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。如下实例：

#!/usr/bin/python3

total = 0; # 这是一个全局变量  
# 可写函数说明  
def sum( arg1, arg2 ):  
 #返回2个参数的和."  
 total = arg1 + arg2; # total在这里是局部变量.  
 print ("函数内是局部变量 : ", total)  
 return total;

#调用sum函数  
sum( 10, 20 );  
print ("函数外是全局变量 : ", total)

以上实例输出结果：

函数内是局部变量 : 30  
函数外是全局变量 : 0

来自 <[*http://www.runoob.com/python3/python3-function.html*](http://www.runoob.com/python3/python3-function.html)>

Wednesday, August 17, 2016

20:47

**函数**

数学上的函数，是指给定一个输入，就会有唯一输出的一种对应关系。编程语言里的函数跟这个意思差不多，但也有不同。函数就是一块语句，这块语句有个名字，你可以在需要时反复地使用这块语句。它有可能需要输入，有可能会返回输出。我们之前已经用到过python里内建的函数，比如input和range。以range(1,10)为例，range是这个函数的名称，后面括号里的1和10是参数。返回结果是一个从1到9的序列。

如果我们要自己写一个函数，就需要用关键字叫def（define的缩写）定义 它：

def sayHello():

   print 'hello world!' # sayHello是这个函数的名字，后面的括号里是参数，这里没有，表示不需要参数。但括号和后面的冒号都不能少。下面缩进的代码块就是整个函数的内容，称作函数体。

sayHello() # 然后我们去调用这个函数，得到和直接执行print 'hello world!'一样的结果。

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000086&itemidx=1&sign=e825894cfb9008b0ea78e81e5d930969*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000086&itemidx=1&sign=e825894cfb9008b0ea78e81e5d930969)>

**函数的参数**

调用函数的时候，可以传入值或变量。要注意提供的参数值的数量和类型需要跟函数定义中的一致，才能顺利调用它。还是注意，字符串类型的值不能少了引号。

例子1，调用时输入值

 def sayHello(someone): # 参数写在括号里。相当于定义了变量someone，但是没有被赋值。

   print someone + ' says Hello!'

sayHello('Crossin') # 函数调用的时候被赋值。相当于变量赋值 someone = Crossin ，然后执行语句print someone + ' says Hello!'

 例子2，调用时输入变量

 def plus(num1, num2): # 多个参数用逗号隔开。相当于定义了变量num1, num2，但是没有被赋值。

   print num1+num2

x = 3

y = 4

plus(x, y) # 相当于将变量x赋值给变量num1， x=3，num1 = x …然后执行语句print num1+num2

**函数应用示例**

函数可以把某个功能的代码分离出来，在需要的时候重复使用，就像拼装积木一样，这会让程序结构更清晰。

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

#第一次定义及使用函数

def youright(num1,num2): #定义函数

if num1<num2:

print ("Your answer is too small. Input another, please.")

return False; # return是函数的结束语句，函数中任何地方的return被执行到的时候，这个函数就会结束。return后面的值被作为这个函数的返回值。这里设定当两数相等的时候返回True，不等就返回False。

if num1>num2:

print ("Your answer is too big. Input another, please.")

return False;

if num1==num2:

print ("Bingo. You are right.")

return True

#函数代码块结束，下面是程序主代码。

from random import randint

num = randint(1,100)

print ("Guess what I think and input a number.")

check = False

while check==False: # 与赋值语句check = False，预设check的值是False，开启循环。

answer = eval(input()) # 经常出的Bug，Py3与Py2的input模块区别。

check = youright(answer,num) #调用函数。在youright函数内部，会输出answer和num的比较结果，如果相等的话，check会得到返回值True，否则返回False，循环继续。

Friday, September 23, 2016

12:47

**Python3 数据结构**

本章节我们主要结合前面所学的知识点来介绍Python数据结构。

**列表**

Python中列表是可变的，这是它区别于字符串和元组的最重要的特点，一句话概括即：列表可以修改，而字符串和元组不能。

以下是 Python 中列表的方法：

|  |  |
| --- | --- |
| **方法** | **描述** |
| list.append(x) | 把一个元素添加到列表的结尾，相当于 a[len(a):] = [x]。 |
| list.extend(L) | 通过添加指定列表的所有元素来扩充列表，相当于 a[len(a):] = L。 |
| list.insert(i, x) | 在指定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，例如 a.insert(0, x) 会插入到整个列表之前，而 a.insert(len(a), x) 相当于 a.append(x) 。 |
| list.remove(x) | 删除列表中值为 x 的第一个元素。如果没有这样的元素，就会返回一个错误。 |
| list.pop([i]) | 从列表的指定位置删除元素，并将其返回。如果没有指定索引，a.pop()返回最后一个元素。元素随即从列表中被删除。（方法中 i 两边的方括号表示这个参数是可选的，而不是要求你输入一对方括号，你会经常在 Python 库参考手册中遇到这样的标记。） |
| list.clear() | 移除列表中的所有项，等于del a[:]。 |
| list.index(x) | 返回列表中第一个值为 x 的元素的索引。如果没有匹配的元素就会返回一个错误。 |
| list.count(x) | 返回 x 在列表中出现的次数。 |
| list.sort() | 对列表中的元素进行排序。 |
| list.reverse() | 倒排列表中的元素。 |
| list.copy() | 返回列表的浅复制，等于a[:]。 |

下面示例演示了列表的大部分方法：

>>> a = [66.25, 333, 333, 1, 1234.5]  
>>> print(a.count(333), a.count(66.25), a.count('x'))  
2 1 0  
>>> a.insert(2, -1)  
>>> a.append(333)  
>>> a  
[66.25, 333, -1, 333, 1, 1234.5, 333]  
>>> a.index(333)  
1  
>>> a.remove(333)  
>>> a  
[66.25, -1, 333, 1, 1234.5, 333]  
>>> a.reverse()  
>>> a  
[333, 1234.5, 1, 333, -1, 66.25]  
>>> a.sort()  
>>> a  
[-1, 1, 66.25, 333, 333, 1234.5]

注意：类似 insert, remove 或 sort 等修改列表的方法没有返回值。

**将列表当做堆栈使用**

列表方法使得列表可以很方便的作为一个堆栈来使用，堆栈作为特定的数据结构，最先进入的元素最后一个被释放（后进先出）。用 append() 方法可以把一个元素添加到堆栈顶。用不指定索引的 pop() 方法可以把一个元素从堆栈顶释放出来。例如：

>>> stack = [3, 4, 5]  
>>> stack.append(6)  
>>> stack.append(7)  
>>> stack  
[3, 4, 5, 6, 7]  
>>> stack.pop()  
7  
>>> stack  
[3, 4, 5, 6]  
>>> stack.pop()  
6  
>>> stack.pop()  
5  
>>> stack  
[3, 4]

**将列表当作队列使用**

也可以把列表当做队列用，只是在队列里第一加入的元素，第一个取出来；但是拿列表用作这样的目的效率不高。在列表的最后添加或者弹出元素速度快，然而在列表里插入或者从头部弹出速度却不快（因为所有其他的元素都得一个一个地移动）。

>>> from collections import deque  
>>> queue = deque(["Eric", "John", "Michael"])  
>>> queue.append("Terry") # Terry arrives  
>>> queue.append("Graham") # Graham arrives  
>>> queue.popleft() # The first to arrive now leaves  
'Eric'  
>>> queue.popleft() # The second to arrive now leaves  
'John'  
>>> queue # Remaining queue in order of arrival  
deque(['Michael', 'Terry', 'Graham'])

**列表推导式**

列表推导式提供了从序列创建列表的简单途径。通常应用程序将一些操作应用于某个序列的每个元素，用其获得的结果作为生成新列表的元素，或者根据确定的判定条件创建子序列。

每个列表推导式都在 for 之后跟一个表达式，然后有零到多个 for 或 if 子句。返回结果是一个根据表达从其后的 for 和 if 上下文环境中生成出来的列表。如果希望表达式推导出一个元组，就必须使用括号。

这里我们将列表中每个数值乘三，获得一个新的列表：

>>> vec = [2, 4, 6]  
>>> [3\*x for x in vec]  
[6, 12, 18]

现在我们玩一点小花样：

>>> [[x, x\*\*2] for x in vec]  
[[2, 4], [4, 16], [6, 36]]

这里我们对序列里每一个元素逐个调用某方法：

>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']  
>>> [weapon.strip() for weapon in freshfruit]  
['banana', 'loganberry', 'passion fruit']

我们可以用 if 子句作为过滤器：

>>> [3\*x for x in vec if x > 3]  
[12, 18]  
>>> [3\*x for x in vec if x < 2]  
[]

以下是一些关于循环和其它技巧的演示：

>>> vec1 = [2, 4, 6]  
>>> vec2 = [4, 3, -9]  
>>> [x\*y for x in vec1 for y in vec2]  
[8, 6, -18, 16, 12, -36, 24, 18, -54]  
>>> [x+y for x in vec1 for y in vec2]  
[6, 5, -7, 8, 7, -5, 10, 9, -3]  
>>> [vec1[i]\*vec2[i] for i in range(len(vec1))]  
[8, 12, -54]

列表推导式可以使用复杂表达式或嵌套函数：

>>> [str(round(355/113, i)) for i in range(1, 6)]  
['3.1', '3.14', '3.142', '3.1416', '3.14159']

**嵌套列表解析**

Python的列表还可以嵌套。

以下实例展示了3X4的矩阵列表：

>>> matrix = [  
... [1, 2, 3, 4],  
... [5, 6, 7, 8],  
... [9, 10, 11, 12],  
... ]

以下实例将3X4的矩阵列表转换为4X3列表：

>>> [[row[i] for row in matrix] for i in range(4)]  
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

以下实例也可以使用以下方法来实现：

>>> transposed = []  
>>> for i in range(4):  
... transposed.append([row[i] for row in matrix])  
...  
>>> transposed  
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

另外一种实现方法：

>>> transposed = []  
>>> for i in range(4):  
... # the following 3 lines implement the nested listcomp  
... transposed\_row = []  
... for row in matrix:  
... transposed\_row.append(row[i])  
... transposed.append(transposed\_row)  
...  
>>> transposed  
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

**del 语句**

使用 del 语句可以从一个列表中依索引而不是值来删除一个元素。这与使用 pop() 返回一个值不同。可以用 del 语句从列表中删除一个切割，或清空整个列表（我们以前介绍的方法是给该切割赋一个空列表）。例如：

>>> a = [-1, 1, 66.25, 333, 333, 1234.5]  
>>> del a[0]  
>>> a  
[1, 66.25, 333, 333, 1234.5]  
>>> del a[2:4]  
>>> a  
[1, 66.25, 1234.5]  
>>> del a[:]  
>>> a  
[]

也可以用 del 删除实体变量：

>>> del a

**元组和序列**

元组由若干逗号分隔的值组成，例如：

>>> t = 12345, 54321, 'hello!'  
>>> t[0]  
12345  
>>> t  
(12345, 54321, 'hello!')  
>>> # Tuples may be nested:  
... u = t, (1, 2, 3, 4, 5)  
>>> u  
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))

如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构。在输入时可能有或没有括号， 不过括号通常是必须的（如果元组是更大的表达式的一部分）。

**集合**

集合是一个无序不重复元素的集。基本功能包括关系测试和消除重复元素。

可以用大括号({})创建集合。注意：如果要创建一个空集合，你必须用 set() 而不是 {} ；后者创建一个空的字典，下一节我们会介绍这个数据结构。

以下是一个简单的演示：

>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}  
>>> print(basket) # 删除重复的  
{'orange', 'banana', 'pear', 'apple'}  
>>> 'orange' in basket # 检测成员  
True  
>>> 'crabgrass' in basket  
False

>>> # 以下演示了两个集合的操作  
...  
>>> a = set('abracadabra')  
>>> b = set('alacazam')  
>>> a # a 中唯一的字母  
{'a', 'r', 'b', 'c', 'd'}  
>>> a - b # 在 a 中的字母，但不在 b 中  
{'r', 'd', 'b'}  
>>> a | b # 在 a 或 b 中的字母  
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}  
>>> a & b # 在 a 和 b 中都有的字母  
{'a', 'c'}  
>>> a ^ b # 在 a 或 b 中的字母，但不同时在 a 和 b 中  
{'r', 'd', 'b', 'm', 'z', 'l'}

集合也支持推导式：

>>> a = {x for x in 'abracadabra' if x not in 'abc'}  
>>> a  
{'r', 'd'}

**字典**

另一个非常有用的 Python 内建数据类型是字典。

序列是以连续的整数为索引，与此不同的是，字典以关键字为索引，关键字可以是任意不可变类型，通常用字符串或数值。

理解字典的最佳方式是把它看做无序的键=>值对集合。在同一个字典之内，关键字必须是互不相同。

一对大括号创建一个空的字典：{}。

这是一个字典运用的简单例子：

>>> tel = {'jack': 4098, 'sape': 4139}  
>>> tel['guido'] = 4127  
>>> tel  
{'sape': 4139, 'guido': 4127, 'jack': 4098}  
>>> tel['jack']  
4098  
>>> del tel['sape']  
>>> tel['irv'] = 4127  
>>> tel  
{'guido': 4127, 'irv': 4127, 'jack': 4098}  
>>> list(tel.keys())  
['irv', 'guido', 'jack']  
>>> sorted(tel.keys())  
['guido', 'irv', 'jack']  
>>> 'guido' in tel  
True  
>>> 'jack' not in tel  
False

构造函数 dict() 直接从键值对元组列表中构建字典。如果有固定的模式，列表推导式指定特定的键值对：

>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])  
{'sape': 4139, 'jack': 4098, 'guido': 4127}

此外，字典推导可以用来创建任意键和值的表达式词典：

>>> {x: x\*\*2 for x in (2, 4, 6)}  
{2: 4, 4: 16, 6: 36}

如果关键字只是简单的字符串，使用关键字参数指定键值对有时候更方便：

>>> dict(sape=4139, guido=4127, jack=4098)  
{'sape': 4139, 'jack': 4098, 'guido': 4127}

**遍历技巧**

在字典中遍历时，关键字和对应的值可以使用 items() 方法同时解读出来：

>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}  
>>> for k, v in knights.items():  
... print(k, v)  
...  
gallahad the pure  
robin the brave

在序列中遍历时，索引位置和对应值可以使用 enumerate() 函数同时得到：

>>> for i, v in enumerate(['tic', 'tac', 'toe']):  
... print(i, v)  
...  
0 tic  
1 tac  
2 toe

同时遍历两个或更多的序列，可以使用 zip() 组合：

>>> questions = ['name', 'quest', 'favorite color']  
>>> answers = ['lancelot', 'the holy grail', 'blue']  
>>> for q, a in zip(questions, answers):  
... print('What is your {0}? It is {1}.'.format(q, a))  
...  
What is your name? It is lancelot.  
What is your quest? It is the holy grail.  
What is your favorite color? It is blue.

要反向遍历一个序列，首先指定这个序列，然后调用 reversesd() 函数：

>>> for i in reversed(range(1, 10, 2)):  
... print(i)  
...  
9  
7  
5  
3  
1

要按顺序遍历一个序列，使用 sorted() 函数返回一个已排序的序列，并不修改原值：

>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']  
>>> for f in sorted(set(basket)):  
... print(f)  
...  
apple  
banana  
orange  
pear

来自 <[*http://www.runoob.com/python3/python3-data-structure.html*](http://www.runoob.com/python3/python3-data-structure.html)>

模块

Friday, September 23, 2016

12:48

Python2的package中必须要有一个\_\_init\_\_.py文件，才能被导入

在time模块下面有个ctime()方法可以获取当前系统时间

#import time

#print(time.ctime())

from time import ctime

print(ctime())

time模块下，还有其他函数，如果想知道有哪些，可以在命令行里 通过 help(time)查看

C: >python 
Python 3.4.3 Feb 24 2015. 22:44: 
064) ] on win32 
•copyright" 
'credits" or "license" for m 
>>> 
>>> elp(time) 
Help on 
NAME 
-in module time: 
This module prouides oar ious functions to m 

time.time() # 获取当前时间戳

time.ctime() # 当前时间的字符串形式

time.localtime() # 当前时间的struct\_time形式

time.strftime() # 获取当前时间，并且格式化成需要的字符串

**同一项目，跨目录的模块调用**

from model.func import \* # 从文件夹model的func模块调用所有的方法

**引入模块**

python还提供了很多模块，用来实现各种常见的功能，比如时间处理、科学计算、网络请求、随机数等等等等。

from random import randint # 引入模块的方法：from 模块名 import 方法名

answer = randint(1,100) # 模块中包含的函数randint,在1-100中取一个随机整数

print ("Guess what number I think? Input a number, please.")

user\_answer = eval(input()) # Python3 的input==raw\_input，加上eval()格式化

while user\_answer != answer: # 如果回答错了，就循环提问

if user\_answer > answer: # 两个if，针对不同的回答，给出不同的提示

print ("Oh, your answer is too big. Input a smaller one, please")

if user\_answer < answer:

print ("Oh, your answer is too small. Input a bigger one, please")

user\_answer = eval(input() ) # 再次输入

print ("You are smart, boy.") #如果回答正确，退出循环提问

**Python3 模块**

在前面的几个章节中我们脚本上是用 python 解释器来编程，如果你从 Python 解释器退出再进入，那么你定义的所有的方法和变量就都消失了。

为此 Python 提供了一个办法，把这些定义存放在文件中，为一些脚本或者交互式的解释器实例使用，这个文件被称为模块。

模块是一个包含所有你定义的函数和变量的文件，其后缀名是.py。模块可以被别的程序引入，以使用该模块中的函数等功能。这也是使用 python 标准库的方法。

下面是一个使用 python 标准库中模块的例子。

#!/usr/bin/python3  
# 文件名: using\_sys.py

import sys

print('命令行参数如下:')  
for i in sys.argv:  
 print(i)

print('\n\nPython 路径为：', sys.path, '\n')

执行结果如下所示：

$ python using\_sys.py 参数1 参数2  
命令行参数如下:  
using\_sys.py  
参数1  
参数2

Python 路径为： ['/root', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-x86\_64-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages']

* 1、import sys 引入 python 标准库中的 sys.py 模块；这是引入某一模块的方法。
* 2、sys.argv 是一个包含命令行参数的列表。
* 3、sys.path 包含了一个 Python 解释器自动查找所需模块的路径的列表。

**import 语句**

想使用 Python 源文件，只需在另一个源文件里执行 import 语句，语法如下：

import module1[, module2[,... moduleN]

当解释器遇到 import 语句，如果模块在当前的搜索路径就会被导入。

搜索路径是一个解释器会先进行搜索的所有目录的列表。如想要导入模块 support，需要把命令放在脚本的顶端：

support.py 文件代码为：

#!/usr/bin/python3  
# Filename: support.py

def print\_func( par ):  
 print ("Hello : ", par)  
 return

test.py 引入 support 模块：

#!/usr/bin/python3  
# Filename: test.py  
   
# 导入模块  
import support  
   
# 现在可以调用模块里包含的函数了  
support.print\_func("Runoob")

以上实例输出结果：

$ python3 test.py   
Hello : Runoob

一个模块只会被导入一次，不管你执行了多少次import。这样可以防止导入模块被一遍又一遍地执行。

当我们使用import语句的时候，Python解释器是怎样找到对应的文件的呢？

这就涉及到Python的搜索路径，搜索路径是由一系列目录名组成的，Python解释器就依次从这些目录中去寻找所引入的模块。

这看起来很像环境变量，事实上，也可以通过定义环境变量的方式来确定搜索路径。

搜索路径是在Python编译或安装的时候确定的，安装新的库应该也会修改。搜索路径被存储在sys模块中的path变量，做一个简单的实验，在交互式解释器中，输入以下代码：

>>> import sys  
>>> sys.path  
['', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-x86\_64-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages']  
>>>

sys.path 输出是一个列表，其中第一项是空串''，代表当前目录（若是从一个脚本中打印出来的话，可以更清楚地看出是哪个目录），亦即我们执行python解释器的目录（对于脚本的话就是运行的脚本所在的目录）。

因此若像我一样在当前目录下存在与要引入模块同名的文件，就会把要引入的模块屏蔽掉。

了解了搜索路径的概念，就可以在脚本中修改sys.path来引入一些不在搜索路径中的模块。

现在，在解释器的当前目录或者 sys.path 中的一个目录里面来创建一个fibo.py的文件，代码如下：

# 斐波那契(fibonacci)数列模块

def fib(n): # 定义到 n 的斐波那契数列  
 a, b = 0, 1  
 while b < n:  
 print(b, end=' ')  
 a, b = b, a+b  
 print()

def fib2(n): # 返回到 n 的斐波那契数列  
 result = []  
 a, b = 0, 1  
 while b < n:  
 result.append(b)  
 a, b = b, a+b  
 return result

然后进入Python解释器，使用下面的命令导入这个模块：

>>> import fibo

这样做并没有把直接定义在fibo中的函数名称写入到当前符号表里，只是把模块fibo的名字写到了那里。

可以使用模块名称来访问函数：

>>> fibo.fib(1000)  
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987  
>>> fibo.fib2(100)  
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]  
>>> fibo.\_\_name\_\_  
'fibo'

如果你打算经常使用一个函数，你可以把它赋给一个本地的名称：

>>> fib = fibo.fib  
>>> fib(500)  
1 1 2 3 5 8 13 21 34 55 89 144 233 377

**from…import 语句**

Python的from语句让你从模块中导入一个指定的部分到当前命名空间中，语法如下：

from modname import name1[, name2[, ... nameN]]

例如，要导入模块 fibo 的 fib 函数，使用如下语句：

>>> from fibo import fib, fib2  
>>> fib(500)  
1 1 2 3 5 8 13 21 34 55 89 144 233 377

这个声明不会把整个fibo模块导入到当前的命名空间中，它只会将fibo里的fib函数引入进来。

**From…import\* 语句**

把一个模块的所有内容全都导入到当前的命名空间也是可行的，只需使用如下声明：

from modname import \*

这提供了一个简单的方法来导入一个模块中的所有项目。然而这种声明不该被过多地使用。

**深入模块**

模块除了方法定义，还可以包括可执行的代码。这些代码一般用来初始化这个模块。这些代码只有在第一次被导入时才会被执行。

每个模块有各自独立的符号表，在模块内部为所有的函数当作全局符号表来使用。

所以，模块的作者可以放心大胆的在模块内部使用这些全局变量，而不用担心把其他用户的全局变量搞花。

从另一个方面，当你确实知道你在做什么的话，你也可以通过 modname.itemname 这样的表示法来访问模块内的函数。

模块是可以导入其他模块的。在一个模块（或者脚本，或者其他地方）的最前面使用 import 来导入一个模块，当然这只是一个惯例，而不是强制的。被导入的模块的名称将被放入当前操作的模块的符号表中。

还有一种导入的方法，可以使用 import 直接把模块内（函数，变量的）名称导入到当前操作模块。比如:

>>> from fibo import fib, fib2  
>>> fib(500)  
1 1 2 3 5 8 13 21 34 55 89 144 233 377

这种导入的方法不会把被导入的模块的名称放在当前的字符表中（所以在这个例子里面，fibo 这个名称是没有定义的）。

这还有一种方法，可以一次性的把模块中的所有（函数，变量）名称都导入到当前模块的字符表:

>>> from fibo import \*  
>>> fib(500)  
1 1 2 3 5 8 13 21 34 55 89 144 233 377

这将把所有的名字都导入进来，但是那些由单一下划线（\_）开头的名字不在此例。大多数情况， Python程序员不使用这种方法，因为引入的其它来源的命名，很可能覆盖了已有的定义。

**\_\_name\_\_属性**

一个模块被另一个程序第一次引入时，其主程序将运行。如果我们想在模块被引入时，模块中的某一程序块不执行，我们可以用\_\_name\_\_属性来使该程序块仅在该模块自身运行时执行。

#!/usr/bin/python3  
# Filename: using\_name.py

if \_\_name\_\_ == '\_\_main\_\_':  
 print('程序自身在运行')  
else:  
 print('我来自另一模块')

运行输出如下：

$ python using\_name.py

程序自身在运行

$ python  
>>> import using\_name  
我来自另一模块  
>>>

**说明：** 每个模块都有一个\_\_name\_\_属性，当其值是'\_\_main\_\_'时，表明该模块自身在运行，否则是被引入。

**dir() 函数**

内置的函数 dir() 可以找到模块内定义的所有名称。以一个字符串列表的形式返回:  
</p>  
<pre>  
>>> import fibo, sys  
>>> dir(fibo)  
['\_\_name\_\_', 'fib', 'fib2']  
>>> dir(sys)   
['\_\_displayhook\_\_', '\_\_doc\_\_', '\_\_excepthook\_\_', '\_\_loader\_\_', '\_\_name\_\_',  
 '\_\_package\_\_', '\_\_stderr\_\_', '\_\_stdin\_\_', '\_\_stdout\_\_',  
 '\_clear\_type\_cache', '\_current\_frames', '\_debugmallocstats', '\_getframe',  
 '\_home', '\_mercurial', '\_xoptions', 'abiflags', 'api\_version', 'argv',  
 'base\_exec\_prefix', 'base\_prefix', 'builtin\_module\_names', 'byteorder',  
 'call\_tracing', 'callstats', 'copyright', 'displayhook',  
 'dont\_write\_bytecode', 'exc\_info', 'excepthook', 'exec\_prefix',  
 'executable', 'exit', 'flags', 'float\_info', 'float\_repr\_style',  
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',  
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',  
 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',  
 'gettrace', 'hash\_info', 'hexversion', 'implementation', 'int\_info',  
 'intern', 'maxsize', 'maxunicode', 'meta\_path', 'modules', 'path',  
 'path\_hooks', 'path\_importer\_cache', 'platform', 'prefix', 'ps1',  
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',  
 'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',  
 'thread\_info', 'version', 'version\_info', 'warnoptions']

如果没有给定参数，那么 dir() 函数会罗列出当前定义的所有名称:

>>> a = [1, 2, 3, 4, 5]  
>>> import fibo  
>>> fib = fibo.fib  
>>> dir() # 得到一个当前模块中定义的属性列表  
['\_\_builtins\_\_', '\_\_name\_\_', 'a', 'fib', 'fibo', 'sys']  
>>> a = 5 # 建立一个新的变量 'a'  
>>> dir()  
['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', 'a', 'sys']  
>>>  
>>> del a # 删除变量名a  
>>>  
>>> dir()  
['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', 'sys']  
>>>

查看unittest模块有哪些成员

**[python]** [view plain copy](http://blog.csdn.net/five3/article/details/7104466)

1. >>import unittest
2. >>dir(unittest)
3. ['FunctionTestCase', 'TestCase', 'TestLoader', 'TestProgram', 'TestResult', 'Tes
4. tSuite', 'TextTestRunner', '\_CmpToKey', '\_TextTestResult', '\_WritelnDecorator',
5. '\_\_all\_\_', '\_\_author\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_email\_\_', '\_\_file\_\_', '\_\_
6. metaclass\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_unittest', '\_\_version\_\_', '\_makeLoad
7. er', '\_strclass', 'defaultTestLoader', 'findTestCases', 'getTestCaseNames', 'mai
8. n', 'makeSuite', 'os', 'sys', 'time', 'traceback', 'types']

可以看到其自身的成员也不是很多，大概包括有：

['FunctionTestCase', 'TestCase', 'TestLoader', 'TestProgram', 'TestResult',

 'TestSuite','TextTestRunner', '\_CmpToKey', '\_TextTestResult', '\_WritelnDecorator',

 'defaultTestLoader','findTestCases', 'getTestCaseNames', 'main', 'makeSuite']

 好吧,我们大概看看具体都是干什么的

**[python]** [view plain copy](http://blog.csdn.net/five3/article/details/7104466)

1. >>memblist = ['FunctionTestCase', 'TestCase', 'TestLoader', 'TestProgram', 'TestResult',\
2. 'TestSuite','TextTestRunner', 'defaultTestLoader','findTestCases', 'getTestCaseNames', \
3. 'main', 'makeSuite']
4. >>for memb in memblist:
5. ..  cur = getattr(unittest, memb)
6. ..  print help(cur)

来自 <[*http://blog.csdn.net/five3/article/details/7104466*](http://blog.csdn.net/five3/article/details/7104466)>

**标准模块**

Python 本身带着一些标准的模块库，在 Python 库参考文档中将会介绍到（就是后面的"库参考文档"）。

有些模块直接被构建在解析器里，这些虽然不是一些语言内置的功能，但是他却能很高效的使用，甚至是系统级调用也没问题。

这些组件会根据不同的操作系统进行不同形式的配置，比如 winreg 这个模块就只会提供给 Windows 系统。

应该注意到这有一个特别的模块 sys ，它内置在每一个 Python 解析器中。变量 sys.ps1 和 sys.ps2 定义了主提示符和副提示符所对应的字符串:

>>> import sys  
>>> sys.ps1  
'>>> '  
>>> sys.ps2  
'... '  
>>> sys.ps1 = 'C> '  
C> print('Yuck!')  
Yuck!  
C>

**包**

包是一种管理 Python 模块命名空间的形式，采用"点模块名称"。

比如一个模块的名称是 A.B， 那么他表示一个包 A中的子模块 B 。

就好像使用模块的时候，你不用担心不同模块之间的全局变量相互影响一样，采用点模块名称这种形式也不用担心不同库之间的模块重名的情况。

这样不同的作者都可以提供 NumPy 模块，或者是 Python 图形库。

不妨假设你想设计一套统一处理声音文件和数据的模块（或者称之为一个"包"）。

现存很多种不同的音频文件格式（基本上都是通过后缀名区分的，例如： .wav，:file:.aiff，:file:.au，），所以你需要有一组不断增加的模块，用来在不同的格式之间转换。

并且针对这些音频数据，还有很多不同的操作（比如混音，添加回声，增加均衡器功能，创建人造立体声效果），所你还需要一组怎么也写不完的模块来处理这些操作。

这里给出了一种可能的包结构（在分层的文件系统中）:

sound/ 顶层包  
 \_\_init\_\_.py 初始化 sound 包  
 formats/ 文件格式转换子包  
 \_\_init\_\_.py  
 wavread.py  
 wavwrite.py  
 aiffread.py  
 aiffwrite.py  
 auread.py  
 auwrite.py  
 ...  
 effects/ 声音效果子包  
 \_\_init\_\_.py  
 echo.py  
 surround.py  
 reverse.py  
 ...  
 filters/ filters 子包  
 \_\_init\_\_.py  
 equalizer.py  
 vocoder.py  
 karaoke.py  
 ...

在导入一个包的时候，Python 会根据 sys.path 中的目录来寻找这个包中包含的子目录。

目录只有包含一个叫做 \_\_init\_\_.py 的文件才会被认作是一个包，主要是为了避免一些滥俗的名字（比如叫做 string）不小心的影响搜索路径中的有效模块。

最简单的情况，放一个空的 :[file:\_\_init\_\_.py就可以了。当然这个文件中也可以包含一些初始化代码或者为（将在后面介绍的）](file://__init__.py就可以了。当然这个文件中也可以包含一些初始化代码或者为（将在后面介绍的）) \_\_all\_\_变量赋值。

用户可以每次只导入一个包里面的特定模块，比如:

import sound.effects.echo

这将会导入子模块:sound.effects.echo。 他必须使用全名去访问:

sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)

还有一种导入子模块的方法是:

from sound.effects import echo

这同样会导入子模块: echo，并且他不需要那些冗长的前缀，所以他可以这样使用:

echo.echofilter(input, output, delay=0.7, atten=4)

还有一种变化就是直接导入一个函数或者变量:

from sound.effects.echo import echofilter

同样的，这种方法会导入子模块: echo，并且可以直接使用他的 echofilter() 函数:

echofilter(input, output, delay=0.7, atten=4)

注意当使用from package import item这种形式的时候，对应的item既可以是包里面的子模块（子包），或者包里面定义的其他名称，比如函数，类或者变量。

import语法会首先把item当作一个包定义的名称，如果没找到，再试图按照一个模块去导入。如果还没找到，恭喜，一个:exc:ImportError 异常被抛出了。

反之，如果使用形如import item.subitem.subsubitem这种导入形式，除了最后一项，都必须是包，而最后一项则可以是模块或者是包，但是不可以是类，函数或者变量的名字。

**从一个包中导入\***

设想一下，如果我们使用 from sound.effects import \*会发生什么？

Python 会进入文件系统，找到这个包里面所有的子模块，一个一个的把它们都导入进来。

但是很不幸，这个方法在 Windows平台上工作的就不是非常好，因为Windows是一个大小写不区分的系统。

在这类平台上，没有人敢担保一个叫做 ECHO.py 的文件导入为模块 echo 还是 Echo 甚至 ECHO。

（例如，Windows 95就很讨厌的把每一个文件的首字母大写显示）而且 DOS 的 8+3 命名规则对长模块名称的处理会把问题搞得更纠结。

为了解决这个问题，只能烦劳包作者提供一个精确的包的索引了。

导入语句遵循如下规则：如果包定义文件 \_\_init\_\_.py 存在一个叫做 \_\_all\_\_ 的列表变量，那么在使用 from package import \* 的时候就把这个列表中的所有名字作为包内容导入。

作为包的作者，可别忘了在更新包之后保证 \_\_all\_\_ 也更新了啊。你说我就不这么做，我就不使用导入\*这种用法，好吧，没问题，谁让你是老板呢。这里有一个例子，在:file:sounds/effects/\_\_init\_\_.py中包含如下代码:

\_\_all\_\_ = ["echo", "surround", "reverse"]

这表示当你使用from sound.effects import \*这种用法时，你只会导入包里面这三个子模块。

如果 **\_\_all\_\_** 真的没有定义，那么使用**from sound.effects import \***这种语法的时候，就不会导入包 sound.effects 里的任何子模块。他只是把包sound.effects和它里面定义的所有内容导入进来（可能运行\_\_init\_\_.py里定义的初始化代码）。

这会把 \_\_init\_\_.py 里面定义的所有名字导入进来。并且他不会破坏掉我们在这句话之前导入的所有明确指定的模块。看下这部分代码:

import sound.effects.echo  
import sound.effects.surround  
from sound.effects import \*

这个例子中，在执行from...import前，包sound.effects中的echo和surround模块都被导入到当前的命名空间中了。（当然如果定义了\_\_all\_\_就更没问题了）

通常我们并不主张使用\*这种方法来导入模块，因为这种方法经常会导致代码的可读性降低。不过这样倒的确是可以省去不少敲键的功夫，而且一些模块都设计成了只能通过特定的方法导入。

记住，使用from Package import specific\_submodule这种方法永远不会有错。事实上，这也是推荐的方法。除非是你要导入的子模块有可能和其他包的子模块重名。

如果在结构中包是一个子包（比如这个例子中对于包sound来说），而你又想导入兄弟包（同级别的包）你就得使用导入绝对的路径来导入。比如，如果模块 sound.filters.vocoder 要使用包sound.effects中的模块echo，你就要写成 from sound.effects import echo。

from . import echo  
from .. import formats  
from ..filters import equalizer

无论是隐式的还是显式的相对导入都是从当前模块开始的。主模块的名字永远是"\_\_main\_\_"，一个Python应用程序的主模块，应当总是使用绝对路径引用。

包还提供一个额外的属性\_\_path\_\_。这是一个目录列表，里面每一个包含的目录都有为这个包服务的\_\_init\_\_.py，你得在其他\_\_init\_\_.py被执行前定义哦。可以修改这个变量，用来影响包含在包里面的模块和子包。

这个功能并不常用，一般用来扩展包里面的模块。

来自 <[*http://www.runoob.com/python3/python3-module.html*](http://www.runoob.com/python3/python3-module.html)>

Friday, September 23, 2016

12:49

**Python3 输入和输出**

在前面几个章节中，我们其实已经接触了 Python 的输入输出的功能。本章节我们将具体介绍 Python 的输入输出。

**输出格式美化**

Python两种输出值的方式: 表达式语句和 print() 函数。

第三种方式是使用文件对象的 write() 方法，标准输出文件可以用 sys.stdout 引用。

如果你希望输出的形式更加多样，可以使用 str.format() 函数来格式化输出值。

如果你希望将输出的值转成字符串，可以使用 repr() 或 str() 函数来实现。

* **str()：** 函数返回一个用户易读的表达形式。
* **repr()：** 产生一个解释器易读的表达形式。

**例如**

>>> s = 'Hello, Runoob'  
>>> str(s)  
'Hello, Runoob'  
>>> repr(s)  
"'Hello, Runoob'"  
>>> str(1/7)  
'0.14285714285714285'  
>>> x = 10 \* 3.25  
>>> y = 200 \* 200  
>>> s = 'x 的值为： ' + repr(x) + ', y 的值为：' + repr(y) + '...'  
>>> print(s)  
x 的值为： 32.5, y 的值为：40000...  
>>> # repr() 函数可以转义字符串中的特殊字符  
... hello = 'hello, runoob\n'  
>>> hellos = repr(hello)  
>>> print(hellos)  
'hello, runoob\n'  
>>> # repr() 的参数可以是 Python 的任何对象  
... repr((x, y, ('Google', 'Runoob')))  
"(32.5, 40000, ('Google', 'Runoob'))"

这里有两种方式输出一个平方与立方的表:

>>> for x in range(1, 11):  
... print(repr(x).rjust(2), repr(x\*x).rjust(3), end=' ')  
... # 注意前一行 'end' 的使用  
... print(repr(x\*x\*x).rjust(4))  
...  
 1 1 1  
 2 4 8  
 3 9 27  
 4 16 64  
 5 25 125  
 6 36 216  
 7 49 343  
 8 64 512  
 9 81 729  
10 100 1000

>>> for x in range(1, 11):  
... print('{0:2d} {1:3d} {2:4d}'.format(x, x\*x, x\*x\*x))  
...  
 1 1 1  
 2 4 8  
 3 9 27  
 4 16 64  
 5 25 125  
 6 36 216  
 7 49 343  
 8 64 512  
 9 81 729  
10 100 1000

**注意：**在第一个例子中, 每列间的空格由 print() 添加。

这个例子展示了字符串对象的 rjust() 方法, 它可以将字符串靠右, 并在左边填充空格。

还有类似的方法, 如 ljust() 和 center()。 这些方法并不会写任何东西, 它们仅仅返回新的字符串。

另一个方法 zfill(), 它会在数字的左边填充 0，如下所示：

>>> '12'.zfill(5)  
'00012'  
>>> '-3.14'.zfill(7)  
'-003.14'  
>>> '3.14159265359'.zfill(5)  
'3.14159265359'

str.format() 的基本使用如下:

>>> print('{}网址： "{}!"'.format('菜鸟教程', '[www.runoob.com](http://www.runoob.com)'))  
菜鸟教程网址： "[www.runoob.com!](http://www.runoob.com!)"

括号及其里面的字符 (称作格式化字段) 将会被 format() 中的参数替换。

在括号中的数字用于指向传入对象在 format() 中的位置，如下所示：

>>> print('{0} 和 {1}'.format('Google', 'Runoob'))  
Google 和 Runoob  
>>> print('{1} 和 {0}'.format('Google', 'Runoob'))  
Runoob 和 Google

如果在 format() 中使用了关键字参数, 那么它们的值会指向使用该名字的参数。

>>> print('{name}网址： {site}'.format(name='菜鸟教程', site='www.runoob.com'))  
菜鸟教程网址： [www.runoob.com](http://www.runoob.com)

位置及关键字参数可以任意的结合:

>>> print('站点列表 {0}, {1}, 和 {other}。'.format('Google', 'Runoob',  
 other='Taobao'))  
站点列表 Google, Runoob, 和 Taobao。

'!a' (使用 ascii()), '!s' (使用 str()) 和 '!r' (使用 repr()) 可以用于在格式化某个值之前对其进行转化:

>>> import math  
>>> print('常量 PI 的值近似为： {}。'.format(math.pi))  
常量 PI 的值近似为： 3.141592653589793。  
>>> print('常量 PI 的值近似为： {!r}。'.format(math.pi))  
常量 PI 的值近似为： 3.141592653589793。

可选项 ':' 和格式标识符可以跟着字段名。 这就允许对值进行更好的格式化。 下面的例子将 Pi 保留到小数点后三位：

>>> import math  
>>> print('常量 PI 的值近似为 {0:.3f}。'.format(math.pi))  
常量 PI 的值近似为 3.142。

在 ':' 后传入一个整数, 可以保证该域至少有这么多的宽度。 用于美化表格时很有用。

>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}  
>>> for name, number in table.items():  
... print('{0:10} ==> {1:10d}'.format(name, number))  
...  
Runoob ==> 2  
Taobao ==> 3  
Google ==> 1

如果你有一个很长的格式化字符串, 而你不想将它们分开, 那么在格式化时通过变量名而非位置会是很好的事情。

最简单的就是传入一个字典, 然后使用方括号 '[]' 来访问键值 :

>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}  
>>> print('Runoob: {0[Runoob]:d}; Google: {0[Google]:d}; '  
 'Taobao: {0[Taobao]:d}'.format(table))  
Runoob: 2; Google: 1; Taobao: 3

也可以通过在 table 变量前使用 '\*\*' 来实现相同的功能：

>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}  
>>> print('Runoob: {Runoob:d}; Google: {Google:d}; Taobao: {Taobao:d}'.format(\*\*table))  
Runoob: 2; Google: 1; Taobao: 3

**旧式字符串格式化**

% 操作符也可以实现字符串格式化。 它将左边的参数作为类似 sprintf() 式的格式化字符串, 而将右边的代入, 然后返回格式化后的字符串. 例如:

>>> import math  
>>> print('常量 PI 的值近似为：%5.3f。' % math.pi)  
常量 PI 的值近似为：3.142。

因为 str.format() 比较新的函数， 大多数的 Python 代码仍然使用 % 操作符。但是因为这种旧式的格式化最终会从该语言中移除, 应该更多的使用 str.format().

**读取键盘输入**

Python提供了 input() 置函数从标准输入读入一行文本，默认的标准输入是键盘。

input 可以接收一个Python表达式作为输入，并将运算结果返回。

#!/usr/bin/python3

str = input("请输入：");  
print ("你输入的内容是: ", str)

这会产生如下的对应着输入的结果：

请输入：菜鸟教程  
你输入的内容是: 菜鸟教程

**读和写文件**

open() 将会返回一个 file 对象，基本语法格式如下:

open(filename, mode)

* filename：filename 变量是一个包含了你要访问的文件名称的字符串值。
* mode：mode决定了打开文件的模式：只读，写入，追加等。所有可取值见如下的完全列表。这个参数是非强制的，默认文件访问模式为只读(r)。

不同模式打开文件的完全列表：

|  |  |
| --- | --- |
| **模式** | **描述** |
| r | 以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。 |
| rb | 以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。 |
| r+ | 打开一个文件用于读写。文件指针将会放在文件的开头。 |
| rb+ | 以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。 |
| w | 打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。 |
| wb | 以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。 |
| w+ | 打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。 |
| wb+ | 以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。 |
| a | 打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。 |
| ab | 以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。 |
| a+ | 打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。 |
| ab+ | 以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。 |

以下实例将字符串写入到文件 foo.txt 中：

#!/usr/bin/python3

# 打开一个文件  
f = open("/tmp/foo.txt", "w")

f.write( "Python 是一个非常好的语言。\n是的，的确非常好!!\n" )

# 关闭打开的文件  
f.close()

* 第一个参数为要打开的文件名。
* 第二个参数描述文件如何使用的字符。 mode 可以是 'r' 如果文件只读, 'w' 只用于写 (如果存在同名文件则将被删除), 和 'a' 用于追加文件内容; 所写的任何数据都会被自动增加到末尾. 'r+' 同时用于读写。 mode 参数是可选的; 'r' 将是默认值。

此时打开文件 foo.txt,显示如下：

$ cat /tmp/foo.txt   
Python 是一个非常好的语言。  
是的，的确非常好!!

**文件对象的方法**

本节中剩下的例子假设已经创建了一个称为 f 的文件对象。

**f.read()**

为了读取一个文件的内容，调用 f.read(size), 这将读取一定数目的数据, 然后作为字符串或字节对象返回。

size 是一个可选的数字类型的参数。 当 size 被忽略了或者为负, 那么该文件的所有内容都将被读取并且返回。

以下实例假定文件 foo.txt 已存在（上面实例中已创建）：

#!/usr/bin/python3

# 打开一个文件  
f = open("/tmp/foo.txt", "r")

str = f.read()  
print(str)

# 关闭打开的文件  
f.close()

执行以上程序，输出结果为：

Python 是一个非常好的语言。  
是的，的确非常好!!

**f.readline()**

f.readline() 会从文件中读取单独的一行。换行符为 '\n'。f.readline() 如果返回一个空字符串, 说明已经已经读取到最后一行。

#!/usr/bin/python3

# 打开一个文件  
f = open("/tmp/foo.txt", "r")

str = f.readline()  
print(str)

# 关闭打开的文件  
f.close()

执行以上程序，输出结果为：

Python 是一个非常好的语言。

**f.readlines()**

f.readlines() 将返回该文件中包含的所有行。

如果设置可选参数 sizehint, 则读取指定长度的字节, 并且将这些字节按行分割。

#!/usr/bin/python3

# 打开一个文件  
f = open("/tmp/foo.txt", "r")

str = f.readlines()  
print(str)

# 关闭打开的文件  
f.close()

执行以上程序，输出结果为：

['Python 是一个非常好的语言。\n', '是的，的确非常好!!\n']

另一种方式是迭代一个文件对象然后读取每行:

#!/usr/bin/python3

# 打开一个文件  
f = open("/tmp/foo.txt", "r")

for line in f:  
 print(line, end='')

# 关闭打开的文件  
f.close()

执行以上程序，输出结果为：

Python 是一个非常好的语言。  
是的，的确非常好!!

这个方法很简单, 但是并没有提供一个很好的控制。 因为两者的处理机制不同, 最好不要混用。

**f.write()**

f.write(string) 将 string 写入到文件中, 然后返回写入的字符数。

#!/usr/bin/python3

# 打开一个文件  
f = open("/tmp/foo.txt", "w")

num = f.write( "Python 是一个非常好的语言。\n是的，的确非常好!!\n" )  
print(num)  
# 关闭打开的文件  
f.close()

执行以上程序，输出结果为：

29

如果要写入一些不是字符串的东西, 那么将需要先进行转换:

#!/usr/bin/python3

# 打开一个文件  
f = open("/tmp/foo1.txt", "w")

value = ('[www.runoob.com](http://www.runoob.com)', 14)  
s = str(value)  
f.write(s)

# 关闭打开的文件  
f.close()

执行以上程序，打开 foo1.txt 文件：

$ cat /tmp/foo1.txt   
('[www.runoob.com](http://www.runoob.com)', 14)

**f.tell()**

f.tell() 返回文件对象当前所处的位置, 它是从文件开头开始算起的字节数。

**f.seek()**

如果要改变文件当前的位置, 可以使用 f.seek(offset, from\_what) 函数。

from\_what 的值, 如果是 0 表示开头, 如果是 1 表示当前位置, 2 表示文件的结尾，例如：

* seek(x,0) ： 从起始位置即文件首行首字符开始移动 x 个字符
* seek(x,1) ： 表示从当前位置往后移动x个字符
* seek(-x,2)：表示从文件的结尾往前移动x个字符

from\_what 值为默认为0，即文件开头。下面给出一个完整的例子：

>>> f = open('/tmp/foo.txt', 'rb+')  
>>> f.write(b'0123456789abcdef')  
16  
>>> f.seek(5) # 移动到文件的第六个字节  
5  
>>> f.read(1)  
b'5'  
>>> f.seek(-3, 2) # 移动到文件的倒数第三字节  
13  
>>> f.read(1)  
b'd'

**f.close()**

在文本文件中 (那些打开文件的模式下没有 b 的), 只会相对于文件起始位置进行定位。

当你处理完一个文件后, 调用 f.close() 来关闭文件并释放系统的资源，如果尝试再调用该文件，则会抛出异常。

>>> f.close()  
>>> f.read()  
Traceback (most recent call last):  
 File "<stdin>", line 1, in ?  
ValueError: I/O operation on closed file  
<pre>  
<p>  
当处理一个文件对象时, 使用 with 关键字是非常好的方式。在结束后, 它会帮你正确的关闭文件。 而且写起来也比 try - finally 语句块要简短:</p>  
<pre>  
>>> with open('/tmp/foo.txt', 'r') as f:  
... read\_data = f.read()  
>>> f.closed  
True

文件对象还有其他方法, 如 isatty() 和 trucate(), 但这些通常比较少用。

**pickle 模块**

python的pickle模块实现了基本的数据序列和反序列化。

通过pickle模块的序列化操作我们能够将程序中运行的对象信息保存到文件中去，永久存储。

通过pickle模块的反序列化操作，我们能够从文件中创建上一次程序保存的对象。

基本接口：

pickle.dump(obj, file, [,protocol])

有了 pickle 这个对象, 就能对 file 以读取的形式打开:

x = pickle.load(file)

**注解：**从 file 中读取一个字符串，并将它重构为原来的python对象。

**file:** 类文件对象，有read()和readline()接口。

实例1：

#!/usr/bin/python3  
import pickle

# 使用pickle模块将数据对象保存到文件  
data1 = {'a': [1, 2.0, 3, 4+6j],  
 'b': ('string', u'Unicode string'),  
 'c': None}

selfref\_list = [1, 2, 3]  
selfref\_list.append(selfref\_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.  
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.  
pickle.dump(selfref\_list, output, -1)

output.close()

实例2：

#!/usr/bin/python3  
import pprint, pickle

#使用pickle模块从文件中重构python对象  
pkl\_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl\_file)  
pprint.pprint(data1)

data2 = pickle.load(pkl\_file)  
pprint.pprint(data2)

pkl\_file.close()

来自 <[*http://www.runoob.com/python3/python3-inputoutput.html*](http://www.runoob.com/python3/python3-inputoutput.html)>

Friday, September 23, 2016

12:50

**Python3 File(文件) 方法** in\_28.py

file 对象使用 open 函数来创建，下表列出了 file 对象常用的函数：

序号        方法及描述

来自 <[*http://www.runoob.com/python3/python3-file-methods.html*](http://www.runoob.com/python3/python3-file-methods.html)>

|  |  |
| --- | --- |
| **序号** | **方法及描述** |
| 2 | [file.flush()](http://www.runoob.com/python3/python3-file-flush.html)  刷新文件内部缓冲，直接把内部缓冲区的数据立刻写入文件, 而不是被动的等待输出缓冲区写入。 |
| 3 | [file.fileno()](http://www.runoob.com/python3/python3-file-fileno.html)  返回一个整型的文件描述符(file descriptor FD 整型), 可以用在如os模块的read方法等一些底层操作上。 |
| 4 | [file.isatty()](http://www.runoob.com/python3/python3-file-isatty.html)  如果文件连接到一个终端设备返回 True，否则返回 False。 |
| 5 | [file.next()](http://www.runoob.com/python3/python3-file-next.html)  返回文件下一行。 |
| 7 | [file.readline([size])](http://www.runoob.com/python3/python3-file-readline.html)  读取整行，包括 "\n" 字符。 |
| 8 | [file.readlines([sizehint])](http://www.runoob.com/python3/python3-file-readlines.html)  读取所有行并返回列表，若给定sizeint>0，返回总和大约为sizeint字节的行, 实际读取值可能比sizhint较大, 因为需要填充缓冲区。 |
| 9 | [file.seek(offset[, whence])](http://www.runoob.com/python3/python3-file-seek.html)  设置文件当前位置 |
| 10 | [file.tell()](http://www.runoob.com/python3/python3-file-tell.html)  返回文件当前位置。 |
| 11 | [file.truncate([size])](http://www.runoob.com/python3/python3-file-truncate.html)  截取文件，截取的字节通过size指定，默认为当前文件位置。 |
| 13 | [file.writelines(sequence)](http://www.runoob.com/python3/python3-file-writelines.html)  向文件写入一个序列字符串列表，如果需要换行则要自己加入每行的换行符。 |

来自 <[*http://www.runoob.com/python3/python3-file-methods.html*](http://www.runoob.com/python3/python3-file-methods.html)>

Friday, September 23, 2016

12:50

**Python3 OS 文件/目录方法**

**os** 模块提供了非常丰富的方法用来处理文件和目录。常用的方法如下表所示：

|  |  |
| --- | --- |
| **序号** | **方法及描述** |
| 1 | [os.access(path, mode)](http://www.runoob.com/python3/python3-os-access.html)  检验权限模式 |
| 2 | [os.chdir(path)](http://www.runoob.com/python3/python3-os-chdir.html)  改变当前工作目录 |
| 3 | [os.chflags(path, flags)](http://www.runoob.com/python3/python3-os-chflags.html)  设置路径的标记为数字标记。 |
| 4 | [os.chmod(path, mode)](http://www.runoob.com/python3/python3-os-chmod.html)  更改权限 |
| 5 | [os.chown(path, uid, gid)](http://www.runoob.com/python3/python3-os-chown.html)  更改文件所有者 |
| 6 | [os.chroot(path)](http://www.runoob.com/python3/python3-os-chroot.html)  改变当前进程的根目录 |
| 7 | [os.close(fd)](http://www.runoob.com/python3/python3-os-close.html)  关闭文件描述符 fd |
| 8 | [os.closerange(fd\_low, fd\_high)](http://www.runoob.com/python3/python3-os-closerange.html)  关闭所有文件描述符，从 fd\_low (包含) 到 fd\_high (不包含), 错误会忽略 |
| 9 | [os.dup(fd)](http://www.runoob.com/python3/python3-os-dup.html)  复制文件描述符 fd |
| 10 | [os.dup2(fd, fd2)](http://www.runoob.com/python3/python3-os-dup2.html)  将一个文件描述符 fd 复制到另一个 fd2 |
| 11 | [os.fchdir(fd)](http://www.runoob.com/python3/python3-os-fchdir.html)  通过文件描述符改变当前工作目录 |
| 12 | [os.fchmod(fd, mode)](http://www.runoob.com/python3/python3-os-fchmod.html)  改变一个文件的访问权限，该文件由参数fd指定，参数mode是Unix下的文件访问权限。 |
| 13 | [os.fchown(fd, uid, gid)](http://www.runoob.com/python3/python3-os-fchown.html)  修改一个文件的所有权，这个函数修改一个文件的用户ID和用户组ID，该文件由文件描述符fd指定。 |
| 14 | [os.fdatasync(fd)](http://www.runoob.com/python3/python3-os-fdatasync.html)  强制将文件写入磁盘，该文件由文件描述符fd指定，但是不强制更新文件的状态信息。 |
| 15 | [os.fdopen(fd[, mode[, bufsize]])](http://www.runoob.com/python3/python3-os-fdopen.html)  通过文件描述符 fd 创建一个文件对象，并返回这个文件对象 |
| 16 | [os.fpathconf(fd, name)](http://www.runoob.com/python3/python3-os-fpathconf.html)  返回一个打开的文件的系统配置信息。name为检索的系统配置的值，它也许是一个定义系统值的字符串，这些名字在很多标准中指定（POSIX.1, Unix 95, Unix 98, 和其它）。 |
| 17 | [os.fstat(fd)](http://www.runoob.com/python3/python3-os-fstat.html)  返回文件描述符fd的状态，像stat()。 |
| 18 | [os.fstatvfs(fd)](http://www.runoob.com/python3/python3-os-fstatvfs.html)  返回包含文件描述符fd的文件的文件系统的信息，像 statvfs() |
| 19 | [os.fsync(fd)](http://www.runoob.com/python3/python3-os-fsync.html)  强制将文件描述符为fd的文件写入硬盘。 |
| 20 | [os.ftruncate(fd, length)](http://www.runoob.com/python3/python3-os-ftruncate.html)  裁剪文件描述符fd对应的文件, 所以它最大不能超过文件大小。 |
| 21 | [os.getcwd()](http://www.runoob.com/python3/python3-os-getcwd.html)  返回当前工作目录 |
| 22 | [os.getcwdu()](http://www.runoob.com/python3/python3-os-getcwdu.html)  返回一个当前工作目录的Unicode对象 |
| 23 | [os.isatty(fd)](http://www.runoob.com/python3/python3-os-isatty.html)  如果文件描述符fd是打开的，同时与tty(-like)设备相连，则返回true, 否则False。 |
| 24 | [os.lchflags(path, flags)](http://www.runoob.com/python3/python3-os-lchflags.html)  设置路径的标记为数字标记，类似 chflags()，但是没有软链接 |
| 25 | [os.lchmod(path, mode)](http://www.runoob.com/python3/python3-os-lchmod.html)  修改连接文件权限 |
| 26 | [os.lchown(path, uid, gid)](http://www.runoob.com/python3/python3-os-lchown.html)  更改文件所有者，类似 chown，但是不追踪链接。 |
| 27 | [os.link(src, dst)](http://www.runoob.com/python3/python3-os-link.html)  创建硬链接，名为参数 dst，指向参数 src |
| 28 | [os.listdir(path)](http://www.runoob.com/python3/python3-os-listdir.html)  返回path指定的文件夹包含的文件或文件夹的名字的列表。 |
| 29 | [os.lseek(fd, pos, how)](http://www.runoob.com/python3/python3-os-lseek.html)  设置文件描述符 fd当前位置为pos, how方式修改: SEEK\_SET 或者 0 设置从文件开始的计算的pos; SEEK\_CUR或者 1 则从当前位置计算; os.SEEK\_END或者2则从文件尾部开始. 在unix，Windows中有效 |
| 30 | [os.lstat(path)](http://www.runoob.com/python3/python3-os-lstat.html)  像stat(),但是没有软链接 |
| 31 | [os.major(device)](http://www.runoob.com/python3/python3-os-major.html)  从原始的设备号中提取设备major号码 (使用stat中的st\_dev或者st\_rdev field)。 |
| 32 | [os.makedev(major, minor)](http://www.runoob.com/python3/python3-os-makedev.html)  以major和minor设备号组成一个原始设备号 |
| 33 | [os.makedirs(path[, mode])](http://www.runoob.com/python3/python3-os-makedirs.html)  递归文件夹创建函数。像mkdir(), 但创建的所有intermediate-level文件夹需要包含子文件夹。 |
| 34 | [os.minor(device)](http://www.runoob.com/python3/python3-os-minor.html)  从原始的设备号中提取设备minor号码 (使用stat中的st\_dev或者st\_rdev field )。 |
| 35 | [os.mkdir(path[, mode])](http://www.runoob.com/python3/python3-os-mkdir.html)  以数字mode的mode创建一个名为path的文件夹.默认的 mode 是 0777 (八进制)。 |
| 36 | [os.mkfifo(path[, mode])](http://www.runoob.com/python3/python3-os-mkfifo.html)  创建命名管道，mode 为数字，默认为 0666 (八进制) |
| 37 | [os.mknod(filename[, mode=0600, device])](http://www.runoob.com/python3/python3-os-mknod.html)  创建一个名为filename文件系统节点（文件，设备特别文件或者命名pipe）。 |
| 38 | [os.open(file, flags[, mode])](http://www.runoob.com/python3/python3-os-open.html)  打开一个文件，并且设置需要的打开选项，mode参数是可选的 |
| 39 | [os.openpty()](http://www.runoob.com/python3/python3-os-openpty.html)  打开一个新的伪终端对。返回 pty 和 tty的文件描述符。 |
| 40 | [os.pathconf(path, name)](http://www.runoob.com/python3/python3-os-pathconf.html)  返回相关文件的系统配置信息。 |
| 41 | [os.pipe()](http://www.runoob.com/python3/python3-os-pipe.html)  创建一个管道. 返回一对文件描述符(r, w) 分别为读和写 |
| 42 | [os.popen(command[, mode[, bufsize]])](http://www.runoob.com/python3/python3-os-popen.html)  从一个 command 打开一个管道 |
| 43 | [os.read(fd, n)](http://www.runoob.com/python3/python3-os-read.html)  从文件描述符 fd 中读取最多 n 个字节，返回包含读取字节的字符串，文件描述符 fd对应文件已达到结尾, 返回一个空字符串。 |
| 44 | [os.readlink(path)](http://www.runoob.com/python3/python3-os-readlink.html)  返回软链接所指向的文件 |
| 45 | [os.remove(path)](http://www.runoob.com/python3/python3-os-remove.html)  删除路径为path的文件。如果path 是一个文件夹，将抛出OSError; 查看下面的rmdir()删除一个 directory。 |
| 46 | [os.removedirs(path)](http://www.runoob.com/python3/python3-os-removedirs.html)  递归删除目录。 |
| 47 | [os.rename(src, dst)](http://www.runoob.com/python3/python3-os-rename.html)  重命名文件或目录，从 src 到 dst |
| 48 | [os.renames(old, new)](http://www.runoob.com/python3/python3-os-renames.html)  递归地对目录进行更名，也可以对文件进行更名。 |
| 49 | [os.rmdir(path)](http://www.runoob.com/python3/python3-os-rmdir.html)  删除path指定的空目录，如果目录非空，则抛出一个OSError异常。 |
| 50 | [os.stat(path)](http://www.runoob.com/python3/python3-os-stat.html)  获取path指定的路径的信息，功能等同于C API中的stat()系统调用。 |
| 51 | [os.stat\_float\_times([newvalue])](http://www.runoob.com/python3/python3-os-stat_float_times.html)  决定stat\_result是否以float对象显示时间戳 |
| 52 | [os.statvfs(path)](http://www.runoob.com/python3/python3-os-statvfs.html)  获取指定路径的文件系统统计信息 |
| 53 | [os.symlink(src, dst)](http://www.runoob.com/python3/python3-os-symlink.html)  创建一个软链接 |
| 54 | [os.tcgetpgrp(fd)](http://www.runoob.com/python3/python3-os-tcgetpgrp.html)  返回与终端fd（一个由os.open()返回的打开的文件描述符）关联的进程组 |
| 55 | [os.tcsetpgrp(fd, pg)](http://www.runoob.com/python3/python3-os-tcsetpgrp.html)  设置与终端fd（一个由os.open()返回的打开的文件描述符）关联的进程组为pg。 |
| 56 | [os.tempnam([dir[, prefix]])](http://www.runoob.com/python3/python3-os-tempnam.html)  返回唯一的路径名用于创建临时文件。 |
| 57 | [os.tmpfile()](http://www.runoob.com/python3/python3-os-tmpfile.html)  返回一个打开的模式为(w+b)的文件对象 .这文件对象没有文件夹入口，没有文件描述符，将会自动删除。 |
| 58 | [os.tmpnam()](http://www.runoob.com/python3/python3-os-tmpnam.html)  为创建一个临时文件返回一个唯一的路径 |
| 59 | [os.ttyname(fd)](http://www.runoob.com/python3/python3-os-ttyname.html)  返回一个字符串，它表示与文件描述符fd 关联的终端设备。如果fd 没有与终端设备关联，则引发一个异常。 |
| 60 | [os.unlink(path)](http://www.runoob.com/python3/python3-os-unlink.html)  删除文件路径 |
| 61 | [os.utime(path, times)](http://www.runoob.com/python3/python3-os-utime.html)  返回指定的path文件的访问和修改的时间。 |
| 62 | [os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]])](http://www.runoob.com/python3/python3-os-walk.html)  输出在文件夹中的文件名通过在树中游走，向上或者向下。 |
| 63 | [os.write(fd, str)](http://www.runoob.com/python3/python3-os-write.html)  写入字符串到文件描述符 fd中. 返回实际写入的字符串长度 |

来自 <[*http://www.runoob.com/python3/python3-os-file-methods.html*](http://www.runoob.com/python3/python3-os-file-methods.html)>

异常处理Exception

Friday, September 23, 2016

12:51

两种错误：错误（语法错误）和异常。

* 1. **错误（语法错误）**

Python 的语法错误，也称为解析错。这是程序员“主动”犯的错。常常出现于初学者，

例：

>>> while True print('Hello world')  
 File "<stdin>", line 1, in ? 在文件 File "<stdin>", 第1行的箭头那里  
 while True print('Hello world')  
 ^  
SyntaxError: invalid syntax 语法错误：无效的语法

函数 print() 被检查到有错误，是它前面缺少了一个冒号（:）。语法分析器指出了出错的一行，并且在最先找到的错误的位置标记了一个小小的箭头。

有时候，即使Python程序的语法是正确的，在运行它的时候，也有可能发生错误。有可能是逻辑错误（比如除零错），也有可能是环境导致运行出错。运行期检测到的错误被称为异常。

* 1. **异常**

大多数的异常都不会被程序处理，都以错误信息的形式展现在这里:

>>> 10 \* (1/0)  
Traceback (most recent call last):  
 File "<stdin>", line 1, in ?  
ZeroDivisionError: division by zero  
>>> 4 + spam\*3  
Traceback (most recent call last):  
 File "<stdin>", line 1, in ?  
NameError: name 'spam' is not defined  
>>> '2' + 2  
Traceback (most recent call last):  
 File "<stdin>", line 1, in ?  
TypeError: Can't convert 'int' object to str implicitly

异常以不同的类型出现，这些类型都作为信息的一部分打印出来。以上3个例子，异常类型有 ZeroDivisionError，NameError 和 TypeError。 错误信息的前面部分显示了异常发生的上下文，并以调用栈的形式显示具体信息。

异常的类是Python官方库中的class，所有的异常类，都继承自BaseException及其子类Exception（见文末列表）。异常类可以像其他的类一样做任何事情，但是通常都会比较简单，只提供一些错误相关的属性，并且允许处理异常的代码方便的获取这些信息。

* + **常见的异常处理：**

**try…except…尝试、异常（最常见）**

try:

pass

except BaseException as msg:

print("XXX异常", msg) # 出现异常，报错语句“自定义语句 + 错误信息”

try语句按照如下方式工作；

* + 首先，执行try子句（在关键字try和关键字except之间的语句），如果没有异常发生，try子句执行后结束。忽略except子句，继续执行try…except之后的代码。
  + 如果在执行try子句的过程中发生了异常，那么try子句余下的部分将被忽略。如果异常的类型和 except 之后的名称相符，那么对应的except子句将被执行。如果一个异常没有与任何的except匹配，那么这个异常将会传递给上层的try中。然后执行try…except之后的语句。

就像switch语句，一个 try 语句可能包含多个except子句，分别来处理不同的特定的异常，最多只有一个分支会被执行（对应的异常种类被执行）。处理程序将只针对对应的try子句中的异常进行处理，而不是其他的 try 的处理程序中的异常。而且，就像Switch的case可以合并，多个异常合成元组，一个except子句就可以同时处理多个异常:

except (RuntimeError, TypeError, NameError):  
 print("OS error: {0}".format(err))

最后一个except子句可以忽略异常的名称，它将被当作通配符使用。你可以使用这种方法打印一个错误信息，然后再次把异常抛出。

import sys

try:  
 f = open('myfile.txt')  
 s = f.readline()  
 i = int(s.strip())  
except OSError as err:  
 print("OS error: {0}".format(err))  
except ValueError:  
 print("Could not convert data to an integer.")  
except:  
 print("Unexpected error:", sys.exc\_info()[0])  
 raise

**try…except…else尝试、异常、没有异常继续**

try:

pass

except BaseException as msg:

print("XXX异常", msg)

else: # else只在没有异常的时候执行。

pass

try except 语句还有一个可选的else子句，如果使用这个子句，那么必须放在所有的except子句之后。这个子句将在try子句没有发生任何异常的时候执行。例如:

for arg in sys.argv[1:]:  
 try:  
 f = open(arg, 'r')  
 except IOError:  
 print('cannot open', arg)  
 else:  
 print(arg, 'has', len(f.readlines()), 'lines')  
 f.close()

使用 else 子句比把所有的语句都放在 try 子句里面要好，这样可以避免一些意想不到的、而except又没有捕获的异常。

异常处理并不仅仅处理那些直接发生在try子句中的异常，而且还能处理子句中调用的函数（甚至间接调用的函数）里抛出的异常。例如:

>>> def this\_fails():  
 x = 1/0  
   
>>> try:  
 this\_fails()  
 except ZeroDivisionError as err:  
 print('Handling run-time error:', err)  
   
Handling run-time error: int division or modulo by zero

**try…except…Finally尝试、异常、一定要执行（自定义清理行为）**

try:

pass

except BaseException as msg:

print("XXX异常", msg)

finally: # 不管有没有异常，都执行。

pass

try 语句还有另外一个可选的子句，它定义了无论在任何情况下都会执行的清理行为。 例如:

try:  
 raise KeyboardInterrupt  
finally:  
 print('Goodbye, world!')  
以上例子不管try子句里面有没有发生异常，finally子句都会执行。

如果一个异常在 try 子句里（或者在 except 和 else 子句里）被抛出，而又没有任何的 except 把它截住，那么这个异常会在 finally 子句执行后再次被抛出。

下面是一个更加复杂的例子（在同一个 try 语句里包含 except 和 finally 子句）:

>>> def divide(x, y):  
 try:  
 result = x / y  
 except ZeroDivisionError:  
 print("division by zero!")  
 else:  
 print("result is", result)  
 finally:  
 print("executing finally clause")   
>>> divide(2, 1)  
result is 2.0  
executing finally clause  
>>> divide(2, 0)  
division by zero!  
executing finally clause  
>>> divide("2", "1")  
executing finally clause  
Traceback (most recent call last):  
 File "<stdin>", line 1, in ?  
 File "<stdin>", line 3, in divide  
TypeError: unsupported operand type(s) for /: 'str' and 'str'

* + **预定义的清理行为**

一些对象定义了标准的清理行为，无论系统是否成功的使用了它（成功执行代码），一旦不需要它了，那么这个标准的清理行为就会执行。

for line in open("myfile.txt"): #打开一个文件，然后把内容打印到屏幕上  
 print(line, end="") #当执行完毕后，文件会保持打开状态，并没有被关闭

关键词 with 语句，保证诸如文件之类的对象 在使用完之后一定会执行清理（比如关闭文件），不管处理过程是不是出现问题:

with open("myfile.txt") as f:  
 for line in f:  
 print(line, end="")

* + **抛出异常**

主动抛出异常

if a == b:

raise NameError("发生XXX异常")

if a > b:

raise NameError("发生XXX异常")

Python 使用 raise 语句抛出一个指定的异常。例如:

>>> raise NameError('HiThere')  
Traceback (most recent call last):  
 File "<stdin>", line 1, in ?  
NameError: HiThere

raise 唯一的一个参数指定了要被抛出的异常。它必须是一个异常的实例或者是异常的类（也就是 Exception 的子类）。

如果你只想知道这是否抛出了一个异常，并不想去处理它，那么一个简单的 raise 语句就可以再次把它抛出。

>>> try:  
 raise NameError('HiThere')  
 except NameError:  
 print('An exception flew by!')  
 raise  
   
An exception flew by!  
Traceback (most recent call last):  
 File "<stdin>", line 2, in ?  
NameError: HiThere

* + **用户自定义异常**

你可以通过创建一个新的exception类来拥有自己的异常。异常应该继承自 Exception 类，或者直接继承，或者间接继承，例如:

>>> class MyError(Exception):  
 def \_\_init\_\_(self, value):  
 self.value = value  
 def \_\_str\_\_(self):  
 return repr(self.value)  
   
>>> try:  
 raise MyError(2\*2)  
 except MyError as e:  
 print('My exception occurred, value:', e.value)  
   
My exception occurred, value: 4  
>>> raise MyError('oops!')  
Traceback (most recent call last):  
 File "<stdin>", line 1, in ?  
\_\_main\_\_.MyError: 'oops!'

在这个例子中，类 Exception 默认的 \_\_init\_\_() 被覆盖。

<p异常的类可以像其他的类一样做任何事情，但是通常都会比较简单，只提供一些错误相关的属性，并且允许处理异常的代码方便的获取这些信息。< p="">

当创建一个模块有可能抛出多种不同的异常时，一种通常的做法是为这个包建立一个基础异常类，然后基于这个基础类为不同的错误情况创建不同的子类:

class Error(Exception):  
 """Base class for exceptions in this module."""  
 pass

class InputError(Error):  
 """Exception raised for errors in the input.

Attributes:  
 expression -- input expression in which the error occurred  
 message -- explanation of the error  
 """

def \_\_init\_\_(self, expression, message):  
 self.expression = expression  
 self.message = message

class TransitionError(Error):  
 """Raised when an operation attempts a state transition that's not  
 allowed.

Attributes:  
 previous -- state at beginning of transition  
 next -- attempted new state  
 message -- explanation of why the specific transition is not allowed  
 """

def \_\_init\_\_(self, previous, next, message):  
 self.previous = previous  
 self.next = next  
 self.message = message

大多数的异常的名字都以"Error"结尾，就跟标准的异常命名一样。

来自 <[*http://www.runoob.com/python3/python3-errors-execptions.html*](http://www.runoob.com/python3/python3-errors-execptions.html)>

* + **异常列表**

BaseException  
 +-- SystemExit  
 +-- KeyboardInterrupt  
 +-- GeneratorExit  
 +-- Exception  
 +-- StopIteration  
 +-- ArithmeticError  
 | +-- FloatingPointError  
 | +-- OverflowError  
 | +-- ZeroDivisionError  
 +-- AssertionError  
 +-- AttributeError  
 +-- BufferError  
 +-- EOFError  
 +-- ImportError  
 +-- LookupError  
 | +-- IndexError  
 | +-- KeyError  
 +-- MemoryError  
 +-- NameError  
 | +-- UnboundLocalError  
 +-- OSError  
 | +-- BlockingIOError  
 | +-- ChildProcessError  
 | +-- ConnectionError  
 | | +-- BrokenPipeError  
 | | +-- ConnectionAbortedError  
 | | +-- ConnectionRefusedError  
 | | +-- ConnectionResetError  
 | +-- FileExistsError  
 | +-- FileNotFoundError  
 | +-- InterruptedError  
 | +-- IsADirectoryError  
 | +-- NotADirectoryError  
 | +-- PermissionError  
 | +-- ProcessLookupError  
 | +-- TimeoutError  
 +-- ReferenceError  
 +-- RuntimeError  
 | +-- NotImplementedError  
 +-- SyntaxError  
 | +-- IndentationError  
 | +-- TabError  
 +-- SystemError  
 +-- TypeError  
 +-- ValueError  
 | +-- UnicodeError  
 | +-- UnicodeDecodeError  
 | +-- UnicodeEncodeError  
 | +-- UnicodeTranslateError  
 +-- Warning  
 +-- DeprecationWarning  
 +-- PendingDeprecationWarning  
 +-- RuntimeWarning  
 +-- SyntaxWarning  
 +-- UserWarning  
 +-- FutureWarning  
 +-- ImportWarning  
 +-- UnicodeWarning  
 +-- BytesWarning  
 +-- ResourceWarning

来自 <[*http://www.cnblogs.com/zhangpengshou/p/3565087.html*](http://www.cnblogs.com/zhangpengshou/p/3565087.html)>

if \_\_name\_\_ == "\_\_main\_\_":

Monday, October 31, 2016

19:57

貌似昨天还是前天看过这个~现学现卖一下，有错误的话大家走过路过请指正一下~谢谢！

我的理解是，呜呜为什么我感觉我可以理解但是说不出来。。。我是看下面这段代码懂了的，引用自[python - What does if \_\_name\_\_ == "\_\_main\_\_" do?](http://link.zhihu.com/?target=http%3A//stackoverflow.com/questions/419163/what-does-if-name-main-do)其中[Adam Rosenfield](http://link.zhihu.com/?target=http%3A//stackoverflow.com/users/9530/adam-rosenfield)的回答。

# file one.py  
def func():  
 print("func() in one.py")

print("top-level in one.py")

if \_\_name\_\_ == "\_\_main\_\_":  
 print("one.py is being run directly")  
else:  
 print("one.py is being imported into another module")

# file two.py  
import one

print("top-level in two.py")  
one.func()

if \_\_name\_\_ == "\_\_main\_\_":  
 print("two.py is being run directly")  
else:  
 print("two.py is being imported into another module")

如果你执行one.py文件，

python one.py

会输出：

top-level in one.py  
one.py is being run directly

如果你执行two.py文件，

python two.py

会输出：

top-level in one.py  
one.py is being imported into another module  
top-level in two.py  
func() in one.py  
two.py is being run directly

Thus, when module one gets loaded, its \_\_name\_\_ equals "one" instead of \_\_main\_\_.

我就不翻译了首先我翻译的很丑其次我觉得英文更好理解。。。

它就是上面这么用的。

然后我觉得它的功能就是，**让if \_\_name\_\_ == '\_\_main\_\_'它后面的代码不执行**。这样代码运行会更简洁更流畅？？？因为只需要用想用的那部分就行了。。。

Monday, October 31, 2016

20:07

**Python中：self和\_\_init\_\_的含义 + 为何要有self和\_\_init\_\_**

[2012 年 11 月 27 日 下午 7:34](http://www.crifan.com/summary_the_meaning_of_self_and___init___in_python_and_why_need_them/)[crifan](http://www.crifan.com/author/crifan/)[已有35315人围观](http://www.crifan.com/summary_the_meaning_of_self_and___init___in_python_and_why_need_them/)[5个评论](http://www.crifan.com/summary_the_meaning_of_self_and___init___in_python_and_why_need_them/#comments)

**背景**

回复：

[我写的一些Python教程，需要的可以看看](http://bbs.chinaunix.net/thread-3987682-1-1.html)

中[SongShouJiong](http://bbs.chinaunix.net/space-uid-28397707.html)的提问：

Python中的self,\_\_init\_\_的含义是啥？

为何要有self,\_\_init这些东西？

**解释之前，先说几句**

1.到目前为止，我虽然也已写了不算很少的python的代码，但是，还真的没有太多接触self和\_\_init\_\_这两个东西。

只能算是大概了解。

2.为了搞懂其含义，现学现卖，去看了些资料，然后整理出来。

其中，很可能理解有误，还请高手指正。

3.解释含义之前，先说说对于学习Python中的self和\_\_init\_\_等内容的态度：

个人的建议是，很多东西，等你有了需求了，再去使用，该东西才算有价值，你也才会更容易理解；

否则，暂时接触不到的话，其实可以将其放到一边，等以后遇到了再说。

否则，现在就去努力的学，态度虽好，但是由于没有相关的编程实践，即没有足够的代码的练习，对于概念的理解，必然很难深入。

因此，对于Python中的self和\_\_init\_\_的建议是：

如果你去写Python代码，去学习Python的过程中，在没有遇到之前，

比如像我这种，一直就没怎么编写过真正的类（Class），那么倒也是可以暂时不用太理会这些东西；

等以后需要编写类的时候，再去详细学习，了解，搞懂其含义。

4.下面的内容，算是为你，的确遇到了self，\_\_init\_\_，的时候，需要理解其含义的时候，提供一些参考。

5.在看下面的解释之前，你首先要对类（Class）和类的实例化的对象（Object，也称实例Instance）的概念，有所了解。

如果不了解，可以参考我总结的：

[【整理】面向对象的基础知识：类Class，对象（Object），实例（Instance）](http://www.crifan.com/summary_the_basic_knowledge_about_object_oriented)

**python中self和\_\_init\_\_的含义 + 为何要有self和\_\_init\_\_**

**Python中的self**

在Python中的类Class的代码中，常看到函数中的第一个参数，都是self。

以及Class中的函数里面，访问对应的变量（读取或者写入），以及调用对应的函数时，都是

self.valueName

self.function()

的形式。

下面就来解释一下self的含义：

**Python中self的含义**

self，英文单词意思很明显，表示自己，本身。

此处有几种潜在含义：

1.这里的自己，指的是，实例Instance本身。

2.同时， 由于说到“自己”这个词，都是和相对而言的“其他”而说的。

而此处的其他，指的是，类Class，和其他变量，比如局部变量，全局变量等。

此处的self，是个对象，Object。

是当前类的实例。

因此，对应的

self.valueName

self.function()

中的

valueName：表示self对象，即实例的变量。与其他的，Class的变量，全局的变量，局部的变量，是相对应的。

function：表示是调用的是self对象，即实例的函数。与其他的全局的函数，是相对应的。

**Python中为何要有self**

上面其实已经基本上说清楚了，为何需要有self。

那就是：

在类的代码（函数）中，需要访问当前的实例中的变量和函数的，即，访问Instance中的：

* 对应的变量（property)：Instance.ProperyNam，去读取之前的值和写入新的值
* 调用对应函数（function）：Instance.function()，即执行对应的动作

-> 而需要访问实例的变量和调用实例的函数，当然需要对应的实例Instance对象本身

-> 而Python中就规定好了，函数的第一个参数，就必须是实例对象本身，并且建议，约定俗成，把其名字写为self

-> 所以，我们需要self（需要用到self）

而如果没有用到self，即代码中，去掉self后，那种写法所使用到的变量，实际上不是你所希望的，不是真正的实例中的变量和函数，而是的访问到了其他部分的变量和函数了。

甚至会由于没有合适的初始化实例变量，而导致后续无法访问的错误。

下面，就通过代码，来演示，如果去掉self，或者没有合理的使用self，会出现哪些错误。

**如果没有在\_\_init\_\_中初始化对应的实例变量的话，导致后续引用实例变量会出错**

如下代码，完整的演示了，如果没有在类Class的最初的\_\_init\_\_函数中，正确的初始化实例变量，则会导致后续没有变量可用，因而出现AttributeError的错误：

[?](http://www.crifan.com/summary_the_meaning_of_self_and___init___in_python_and_why_need_them/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | #!/usr/bin/python  # -\*- coding: utf-8 -\*-  """  -------------------------------------------------------------------------------  Function:  【整理】Python中：self和init\_\_的含义 + 为何要有self和\_\_init\_\_  <http://www.crifan.com/summary_the_meaning_of_self_and___init___in_python_and_why_need_them>    Author:     Crifan  Verison:    2012-11-27  -------------------------------------------------------------------------------  """    #注：此处全局的变量名，写成name，只是为了演示而用  #实际上，好的编程风格，应该写成gName之类的名字，以表示该变量是Global的变量  name = "whole global name";    class Person:      def \_\_init\_\_(self, newPersionName):          #self.name = newPersionName;            #1.如果此处不写成self.name          #那么此处的name，只是\_\_init\_\_函数中的局部临时变量name而已          #和全局中的name，没有半毛钱关系          name = newPersionName;            #此处只是为了代码演示，而使用了局部变量name，          #不过需要注意的是，此处很明显，由于接下来的代码也没有利用到此处的局部变量name          #则就导致了，此处的name变量，实际上被浪费了，根本没有利用到        def sayYourName(self):          #此处由于找不到实例中的name变量，所以会报错：          #AttributeError: Person instance has no attribute 'name'          print 'My name is %s'%(self.name);    def selfAndInitDemo():      persionInstance = Person("crifan");      persionInstance.sayYourName();    ###############################################################################  if \_\_name\_\_=="\_\_main\_\_":      selfAndInitDemo(); |

从上述代码可见，由于在类的初始化（实例化）的\_\_init\_\_函数中，没有给self.name设置值，使得实例中，根本没有name这个变量，导致后续再去访问self.name，就会出现AttributeError的错误了。

对应的，如果写成self.name，则意思就正确了，就是初始化的时候，给实例中新增加，并且正常设置了正确的值newPersionName了，所以后续再去通过self.name，就可以访问到，当前实例中正确的变量name了。

相应的正确写法的代码如下：

[?](http://www.crifan.com/summary_the_meaning_of_self_and___init___in_python_and_why_need_them/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | #!/usr/bin/python  # -\*- coding: utf-8 -\*-  """  -------------------------------------------------------------------------------  Function:  【整理】Python中：self和init\_\_的含义 + 为何要有self和\_\_init\_\_  <http://www.crifan.com/summary_the_meaning_of_self_and___init___in_python_and_why_need_them>    Author:     Crifan  Verison:    2012-11-27  -------------------------------------------------------------------------------  """    #注：此处全局的变量名，写成name，只是为了演示而用  #实际上，好的编程风格，应该写成gName之类的名字，以表示该变量是Global的变量  name = "whole global name";    class Person:      def \_\_init\_\_(self, newPersionName):          #此处正确的，通过访问self.name的形式，实现了：          #1.给实例中，增加了name变量          #2.并且给name赋了初值，为newPersionName          self.name = newPersionName;        def sayYourName(self):          #此处由于开始正确的初始化了self对象，使得其中有了name变量，所以此处可以正确访问了name值了，可以正确的输出了：          #My name is crifan          print 'My name is %s'%(self.name);    def selfAndInitDemo():      persionInstance = Person("crifan");      persionInstance.sayYourName();    ###############################################################################  if \_\_name\_\_=="\_\_main\_\_":      selfAndInitDemo(); |

**在函数中，使用对应的变量，虽然代码是可以运行的，但是实际上却是使用的，不是实例中的变量**

有时候，虽然你写的代码，可以运行，但是使用到的变量，由于没有加self，实际上是用到的不是实例的变量，而是其他的变量。

此类问题，主要和Python中的变量的作用域有关，但是此处例子中，也和是否使用self有关：

[?](http://www.crifan.com/summary_the_meaning_of_self_and___init___in_python_and_why_need_them/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | #!/usr/bin/python  # -\*- coding: utf-8 -\*-  """  -------------------------------------------------------------------------------  Function:  【整理】Python中：self和init\_\_的含义 + 为何要有self和\_\_init\_\_  <http://www.crifan.com/summary_the_meaning_of_self_and___init___in_python_and_why_need_them>    Author:     Crifan  Verison:    2012-11-27  -------------------------------------------------------------------------------  """    #注：此处全局的变量名，写成name，只是为了演示而用  #实际上，好的编程风格，应该写成gName之类的名字，以表示该变量是Global的变量  name = "whole global name";    class Person:      name = "class global name"        def \_\_init\_\_(self, newPersionName):          #self.name = newPersionName;            #此处，没有使用self.name          #而使得此处的name，实际上仍是局部变量name          #虽然此处赋值了，但是后面没有被利用到，属于被浪费了的局部变量name          name = newPersionName;        def sayYourName(self):          #此处，之所以没有像之前一样出现：          #AttributeError: Person instance has no attribute 'name'          #那是因为，虽然当前的实例self中，没有在\_\_init\_\_中初始化对应的name变量，实例self中没有对应的name变量          #但是由于实例所对应的类Person，有对应的name变量,所以也是可以正常执行代码的          #对应的，此处的self.name，实际上是Person.name          print 'My name is %s'%(self.name); # -> class global name          print 'name within class Person is actually the global name: %s'%(name); #-> whole global name          print "only access Person's name via Person.name=%s"%(Person.name); # -> class global name    def selfAndInitDemo():      persionInstance = Person("crifan");      persionInstance.sayYourName();      print "whole global name is %s"%(name); # -> whole global name    ###############################################################################  if \_\_name\_\_=="\_\_main\_\_":      selfAndInitDemo(); |

其中，可见，此处开始\_\_init\_\_中，没有给self实例初始化对应的name

而后面的函数sayYourName中，虽然可以调用到self.name而没有出现AttributeError错误

但是实际上此处的值，不是所期望的，传入的name，即"crifan"，而是类中的name的值，即"class global name"。

关于Python中变量的作用域的详细解释，可参考：

[【整理】Python中变量的作用域(variable scope)](http://www.crifan.com/summary_python_variable_effective_scope/)

**Python中的\_\_init\_\_**

Python中，常会看到，很多类中，都有一个\_\_init\_\_函数。

下面就来解释一下，\_\_init\_\_的含义。

**Python中\_\_init\_\_的含义**

首先来看init单词本身，其含义是 初始化。

而\_\_init\_\_的含义，也如同其单词本意一样，用来初始化。

但是给谁初始化呢？

很明显，是给对应的类Class本身，去初始化。

**Python中为何要有\_\_init\_\_**

我的理解是，\_\_init\_\_的出现，主要有两方面的作用：

一般常见的初始化，我的理解，可能主要有两方面：

**支持带参数的类的初始化**

这个用法，感觉就像，其他语言中的，对于Class初始化时，可以运行传递不同的参数一样；

用代码演示如下：

[?](http://www.crifan.com/summary_the_meaning_of_self_and___init___in_python_and_why_need_them/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | #!/usr/bin/python  # -\*- coding: utf-8 -\*-  """  -------------------------------------------------------------------------------  Function:  【整理】Python中：self和init\_\_的含义 + 为何要有self和\_\_init\_\_  <http://www.crifan.com/summary_the_meaning_of_self_and___init___in_python_and_why_need_them>    Author:     Crifan  Verison:    2012-11-27  -------------------------------------------------------------------------------  """    class Person:      def \_\_init\_\_(self, newPersionName):          #在开始初始化新的类Class的示例Instance的时候，给对应的，不同的Instance，设置不同的人名（Person name）          self.name = newPersionName;        def sayYourName(self):          #不同的Person的示例，调用同样的方法的时候，说出自己的名字，结果都是对应着自己的，各自不同的名字          print 'My name is %s'%(self.name); #My name is crifan    def initDemo():      persionInstance = Person("crifan");      persionInstance.sayYourName();    ###############################################################################  if \_\_name\_\_=="\_\_main\_\_":      initDemo(); |

其中，就是针对Person这个类，

不同的示例，在初始化的时候，都传递一个对应的参数，这样不同的Person，就都有了自己的不同的名字了。

这个，至少看起来，有点类似于，其他语言中的，传递特定参数去对类进行初始化。

**实现类本身相关内容的初始化**

当一个Class，稍微复杂一点的时候，或者内部函数需要用得到的时候，往往都需要在，别人实例化你这个类之前，使用你这个类之前，做一些基本的，与自己的类有关的，初始化方面的工作。

而这部分工作，往往就放到\_\_init\_\_函数中去了。

换句话说，你要用人家的类（中的变量和函数）之前，总要给人家一个机会，做点准备工作，然后才能为你服务吧。

这个概念，相对还是很好理解的，就不多赘述。

演示代码就不用了，因为上面的代码，也可以算是这方面的例子了，不同的人，应该有不同的名字，而给不同的人设置不同的名字，也是需要在调用sayYourName之前，就先初始化好的。

**参考资料**

1. [使用\_\_init\_\_方法](http://sebug.net/paper/python/ch11s05.html)

2. [了解何时去使用 self 和 \_\_init\_\_](http://woodpecker.org.cn/diveintopython/object_oriented_framework/defining_classes.html#d0e12045)

3.[Python 'self' explained](http://stackoverflow.com/questions/2709821/python-self-explained)

4. [Python \_\_init\_\_ and self what do they do?](http://stackoverflow.com/questions/625083/python-init-and-self-what-do-they-do)

来自 <[*http://www.crifan.com/summary\_the\_meaning\_of\_self\_and\_\_\_init\_\_\_in\_python\_and\_why\_need\_them/*](http://www.crifan.com/summary_the_meaning_of_self_and___init___in_python_and_why_need_them/)>

Monday, October 31, 2016

20:09

[**Python为什么要self**](http://www.cnblogs.com/wangkangluo1/archive/2011/09/23/2186479.html)

原文地址：<http://sjolzy.cn/Why-should-self-Python.html>

接触Python以来，看到类里的函数要带个self参数，一直搞不懂啥麻子原因。晚上特别针对Python的self查了一下，理理。

**Python要self的理由**

Python的类的方法和普通的函数有一个很明显的区别，在类的方法必须有个额外的第一个参数 (**self** )，但在调用这个方法的时候不必为这个参数赋值 （显胜于隐 的引发）。Python的类的方法的这个特别的参数指代的是对象本身。

**为何Python给self赋值而你不必给self赋值？**

例子说明：创建了一个类MyClass，实例化MyClass得到了MyObject这个对象，然后调用这个对象的方法MyObject.method(arg1,arg2) ，这个过程中，Python会自动转为Myclass.mehod(MyObject,arg1,arg2)

这就是Python的self的原理了。即使你的类的方法不需要任何参数，但还是得给这个方法定义一个self参数，虽然我们在实例化调用的时候不用理会这个参数不用给它赋值。

实例：

class Python:

 def selfDemo(self):

  print 'Python,why self?'

p = Python()

p.selfDemo()

输出：Python,why self?

把p.selfDemo()带个参数如：p.selfDemo(p)，得到同样的输出结果

如果把self去掉的话，

class Python:

 def selfDemo():

  print 'Python,why self?'

p = Python()

p.selfDemo()

这样就报错了：**TypeError: selfDemo() takes no arguments (1 given)**

**扩展**

self在Python里不是关键字。self代表当前对象的地址。self能避免非限定调用造成的全局变量。

来自 <[*http://www.cnblogs.com/wangkangluo1/archive/2011/09/23/2186479.html*](http://www.cnblogs.com/wangkangluo1/archive/2011/09/23/2186479.html)>

面向对象

Tuesday, November 1, 2016

13:56

面向过程编程：按照功能需求的顺序来设计，写出的程序按照代码循序运行。

面向对象编程：把数据和对数据的操作用一种叫做“对象”的东西包裹起来。这种方法更适合大型程序开发。

面向对象编程最主要的两个概念就是：类（class）和对象（object）。类是一种抽象的类型，而对象是这种类型的实例。一个类可以有属于它的函数，这种函数被称为类的“方法”。一个类/对象可以有属于它的变量，这种变量被称作“域”。域根据所属不同，又分别被称作“类变量”和“实例变量”。域和方法被合称为类的属性。

继续笔的例子。一个笔有书写的功能，所以“书写”就是笔这个类的一种方法。每支笔有自己的颜色，“颜色”就是某支笔的域，也是这支笔的实例变量。而关于“类变量”，我们假设有一种限量版钢笔，我们为这种笔创建一种类。而这种笔的“产量”就可以看做这种笔的类变量。因为这个域不属于某一支笔，而是这种类型的笔的共有属性。

python是一种高度面向对象的语言，它其中的所有东西其实都是对象。在Python中，不论是数值（整型、浮点型），字符串，字典，元组对象，还是他们所对应的类型，以及函数，模块等你所能看到的都是对象，他们的祖先是PyObject。而Java中至少函数，基本数据类型都不算对象。

s = 'how are you' #s被赋值后就是一个字符串类型的对象

l = s.split() #split是字符串的方法，这个方法返回一个list类型的对象，l是一个list类型的对象

dir(s)显示这种对象有什么属性和方法

type()显示对象的类型

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000196&itemidx=1&sign=2176872b0086314a9463ccc811731cc4*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000196&itemidx=1&sign=2176872b0086314a9463ccc811731cc4)>

class MyClass: #关键字class加上类名用来创建一个类，之后缩进的代码块是这个类的内部。

    pass #pass语句，表示一个空的代码块

mc = MyClass() #创建一个类的实例，也就是对象。然后赋值给变量mc。mc是一个MyClass类的对象。

print mc # <\_\_main\_\_.MyClass instance at 0x7fd1c8d01200>表示：mc是\_\_main\_\_模块中MyClass来的一个实例（instance），后面的一串十六进制的数字是这个对象的内存地址。

我们给这个类加上一些域：

class MyClass:

    name = 'Sam' # 增加了一个类变量name，并把它的值设为'Sam'

    def sayHi(self): # 增加了一个类方法sayHi。类方法和普通函数的区别，第一个参数必须为self

        print 'Hello %s' % self.name

mc = MyClass()

print mc.name

mc.name = 'Lily' #调用类变量的方法是“对象.变量名”。你可以得到它的值，也可以改变它的值

mc.sayHi() # 通过“对象.方法名()”格式进行调用类方法，不需要提供self的值。因为，self在类方法中的值，就是调用的这个对象本身。

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000199&itemidx=1&sign=0e85961e537d8e1c8a6d65568ef7be54*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000199&itemidx=1&sign=0e85961e537d8e1c8a6d65568ef7be54)>

面向对象是比较复杂的概念，初学很难理解。一般人最先学习面向过程，从上到下一行行执行的简单程序容易被理解，即使加上if、while、for之类的语句以及函数调用，也还是不算困难。有了面向对象之后，程序的执行路径就变得复杂，很容易让人混乱。不过当你熟悉之后会发现，面向对象是比面向过程更合理的程序设计方式。

面向过程把数据和处理数据的计算全部放在一起，当功能复杂之后，就会显得很混乱，且容易产生很多重复的代码。而面向对象，把一类数据和处理这类数据的方法封装在一个类中，让程序的结构更清晰，不同的功能之间相互独立。**这样更有利于进行模块化的开发方式。**

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000202&itemidx=1&sign=bd57c81f02acd50318b66abe2b9bc120*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000202&itemidx=1&sign=bd57c81f02acd50318b66abe2b9bc120)>

Vehicle类被称为基本类或超类（Java称作父类），Car类和Bike类被成为导出类或子类。

class Vehicle:

    def \_\_init\_\_(self, speed):

        self.speed = speed

    def drive(self, distance):

        print 'need %f hour(s)' % (distance / self.speed)

class Bike(Vehicle):

    pass

class Car(Vehicle):

    def \_\_init\_\_(self, speed, fuel):

        Vehicle.\_\_init\_\_(self, speed)

        self.fuel = fuel

    def drive(self, distance):

        Vehicle.drive(self, distance)

        print 'need %f fuels' % (distance \* self.fuel)

b = Bike(15.0)

c = Car(80.0, 0.012)

b.drive(100.0)

c.drive(100.0)

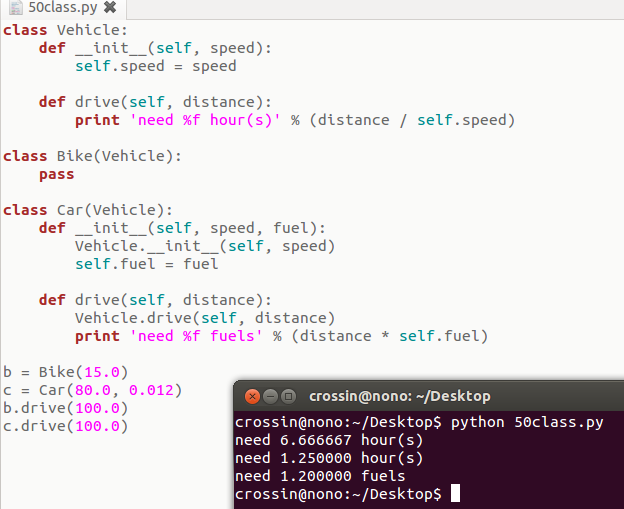
解释一下代码：

\_\_init\_\_函数会在类被创建的时候自动调用，用来初始化类。它的参数，要在创建类的时候提供。于是我们通过提供一个数值来初始化speed的值。

class定义后面的括号里表示这个类继承于哪个类。Bike(Vehicle)就是说Bike是继承自Vehicle中的子类。Vehicle中的属性和方法，Bike都会有。因为Bike不需要有额外的功能，所以用pass在类中保留空块，什么都不用写。

Car类中，我们又重新定义了\_\_init\_\_和drive函数，这样会覆盖掉它继承自Vehicle的同名函数。但我们依然可以通过“Vehicle.函数名”来调用它的超类方法。以此来获得它作为Vehicle所具有的功能。注意，因为是通过类名调用方法，而不是像之前一样通过对象来调用，所以这里必须提供self的参数值。在调用超类的方法之后，我们又给Car增加了一个fuel属性，并且在drive中多输出一行信息。

最后，我们分别创建一个速度为15的自行车对象，和一个速度为80、耗油量为0.012的汽车，然后让它们去行驶100的距离。



#==== Crossin的编程教室 ====#

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000209&itemidx=1&sign=e05cb29e0272f0a04aeeaede3d2272a6*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000209&itemidx=1&sign=e05cb29e0272f0a04aeeaede3d2272a6)>

Tuesday, November 1, 2016

14:09

**and-or技巧**

*2015-12-21* [Crossin的编程教室](javascript:void(0);)



今天介绍一个python中的小技巧：and-or

看下面这段代码：

a = "heaven"

b = "hell"

c = True and a or b

print c

d = False and a or b

print d

输出：

heaven

hell

结果很奇怪是不是？

表达式从左往右运算，1和"heaven"做and的结果是"heaven"，再与"hell"做or的结果是"heaven"；0和"heaven"做and的结果是0，再与"hell"做or的结果是"hell"。

抛开绕人的and和or的逻辑，你只需记住，在一个bool and a or b语句中，当bool条件为真时，结果是a；当bool条件为假时，结果是b。

有学过c/c++的同学应该会发现，这和bool?a:b表达式很像。

有了它，原本需要一个if-else语句表述的逻辑：

if a > 0:

print "big"

else:

print "small"

就可以直接写成：

print (a > 0) and "big" or "small"

然而不幸的是，如果直接这么用，有一天你会踩到坑的。和c语言中的?:表达式不同，这里的and or语句是利用了python中的逻辑运算实现的。当a本身是个假值（如0，""）时，结果就不会像你期望的那样。

比如：

a = ""

b = "hell"

c = True and a or b

print c

得到的结果不是""而是"hell"。因为""和"hell"做or的结果是"hell"。

所以，and-or真正的技巧在于，确保a的值不会为假。最常用的方式是使 a 成为 [a] 、 b 成为 [b]，然后使用返回值列表的第一个元素：

a = ""

b = "hell"

c = (True and [a] or [b])[0]

print c

由于[a]是一个非空列表，所以它决不会为假。即使a是0或者''或者其它假值，列表[a]也为真，因为它有一个元素。

在两个常量值进行选择时，and-or会让你的代码更简单。但如果你觉得这个技巧带来的副作用已经让你头大了，没关系，用if-else可以做相同的事情。不过在python的某些情况下，你可能没法使用if语句，比如lambda函数中，这时候你可能就需要and-or的帮助了。

什么是lambda函数？呵呵，这是python的高阶玩法，暂且按住不表，以后有机会再说。

来自 <[*http://mp.weixin.qq.com/mp/appmsg/show?\_\_biz=MjM5MDEyMDk4Mw==&appmsgid=10000232&itemidx=1&sign=5dcb973d248434af133506d732dd426d*](http://mp.weixin.qq.com/mp/appmsg/show?__biz=MjM5MDEyMDk4Mw==&appmsgid=10000232&itemidx=1&sign=5dcb973d248434af133506d732dd426d)>

Monday, November 21, 2016

18:33

[**判断异常类型，全面捕获Python的异常**](http://blog.csdn.net/xiaoxiaoluo/article/details/7928891)

标签： [python](http://www.csdn.net/tag/python)[file](http://www.csdn.net/tag/file)[c](http://www.csdn.net/tag/c)

2012-08-31 13:43 2439人阅读 [评论](http://blog.csdn.net/xiaoxiaoluo/article/details/7928891#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/xiaoxiaoluo/article/details/7928891#report)

C:\75815A85\346EA69A-A92A-4C28-A2E0-DACAA0096A53.files\image010.jpg

分类：

python*（1）*

C:\75815A85\346EA69A-A92A-4C28-A2E0-DACAA0096A53.files\image011.jpg

<http://ar.newsmth.net/thread-20bf9f3e95cd35-1.html>

正文：

    如果你在写程序时遇到异常后想进行如下处理的话请参考我下面写的对异常处理的方法：

假设有下面的一段程序：

try:

    语句1

    语句2

    .

    语句N

except .........:

    print .......

    但是你并不知道“语句1至语句N”在执行会出什么样的异常，但你还要做异常处理，且想把出现的异常打印出来，并不停止程序的运行，

所以在“except ......”这句应怎样来写呢？

至少3个方法：

方法一：捕获所有异常

--------------------------------------------------------------------------------

code:

try:

    a=b

    b=c

except Exception,ex:

    print Exception,":",ex

--------------------------------------------------------------------------------

方法二：采用traceback模块查看异常

--------------------------------------------------------------------------------

code:

import traceback

try:

    a=b

    b=c

except:

    traceback.print\_exc()

----------------------------------------------------------------------------

方法三：采用sys模块回溯最后的异常

----------------------------------------------------------------------------

code:

import sys

try:

    a=b

    b=c

except:

    info=sys.exc\_info()

    print info[0],":",info[1]

--------------------------------------------------------------------------------

但是，如果你还想把这些异常保存到一个日志文件中，来分析这些异常，那么请看下面的方法：

把　traceback.print\_exc()　打印在屏幕上的信息保存到一个文本文件中

code:

try:

    a=b

    b=c

except:

    f=open("c:\\log.txt",'a')

    traceback.print\_exc(file=f)

    f.flush()

    f.close()

来自 <[*http://blog.csdn.net/xiaoxiaoluo/article/details/7928891*](http://blog.csdn.net/xiaoxiaoluo/article/details/7928891)>

Sunday, November 27, 2016

22:42

**Python yield 使用浅析**

[廖 雪峰](http://www.ibm.com/developerworks/cn/opensource/os-cn-python-yield/#authorN1001E), 软件工程师, HP

初学 Python 的开发者经常会发现很多 Python 函数中用到了 yield 关键字，然而，带有 yield 的函数执行流程却和普通函数不一样，yield 到底用来做什么，为什么要设计 yield ？本文将由浅入深地讲解 yield 的概念和用法，帮助读者体会 Python 里 yield 简单而强大的功能。

**带有 yield 的函数在 Python 中被称之为 generator（生成器）**

我们先抛开 generator，以一个常见的编程题目来展示 yield 的概念。

**如何生成斐波那契數列**

斐波那契（Fibonacci）數列是一个非常简单的递归数列，除第一个和第二个数外，任意一个数都可由前两个数相加得到。用计算机程序输出斐波那契數列的前 N 个数是一个非常简单的问题，许多初学者都可以轻易写出如下函数：

**清单 1. 简单输出斐波那契數列前 N 个数**

def fab(max):   
 n, a, b = 0, 0, 1   
 while n < max:   
 print b   
 a, b = b, a + b   
 n = n + 1

执行 fab(5)，我们可以得到如下输出：

>>> fab(5)   
 1   
 1   
 2   
 3   
 5

结果没有问题，但有经验的开发者会指出，直接在 fab 函数中用 print 打印数字会导致该函数可复用性较差，因为 fab 函数返回 None，其他函数无法获得该函数生成的数列。

要提高 fab 函数的可复用性，最好不要直接打印出数列，而是返回一个 List。以下是 fab 函数改写后的第二个版本：

**清单 2. 输出斐波那契數列前 N 个数第二版**

def fab(max):   
 n, a, b = 0, 0, 1   
 L = []   
 while n < max:   
 L.append(b)   
 a, b = b, a + b   
 n = n + 1   
 return L

可以使用如下方式打印出 fab 函数返回的 List：

>>> for n in fab(5):   
 ... print n   
 ...   
 1   
 1   
 2   
 3   
 5

改写后的 fab 函数通过返回 List 能满足复用性的要求，但是更有经验的开发者会指出，该函数在运行中占用的内存会随着参数 max 的增大而增大，如果要控制内存占用，最好不要用 List

来保存中间结果，而是通过 iterable 对象来迭代。例如，在 Python2.x 中，代码：

**清单 3. 通过 iterable 对象来迭代**

for i in range(1000): pass

会导致生成一个 1000 个元素的 List，而代码：

for i in xrange(1000): pass

则不会生成一个 1000 个元素的 List，而是在每次迭代中返回下一个数值，内存空间占用很小。因为 xrange 不返回 List，而是返回一个 iterable 对象。

利用 iterable 我们可以把 fab 函数改写为一个支持 iterable 的 class，以下是第三个版本的 Fab：

**清单 4. 第三个版本**

class Fab(object):

def \_\_init\_\_(self, max):   
 self.max = max   
 self.n, self.a, self.b = 0, 0, 1

def \_\_iter\_\_(self):   
 return self

def next(self):   
 if self.n < self.max:   
 r = self.b   
 self.a, self.b = self.b, self.a + self.b   
 self.n = self.n + 1   
 return r   
 raise StopIteration()

Fab 类通过 next() 不断返回数列的下一个数，内存占用始终为常数：

>>> for n in Fab(5):   
 ... print n   
 ...   
 1   
 1   
 2   
 3   
 5

然而，使用 class 改写的这个版本，代码远远没有第一版的 fab 函数来得简洁。如果我们想要保持第一版 fab 函数的简洁性，同时又要获得 iterable 的效果，yield 就派上用场了：

**清单 5. 使用 yield 的第四版**

def fab(max):   
 n, a, b = 0, 0, 1   
 while n < max:   
 yield b   
 # print b   
 a, b = b, a + b   
 n = n + 1

'''

第四个版本的 fab 和第一版相比，仅仅把 print b 改为了 yield b，就在保持简洁性的同时获得了 iterable 的效果。

调用第四版的 fab 和第二版的 fab 完全一致：

>>> for n in fab(5):   
 ... print n   
 ...   
 1   
 1   
 2   
 3   
 5

简单地讲，yield 的作用就是把一个函数变成一个 generator，带有 yield 的函数不再是一个普通函数，Python 解释器会将其视为一个 generator，调用 fab(5) 不会执行 fab 函数，而是返回一个 iterable 对象！在 for 循环执行时，每次循环都会执行 fab 函数内部的代码，执行到 yield b 时，fab 函数就返回一个迭代值，下次迭代时，代码从 yield b 的下一条语句继续执行，而函数的本地变量看起来和上次中断执行前是完全一样的，于是函数继续执行，直到再次遇到 yield。

也可以手动调用 fab(5) 的 next() 方法（因为 fab(5) 是一个 generator 对象，该对象具有 next() 方法），这样我们就可以更清楚地看到 fab 的执行流程：

**清单 6. 执行流程**

>>> f = fab(5)   
 >>> f.next()   
 1   
 >>> f.next()   
 1   
 >>> f.next()   
 2   
 >>> f.next()   
 3   
 >>> f.next()   
 5   
 >>> f.next()   
 Traceback (most recent call last):   
 File "<stdin>", line 1, in <module>   
 StopIteration

当函数执行结束时，generator 自动抛出 StopIteration 异常，表示迭代完成。在 for 循环里，无需处理 StopIteration 异常，循环会正常结束。

我们可以得出以下结论：

一 个带有 yield 的函数就是一个 generator，它和普通函数不同，生成一个 generator 看起来像函数调用，但不会执行任何函数代码，直到对其调用 next()（在 for 循环中会自动调用 next()）才开始执行。虽然执行流程仍按函数的流程执行，但每执行到一个 yield 语句就会中断，并返回一个迭代值，下次执行时从 yield 的下一个语句继续执行。看起来就好像一个函数在正常执行的过程中被 yield 中断了数次，每次中断都会通过 yield 返回当前的迭代值。

yield 的好处是显而易见的，把一个函数改写为一个 generator 就获得了迭代能力，比起用类的实例保存状态来计算下一个 next() 的值，不仅代码简洁，而且执行流程异常清晰。

如何判断一个函数是否是一个特殊的 generator 函数？可以利用 isgeneratorfunction 判断：

**清单 7. 使用 isgeneratorfunction 判断**

>>> from inspect import isgeneratorfunction   
 >>> isgeneratorfunction(fab)   
 True

要注意区分 fab 和 fab(5)，fab 是一个 generator function，而 fab(5) 是调用 fab 返回的一个 generator，好比类的定义和类的实例的区别：

**清单 8. 类的定义和类的实例**

>>> import types   
 >>> isinstance(fab, types.GeneratorType)   
 False   
 >>> isinstance(fab(5), types.GeneratorType)   
 True

fab 是无法迭代的，而 fab(5) 是可迭代的：

>>> from collections import Iterable   
 >>> isinstance(fab, Iterable)   
 False   
 >>> isinstance(fab(5), Iterable)   
 True

每次调用 fab 函数都会生成一个新的 generator 实例，各实例互不影响：

>>> f1 = fab(3)   
 >>> f2 = fab(5)   
 >>> print 'f1:', f1.next()   
 f1: 1   
 >>> print 'f2:', f2.next()   
 f2: 1   
 >>> print 'f1:', f1.next()   
 f1: 1   
 >>> print 'f2:', f2.next()   
 f2: 1   
 >>> print 'f1:', f1.next()   
 f1: 2   
 >>> print 'f2:', f2.next()   
 f2: 2   
 >>> print 'f2:', f2.next()   
 f2: 3   
 >>> print 'f2:', f2.next()   
 f2: 5

[回页首](http://www.ibm.com/developerworks/cn/opensource/os-cn-python-yield/#ibm-pcon)

**return 的作用**

在一个 generator function 中，如果没有 return，则默认执行至函数完毕，如果在执行过程中 return，则直接抛出 StopIteration 终止迭代。

[回页首](http://www.ibm.com/developerworks/cn/opensource/os-cn-python-yield/#ibm-pcon)

**另一个例子**

另一个 yield 的例子来源于文件读取。如果直接对文件对象调用 read() 方法，会导致不可预测的内存占用。好的方法是利用固定长度的缓冲区来不断读取文件内容。通过 yield，我们不再需要编写读文件的迭代类，就可以轻松实现文件读取：

**清单 9. 另一个 yield 的例子**

def read\_file(fpath):   
 BLOCK\_SIZE = 1024   
 with open(fpath, 'rb') as f:   
 while True:   
 block = f.read(BLOCK\_SIZE)   
 if block:   
 yield block   
 else:   
 return

以上仅仅简单介绍了 yield 的基本概念和用法，yield 在 Python 3 中还有更强大的用法，我们会在后续文章中讨论。

注：本文的代码均在 Python 2.7 中调试通过

来自 <[*http://www.ibm.com/developerworks/cn/opensource/os-cn-python-yield/*](http://www.ibm.com/developerworks/cn/opensource/os-cn-python-yield/)>

Monday, November 28, 2016

01:28

[**Python：使用Scrapy框架的ImagesPipeline下载图片如何保持原图片名称呢？**](http://blog.csdn.net/php_fly/article/details/19688595)

标签： [python](http://www.csdn.net/tag/python)[scrapy](http://www.csdn.net/tag/scrapy)[ImagePipeline](http://www.csdn.net/tag/ImagePipeline)[image\_key](http://www.csdn.net/tag/image_key)[file\_path](http://www.csdn.net/tag/file_path)

2014-02-22 16:18 3357人阅读 [评论](http://blog.csdn.net/php_fly/article/details/19688595#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/php_fly/article/details/19688595#report)

C:\75815A85\346EA69A-A92A-4C28-A2E0-DACAA0096A53.files\image010.jpg

分类：

Python*（5）*

C:\75815A85\346EA69A-A92A-4C28-A2E0-DACAA0096A53.files\image011.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

默认情况下，使用[ImagePipeline](http://doc.scrapy.org/en/latest/topics/images.html#downloading-item-images)组件下载图片的时候，[图片名称是以图片URL的SHA1值进行保存的](http://doc.scrapy.org/en/latest/topics/images.html#images-storage)。

如：

图片URL:http://www.example.com/image.jpg

SHA1结果：3afec3b4765f8f0a07b78f98c07b83f013567a0a

则图片名称：3afec3b4765f8f0a07b78f98c07b83f013567a0a.jpg

但是，我想要以原来的图片名称进行保存，比如上面例子中的图片保存到本地的话，图片名称就应该是：image.jpg

[stackoverflow](http://stackoverflow.com/questions/6194041/scrapy-image-download-how-to-use-custom-filename?rq=1)上说是可以重写image\_key函数，不过我试了下，结果发现不行，重写的image\_key函数没被调用。

后面查看了下ImagePipeline的源码：

**[python]** [view plain copy](http://blog.csdn.net/php_fly/article/details/19688595)

[在CODE上查看代码片](https://code.csdn.net/snippets/199626)

* 1. class ImagesPipeline(FilesPipeline):
  2. """Abstract pipeline that implement the image thumbnail generation logic
  4. """
  6. MEDIA\_NAME = 'image'
  7. MIN\_WIDTH = 0
  8. MIN\_HEIGHT = 0
  9. THUMBS = {}
  10. DEFAULT\_IMAGES\_URLS\_FIELD = 'image\_urls'
  11. DEFAULT\_IMAGES\_RESULT\_FIELD = 'images'
  13. @classmethod
  14. def from\_settings(cls, settings):
  15. cls.MIN\_WIDTH = settings.getint('IMAGES\_MIN\_WIDTH', 0)
  16. cls.MIN\_HEIGHT = settings.getint('IMAGES\_MIN\_HEIGHT', 0)
  17. cls.EXPIRES = settings.getint('IMAGES\_EXPIRES', 90)
  18. cls.THUMBS = settings.get('IMAGES\_THUMBS', {})
  19. s3store = cls.STORE\_SCHEMES['s3']
  20. s3store.AWS\_ACCESS\_KEY\_ID = settings['AWS\_ACCESS\_KEY\_ID']
  21. s3store.AWS\_SECRET\_ACCESS\_KEY = settings['AWS\_SECRET\_ACCESS\_KEY']
  23. cls.IMAGES\_URLS\_FIELD = settings.get('IMAGES\_URLS\_FIELD', cls.DEFAULT\_IMAGES\_URLS\_FIELD)
  24. cls.IMAGES\_RESULT\_FIELD = settings.get('IMAGES\_RESULT\_FIELD', cls.DEFAULT\_IMAGES\_RESULT\_FIELD)
  25. store\_uri = settings['IMAGES\_STORE']
  26. return cls(store\_uri)
  28. def file\_downloaded(self, response, request, info):
  29. return self.image\_downloaded(response, request, info)
  31. def image\_downloaded(self, response, request, info):
  32. checksum = None
  33. for path, image, buf in self.get\_images(response, request, info):
  34. if checksum is None:
  35. buf.seek(0)
  36. checksum = md5sum(buf)
  37. width, height = image.size
  38. self.store.persist\_file(
  39. path, buf, info,
  40. meta={'width': width, 'height': height},
  41. headers={'Content-Type': 'image/jpeg'})
  42. return checksum
  44. def get\_images(self, response, request, info):
  45. path = self.file\_path(request, response=response, info=info)
  46. orig\_image = Image.open(StringIO(response.body))
  48. width, height = orig\_image.size
  49. if width < self.MIN\_WIDTH or height < self.MIN\_HEIGHT:
  50. raise ImageException("Image too small (%dx%d < %dx%d)" %
  51. (width, height, self.MIN\_WIDTH, self.MIN\_HEIGHT))
  53. image, buf = self.convert\_image(orig\_image)
  54. yield path, image, buf
  56. for thumb\_id, size in self.THUMBS.iteritems():
  57. thumb\_path = self.thumb\_path(request, thumb\_id, response=response, info=info)
  58. thumb\_image, thumb\_buf = self.convert\_image(image, size)
  59. yield thumb\_path, thumb\_image, thumb\_buf
  61. def convert\_image(self, image, size=None):
  62. if image.format == 'PNG' and image.mode == 'RGBA':
  63. background = Image.new('RGBA', image.size, (255, 255, 255))
  64. background.paste(image, image)
  65. image = background.convert('RGB')
  66. elif image.mode != 'RGB':
  67. image = image.convert('RGB')
  69. if size:
  70. image = image.copy()
  71. image.thumbnail(size, Image.ANTIALIAS)
  73. buf = StringIO()
  74. image.save(buf, 'JPEG')
  75. return image, buf
  77. def get\_media\_requests(self, item, info):
  78. return [Request(x) for x in item.get(self.IMAGES\_URLS\_FIELD, [])]
  80. def item\_completed(self, results, item, info):
  81. if self.IMAGES\_RESULT\_FIELD in item.fields:
  82. item[self.IMAGES\_RESULT\_FIELD] = [x for ok, x in results if ok]
  83. return item
  85. def file\_path(self, request, response=None, info=None):
  86. ## start of deprecation warning block (can be removed in the future)
  87. def \_warn():
  88. from scrapy.exceptions import ScrapyDeprecationWarning
  89. import warnings
  90. warnings.warn('ImagesPipeline.image\_key(url) and file\_key(url) methods are deprecated, '
  91. 'please use file\_path(request, response=None, info=None) instead',
  92. category=ScrapyDeprecationWarning, stacklevel=1)
  94. # check if called from image\_key or file\_key with url as first argument
  95. if not isinstance(request, Request):
  96. \_warn()
  97. url = request
  98. else:
  99. url = request.url
  101. # detect if file\_key() or image\_key() methods have been overridden
  102. if not hasattr(self.file\_key, '\_base'):
  103. \_warn()
  104. return self.file\_key(url)
  105. elif not hasattr(self.image\_key, '\_base'):
  106. \_warn()
  107. return self.image\_key(url)
  108. ## end of deprecation warning block
  110. image\_guid = hashlib.sha1(url).hexdigest()  # change to request.url after deprecation
  111. return 'full/%s.jpg' % (image\_guid)
  113. def thumb\_path(self, request, thumb\_id, response=None, info=None):
  114. ## start of deprecation warning block (can be removed in the future)
  115. def \_warn():
  116. from scrapy.exceptions import ScrapyDeprecationWarning
  117. import warnings
  118. warnings.warn('ImagesPipeline.thumb\_key(url) method is deprecated, please use '
  119. 'thumb\_path(request, thumb\_id, response=None, info=None) instead',
  120. category=ScrapyDeprecationWarning, stacklevel=1)
  122. # check if called from thumb\_key with url as first argument
  123. if not isinstance(request, Request):
  124. \_warn()
  125. url = request
  126. else:
  127. url = request.url
  129. # detect if thumb\_key() method has been overridden
  130. if not hasattr(self.thumb\_key, '\_base'):
  131. \_warn()
  132. return self.thumb\_key(url, thumb\_id)
  133. ## end of deprecation warning block
  135. thumb\_guid = hashlib.sha1(url).hexdigest()  # change to request.url after deprecation
  136. return 'thumbs/%s/%s.jpg' % (thumb\_id, thumb\_guid)
  138. # deprecated
  139. def file\_key(self, url):
  140. return self.image\_key(url)
  141. file\_key.\_base = True
  143. # deprecated
  144. def image\_key(self, url):
  145. return self.file\_path(url)
  146. image\_key.\_base = True
  148. # deprecated
  149. def thumb\_key(self, url, thumb\_id):
  150. return self.thumb\_path(url, thumb\_id)
  151. thumb\_key.\_base = True

其中，有这么一句话：

**ImagesPipeline.image\_key(url) and file\_key(url) methods are deprecated, please use file\_path(request, response=None, info=None) instead**

也就是说，在最新版本的Scrapy中（0.22.2），使用file\_path代替**image\_key**函数。

因此，我在自定义的ImagePipeline类中，重写了**file\_path**函数，代码如下：

**[python]** [view plain copy](http://blog.csdn.net/php_fly/article/details/19688595)

* 1. \_\_author\_\_ = 'Fly'
  2. #coding:utf-8
  3. from scrapy.contrib.pipeline.images import ImagesPipeline
  4. from scrapy.http import Request
  5. from scrapy.exceptions import DropItem
  7. class MyImagesPipeline(ImagesPipeline):
  8. def file\_path(self, request, response=None, info=None):
  9. image\_guid = request.url.split('/')[-1]
  10. return 'full/%s' % (image\_guid)
  12. def get\_media\_requests(self, item, info):
  13. for image\_url in item['image\_urls']:
  14. yield Request(image\_url)
  16. def item\_completed(self, results, item, info):
  17. image\_paths = [x['path'] for ok, x in results if ok]
  18. if not image\_paths:
  19. raise DropItem("Item contains no images")
  20. return item

以上代码主要返回原图片名称+图片后缀。

来自 <[*http://blog.csdn.net/php\_fly/article/details/19688595*](http://blog.csdn.net/php_fly/article/details/19688595)>

Tuesday, November 29, 2016

23:13

**Python装饰器**

简单来讲，可以不严谨地把Python的装饰器看做一个包装函数的函数。

比如，有一个函数：

def func():  
 print 'func() run.'

if '\_\_main\_\_' == \_\_name\_\_:  
 func()

运行后将输出:

func() run.

现在需要在函数运行前后打印一条日志, 但是又不希望或者没有权限修改函数内部的结构, 就可以用到装饰器(decorator):

def log(function):  
 def wrapper(\*args, \*\*kwargs):  
 print 'before function [%s()] run.' % function.\_\_name\_\_  
 rst = function(\*args, \*\*kwargs)  
 print 'after function [%s()] run.' % function.\_\_name\_\_  
 return rst  
 return wrapper

@log  
def func():  
 print 'func() run.'

if '\_\_main\_\_' == \_\_name\_\_:  
 func()

对于原来的函数func()并没有做修改,而是给其使用了装饰器log,运行后的输出为:

before function [func()] run.  
func() run.  
after function [func()] run.

把@log放到func()函数定义的地方,相当于执行了如下语句:

func = log(func)

因为log()返回了一个函数, 所以原本的func指向了log()返回的函数wrapper。wrapper的参数列表为(\*args, \*\*kwargs), 所以其可以接受所有的参数调用, 在wrapper中,先打印了一行'before function [%s()] run.' % function.\_\_name\_\_(在Python中函数也是对象,函数的\_\_name\_\_是它的名字),然后执行了原来的函数并记录了返回值,在输出 'after function [%s()] run.' % function.\_\_name\_\_后返回了函数的执行结果。

如果decorator本身需要传入参数，那就需要编写一个返回decorator的decorator。比如在Flask中:

@app.route('/')  
def index():  
 return 'hello, world!'

实现如下:

def log(text=''):  
 def decorator(function):  
 def wrapper(\*args, \*\*kwargs):  
 print 'before function [%s()] run, text: [%s].' % (function.\_\_name\_\_, text)  
 rst = function(\*args, \*\*kwargs)  
 print 'after function [%s()] run, text: [%s].' % (function.\_\_name\_\_, text)  
 return rst   
 return wrapper  
 return decorator

@log('log text')  
def func():  
 print 'func() run.'

if '\_\_main\_\_' == \_\_name\_\_:  
 func()

输出如下:

before function [func()] run, text: [log text].  
func() run.  
after function [func()] run, text: [log text].

最后脑洞小开一下, 有没有办法实现既支持不带参数(如log), 又支持带参数(如log('text'))的decorator吗?

def log(argument):  
 if not callable(argument):  
 def decorator(function):  
 def wrapper(\*args, \*\*kwargs):  
 print 'before function [%s()] run, text: [%s].' % (function.\_\_name\_\_, text)  
 rst = function(\*args, \*\*kwargs)  
 print 'after function [%s()] run, text: [%s].' % (function.\_\_name\_\_, text)  
 return rst   
 return wrapper  
 return decorator  
 def wrapper(\*args, \*\*kwargs):  
 print 'before function [%s()] run.' % function.\_\_name\_\_  
 rst = function(\*args, \*\*kwargs)  
 print 'after function [%s()] run.' % function.\_\_name\_\_  
 return rst  
 return wrapper

如上~~~

来自 <[*https://zhuanlan.zhihu.com/p/20259386?refer=xlz-d*](https://zhuanlan.zhihu.com/p/20259386?refer=xlz-d)>

Tuesday, November 29, 2016

23:13

**Python面向对象高级特性 -- @property**

在 Python的面向对象编程中，一般会将对象的字段暴露在外面，直接通过foo.bar的方式赋值或者取值。相比之下，Java的常规做法是封装一个 private field，然后通过get/set方法取值赋值。Java的好处在于，可以严格控制变量值的正确性，比如：

public class Person {  
 private int age;  
 public int getAge() {return age;}  
 public void setAge(int age){  
 if (0 <= age && age <= 100)  
 this.age = age;  
 else  
 throw new IllegalArgumentException("age error.");  
 }  
}

Java通过函数的封装，可以控制age的值在[0， 100]的范围内。一般来讲，Python的写法是这样的：

class Person(object):  
 def \_\_init\_\_(self, age):  
 self.age = age

然后，就可以通过obj.age对age进行操作了，但是如上面所说，外界的直接调用可能会导致字段的值不可预料。我们可以像Java一样做一点修改：

class Person(object):  
 def get\_age(self):  
 return self.\_age

def set\_age(self, age):  
 if 0 <= age <= 100:  
 self.\_age = age  
 else:  
 raise ValueError("age error.")

我们可以试着调用一下：

In [1]: p = Person()

In [2]: p.set\_age(12) # 赋值成功

In [3]: p.get\_age() # 取值成功  
Out[3]: 12

In [4]: p.set\_age(123) # 赋值失败(数字超出预期)  
---------------------------------------------------------------------------  
ValueError Traceback (most recent call last)  
<ipython-input-7-a868d6f8da36> in <module>()  
----> 1 p.set\_age(123)

<ipython-input-3-8d90d1c4b785> in set\_age(self, age)  
 6 self.\_age = age  
 7 else:  
----> 8 raise ValueError("age error.")  
 9

ValueError: age error.

从上面的调用可以看出，一定程度上，我们完成了需求，但是我希望可以直接通过对象.属性来操作，同时又不想失去检查参数的严谨性，这就用到了@property。

【[这篇博客](https://link.zhihu.com/?target=http%3A//xlzd.me/2015/08/01/python-decorator)】介绍了Python的装饰器（decorator），@property就是一个装饰器，Python内置的@property装饰器就是负责把一个方法变成属性调用的：

class Person(object):  
 @property  
 def age(self):  
 return self.\_age

@age.setter  
 def age(self, age):  
 if 0 <= age <= 100:   
 self.\_age = age  
 else:  
 raise ValueError("age error.")

在一个getter方法前加上@property就可以将其变为属性了，这时候，@property自己本身又创建了另一个装饰器@age.setter，作用是将setter方法变成直接对属性的赋值。现在，就可以这样操作了：

In [1]: p = Person()

In [2]: p.age = 13 # 赋值成功

In [3]: p.age # 取值  
Out[3]: 13

In [4]: p.age = 123 # 赋一个错误的值会抛出异常  
---------------------------------------------------------------------------  
ValueError Traceback (most recent call last)  
<ipython-input-27-10cb4da6198b> in <module>()  
----> 1 p.age = 123

<ipython-input-23-7b751228fdb9> in age(self, age)  
 8 self.\_age = age  
 9 else:  
---> 10 raise ValueError("age error.")  
 11

ValueError: age error.

至此，我们可以直接通过对象.属性来操作，同时又保证了数据的正确性，这就是@property的神奇功效。

另外我们也可以定义只读属性，方法是只定义getter方法而不定义setter：

注意到这个神奇的@property，我们在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过getter和setter方法来实现的。

还可以定义只读属性，只定义getter方法，不定义setter方法就是一个只读属性：

class Person(object):  
 def \_\_init\_\_(self, birthday):  
 self.\_birthday = birthday  
 @property  
 def birthday(self):  
 return self.\_birthday

上面的birthday就是一个只读属性，只有在创建对象的时候赋值，后面便不能做修改了：

In [1]: p = Person(birthday='2000-01-01') # 创建对象并给birthday赋值

In [2]: p.birthday # 取值  
Out[2]: '2000-01-01'

In [3]: p.birthday = '1999-12-12' # 重新给birthday赋值，失败  
---------------------------------------------------------------------------  
AttributeError Traceback (most recent call last)  
<ipython-input-31-3a5b26e3623a> in <module>()  
----> 1 p.birthday = '1999-12-12'

AttributeError: can't set attribute

**小结**

@property可以让我们的代码在保持易用性的同时保证了对数据的必要的检查。

来自 <[*https://zhuanlan.zhihu.com/p/20262952?refer=xlz-d*](https://zhuanlan.zhihu.com/p/20262952?refer=xlz-d)>

Tuesday, November 29, 2016

23:29

[**Python: generator与yield**](http://xlzd.me/2015/08/24/python-generator)

作者：[xlzd](http://xlzd.me) | 时间：2015-08-24 03:21:00 | 分类：[技术](http://xlzd.me/category/tech/) | 评论：[0 评论](http://xlzd.me/2015/08/24/python-generator)

第一步，我们想要生成fibonacci数列前N项，我们可以这样做：

def fib(size):  
 a, b = 0, 1  
 while size:  
 print a,   
 a, b = b, a + b  
 size -= 1

执行可以得到输出如下：

In [1]: fib(10)  
0 1 1 2 3 5 8 13 21 34

这个函数的问题在于，我们只能调用它输出结果，并没办法拿到返回值，于是通用性不够。现在对这个函数做一点修改如下：

def fib(size):  
 a, b = 0, 1  
 lst = []  
 while size:  
 lst.append(a)  
 a, b = b, a + b  
 size -= 1  
 return lst

现在再次调用：

In [15]: fib(10)  
Out[15]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

现在成功拿到fibonacci数列的前N项了，但是还有一个问题就是，当传入参数size过大的时候，待返回的list可能会因为内存不够而撑爆。如果要控制内存为O(1)，一个做法便是使用可迭代（iterable）的对象。Python中的list|set|str|tuple|dict|file都是iterable的， 要自己实现一个iterable对象，方法之一便是定义class的\_\_iter\_\_()方法，在\_\_iter\_\_()方法中返回一个迭代器(iterator)即可。迭代器(iterator)可以看做一个数据流，重复调用它的next()方法可以依次取出这个数据流中的每一个元素，当没有更多元素的时候则抛出StopIteration。

下面是Python文档中对iterable和iterator的描述：

**iterable**

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as list, str, and tuple) and some non-sequence types like dict and file and objects of any classes you define with an **iter**() or **getitem**() method. Iterables can be used in a for loop and in many other places where a sequence is needed (zip(), map(), ...). When an iterable object is passed as an argument to the built-in function iter(), it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call iter() or deal with iterator objects yourself. The for statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop.

**iterator**

An object representing a stream of data. Repeated calls to the iterator’s next() method return successive items in the stream. When no more data are available a StopIteration exception is raised instead. At this point, the iterator object is exhausted and any further calls to its next() method just raise StopIteration again. Iterators are required to have an **iter**() method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a list) produces a fresh new iterator each time you pass it to the iter() function or use it in a for loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

所以上面的代码又可以改为如下：

class Fib(object):   
 def \_\_init\_\_(self, size):   
 self.size = size  
 self.a, self.b = 0, 1   
 def \_\_iter\_\_(self):   
 return self   
 def next(self):   
 if self.size:   
 now = self.a  
 self.a, self.b = self.b, self.a + self.b   
 self.size -= 1  
 return now  
 raise StopIteration()

现在的调用结果如下：

In [1]: for x in Fib(10):  
 ....: print x,  
 ....:   
0 1 1 2 3 5 8 13 21 34

至此，我们可以拿到fibonacci的前N项，又不用担心过大的内存开销。但是我们也会发现，上面的代码相比之前的函数冗杂了很多。我们希望兼顾简洁的风格和iterable的优势，就用到了Python的yield：

def fib(size):  
 a, b = 0, 1  
 while size:  
 yield a  
 a, b = b, a + b  
 size -= 1

调用如下：

In [1]: for x in fib(10):  
 ....: print x,  
 ....:   
0 1 1 2 3 5 8 13 21 34

通俗的理解就是，yield把一个普通的函数变成了一个生成器（generator），对fib函数的调用将会返回一个iterable对象，在对其在for循环中迭代时，每次执行到yield的时候，就返回一个值。到下一次循环，代码从yield的下一句开始继续执行。

现在我们手动执行一次，就明白了：

In [1]: f = fib(10)

In [2]: type(f)  
Out[2]: generator  
   
In [3]: f.next()  
Out[3]: 0

In [4]: f.next()  
Out[4]: 1

In [5]: f.next()  
Out[5]: 1

In [6]: f.next()  
Out[6]: 2

In [7]: f.next()  
Out[7]: 3

In [8]: f.next()  
Out[8]: 5

In [9]: f.next()  
Out[9]: 8

In [10]: f.next()  
Out[10]: 13

In [11]: f.next()  
Out[11]: 21

In [12]: f.next()  
Out[12]: 34

In [13]: f.next()  
---------------------------------------------------------------------------  
StopIteration Traceback (most recent call last)  
<ipython-input-26-c3e65e5362fb> in <module>()  
----> 1 f.next()

StopIteration:

当函数执行结束以后，就会抛出一个StopIteration，for循环会根据StopIteration判断循环结束。

值得注意的是，fib是一个generator function(生成器函数)，而fib(10)则是一个generator(生成器)。

另一个概念叫做generator expression(生成器表达式)：

我们知道在Python中可以通过列表推导式创建一个List：

In [1]: [x for x in xrange(10)]  
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

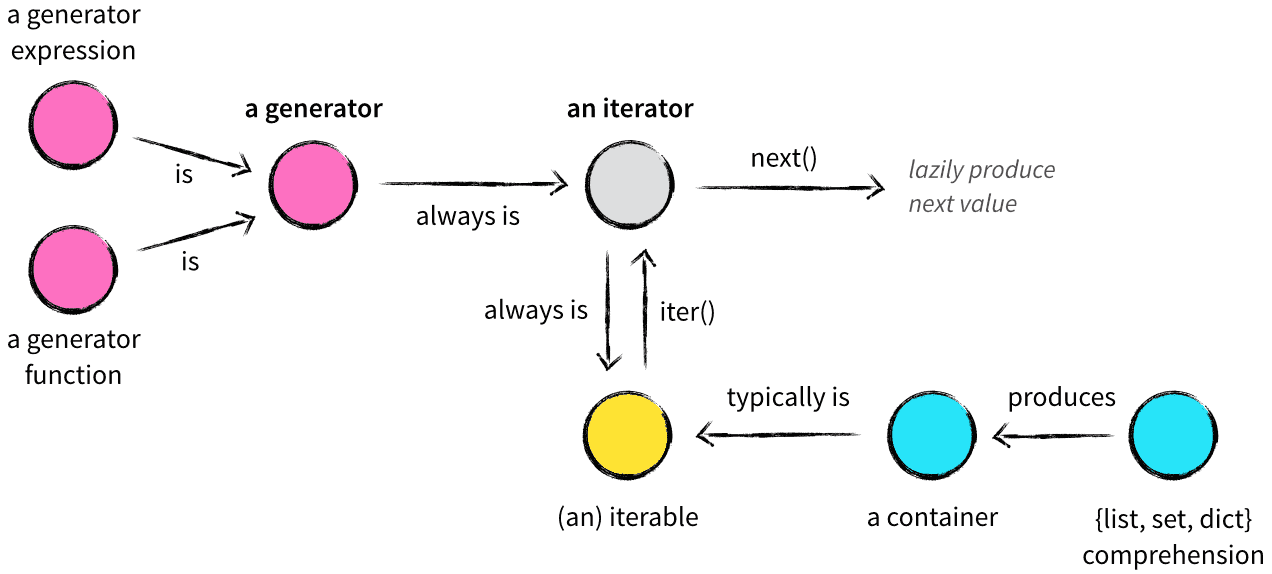
这样的做法同样可能因为列表过大导致内存占用过多甚至溢出的问题，生成器表达式的作用在于生成一个generator，然后在我们需要的时候（迭代到的时候）再计算生成需要的值：

In [1]: (x for x in xrange(100))  
Out[1]: <generator object <genexpr> at 0x7fe221ad29b0>

生成器表达式和列表推导一样简单灵活又强大，这里不过多赘述。

**总结**

关于generator expression、generator function、generator、iterator、iterable之间的关系，下图阐释的非常清除了（图片摘自一个国外博客）。



来自 <[*http://xlzd.me/2015/08/24/python-generator*](http://xlzd.me/2015/08/24/python-generator)>

Tuesday, November 29, 2016

23:29

[**最大字数组问题——分治策略**](http://xlzd.me/2015/02/13/divide-and-conquer-1)

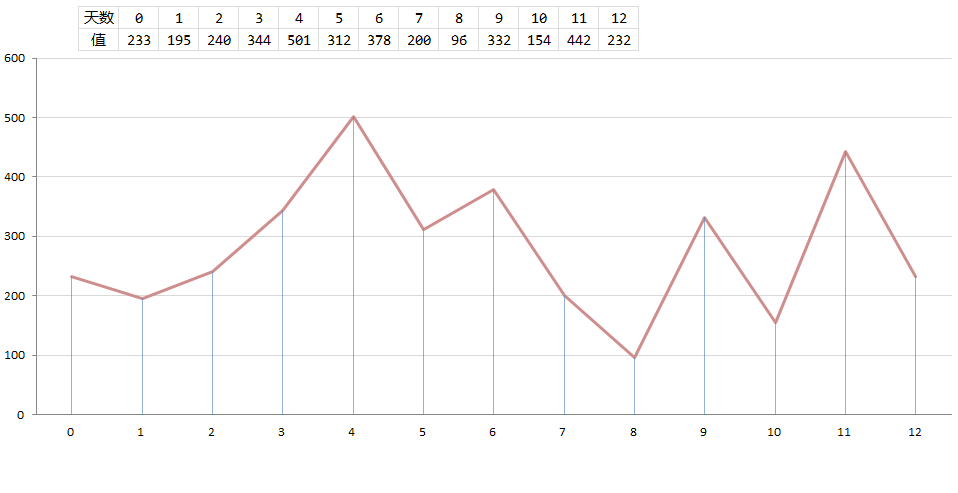
作者：[xlzd](http://xlzd.me) | 时间：2015-02-13 13:01:00 | 分类：[技术](http://xlzd.me/category/tech/) | 评论：[0 评论](http://xlzd.me/2015/02/13/divide-and-conquer-1)

这篇博客是学习《算法导论》的笔记（P.S.代码使用Python）。

用分治法解决问题，大致的步骤如下：

* **分解*(Divide)*：**将问题划分为一些子问题，子问题形式与原问题一样，但是规模变小了。
* **解决*(Conquer)*：**递归地将求解子问题，当子问题规模足够小的时候停止递归，直接求解。
* **合并*(Combine)*：**将子问题的解组合成原问题的解。

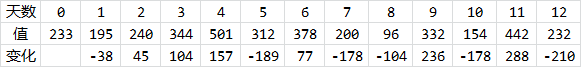
如果你通过某种手段获得了未来若干天里的某家公司的股票价格走势，你可以在某一时刻买入股票并在另一个时刻抛出。当然，你希望可以从中获得最大的利益，比如，对于如下的走势：



并不是所有的情况下都可以在最低价买入、最高价抛出的，比如如上走势图中，最高价发生在第4天但是最低价发生在第8天。

**暴力求解：**如果简单地尝试每个可能的买入和抛出的组合，只需要买入发生在抛出之前即可满足条件，但是这种方法的效率实在太低，尤其是在天数足够长的情况下。

**使用分治法求解：**我们的目的是寻找一段时间，使得获得的利益最大，如果我们从每日价格的变化入手，第i天的价格变化表示为与前一天的价格差，如果将价格的变化看成一个数组arr，如此，问题就变成了寻找arr的和最大的非空连续子数组，也即是**最大字数组**。



假定我们寻找数组arr[low..high]的最大子数组，使用分治法的意思是我们需要将它划分为尽量规模一样的两个子数组。就是说，找到数组的中间位置mid，然后分别求解arr[low..mid]和arr[mid+1..high]，arr[low, high]的最大子数组arr[i..j]必然属于如下三种情况之一：

* 完全位于子数组arr[low..mid]中，因此low<=i<=j<=mid。
* 完全位于子数组arr[mid+1..high]中，因此mid<i<=j<=high。
* 跨越了中点mid，此时low<=i<=mid<j<=high。  
  于是我们使用函数find\_max\_crossing\_subarray()来求解跨越中点的情况下的解，它返回跨越中点的最大子数组的左右边界以及最大子数组的值，然后分别与arr[low..mid]和arr[mid+1..high]的解比较，得出最后的答案。

def find\_max\_crossing\_subarray(arr, low, mid, high):  
 left\_sum = -9999999999  
 sum = 0  
 # 求出左边的最大子数组arr[max\_left\_pos, mid]  
 for left\_pos in xrange(mid, low-1, -1):  
 sum += arr[left\_pos]  
 if sum > left\_sum:  
 left\_sum = sum  
 max\_left\_pos = left\_pos

right\_sum = -9999999999  
 sum = 0  
 # 求出右边的最大子数组arr[mid+1, max\_right\_pos]  
 for right\_pos in xrange(mid+1, high+1):  
 sum += arr[right\_pos]  
 if sum > right\_sum:  
 right\_sum = sum  
 max\_right\_pos = right\_pos

return max\_left\_pos, max\_right\_pos, left\_sum + right\_sum

有了这个求解跨越中点的最大子数组的函数，我们就可以设计求解最大子数组的问题的分治算法了，find\_max\_subarray()接收数组arr以及其左右下标为输入，返回arr[low..high]的最大子数字的左右下标以及值：

def find\_max\_subarray(arr, low, high):  
 if high == low:  
 return low, high, arr[low]

mid = int((low + high) >> 1)  
 left\_low, left\_high, left\_sum = find\_max\_subarray(arr, low, mid)  
 right\_low, right\_high, right\_sum = find\_max\_subarray(arr, mid+1, high)  
 cross\_low, cross\_high, cross\_sum = find\_max\_crossing\_subarray(arr, low, mid, high)  
   
 if left\_sum >= right\_sum and left\_sum >= cross\_sum:  
 return left\_low, left\_high, left\_sum  
 elif right\_sum >= left\_sum and right\_sum >= cross\_sum:  
 return right\_low, right\_high, right\_sum  
 else:  
 return cross\_low, cross\_high, cross\_sum

在函数find\_max\_subarray()中，第2行中如果low与high相等，则表示只有一个元素待求解，于是直接返回那一个元素和下标。反之，6~8行分别表示求解arr[low..mid]和arr[mid+1..high]的最大子数组和跨越中点的最大子数组。10~15行比较孰是真正的最大子数组。

通过find\_max\_subarray()求得，在第8天（第7天结束后）买入，第11天抛出可以得到最大利益。

P.S.最大子数组问题我学会的最优解：

def find\_max\_subarray(arr):  
 now\_sum = max\_sum = 0  
 for item in arr:  
 now\_sum += item  
 max\_sum = (max\_sum, now\_sum)[max\_sum < now\_sum]  
 now\_sum = (now\_sum, 0)[now\_sum < 0]  
 return max\_sum

这个解法叫『在线处理』，意思是随时停止它，都可以得到目前为止的正确解。 它只需O(n)的时间复杂度，就可以完成求解~~~

来自 <[*http://xlzd.me/2015/02/13/divide-and-conquer-1*](http://xlzd.me/2015/02/13/divide-and-conquer-1)>

Tuesday, November 29, 2016

23:32

[**Python：Pythonic Code Styles**](http://xlzd.me/2015/11/29/pythonic)

作者：[xlzd](http://xlzd.me) | 时间：2015-11-29 11:41:00 | 分类：[技术](http://xlzd.me/category/tech/) | 评论：[3 评论](http://xlzd.me/2015/11/29/pythonic)

[上一篇博客](http://xlzd.me/2015/11/28/hello-python)分享一些Python的基础内容，今天分享一点Pythonic的Python代码写法，帮助初学者迅速提升代码逼格。

**交换两个数**

常规写法：

t = a  
a = b  
b = t

二逼写法：

a = a ^ b  
b = a ^ b  
a = b ^ a

xlzd推荐写法：

a, b = b, a

**迭代的时候带上序号**

常规写法：

index = 0  
for item in iterable:  
 print index, item  
 index += 1

xlzd推荐写法：

for index, item in enumerate(iterable):  
 print index, item

其中，enumerate还可以接收第二个参数，指定开始的位置。

**同时迭代两个可迭代对象**

常规写法：

for pos in xrange(len(iterable1)):  
 item1 = iterable1[pos]  
 item2 = iterable2[pos]  
 print item1, item2

xlzd推荐写法：

for item1, item2 in zip(iterable1, iterable2):  
 print item1, item2

**获取对象的方法列表**

使用dir(obj)即可得到obj对象的所有属性（字段）、方法。比如：

In [26]: l = ['a', 'b', 'c']

In [27]: dir(l)  
Out[27]:  
['\_\_add\_\_', '\_\_class\_\_', '\_\_contains\_\_', '\_\_delattr\_\_', '\_\_delitem\_\_',  
 '\_\_delslice\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_',  
 '\_\_getattribute\_\_', '\_\_getitem\_\_', '\_\_getslice\_\_', '\_\_gt\_\_',  
 '\_\_hash\_\_', '\_\_iadd\_\_', '\_\_imul\_\_', '\_\_init\_\_', '\_\_iter\_\_',  
 '\_\_le\_\_', '\_\_len\_\_', '\_\_lt\_\_', '\_\_mul\_\_', '\_\_ne\_\_', '\_\_new\_\_',  
 '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_reversed\_\_',  
 '\_\_rmul\_\_', '\_\_setattr\_\_', '\_\_setitem\_\_', '\_\_setslice\_\_',  
 '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', 'append',  
 'count', 'extend', 'index', 'insert', 'pop', 'remove',  
 'reverse', 'sort']

**三元表达式**

Python里没有C-Like语言中condition ? value1 : value2的语法，但是我们可以有很多中方法实现同样的功能：

value1 if condition else value2  
(value2, value1)[condition]  
condition and value1 or value2

以上语法都可以实现同样的功能。

**if**

# 如果要判断一个变量是否等于某几个数字, this is bad  
if num == 2 or num ==4 or num ==6:  
 pass  
# 如果要判断一个变量是否等于某几个数字, this is good  
if num in (2, 4, 6):  
 pass

# 如果判断一个数字是否在一个区间里，this is bad  
if num >= 10 and num <=100:  
 pass  
# 如果判断一个数字是否在一个区间里，this is good  
if 10 <= num <= 100:  
 pass

**字符串格式化**

普通程序员：

name = 'xlzd'  
age = 21  
string = 'My name is %s and I am %d years old' % (name, age)

二逼程序员：

name = 'xlzd'  
age = 21  
string = 'My name is ' + name + ' and I am ' + str(age) + ' years old.'

xlzd推荐写法：

string = 'My name is {name} and I am {age} years old.'.format(name='xlzd', age=21)

**列表推倒**

要实现从一个list中取出满足某个条件(在100-1000开区间中)的元素组成一个新的list

普通写法：

l = [1, 2, 4, 34, 76, 123, 560, 590, 777, 1200]  
new\_list = []  
for item in l:  
 if 100 < item < 1000:  
 new\_list.append(item)

xlzd推荐写法：

l = [1, 2, 4, 34, 76, 123, 560, 590, 777, 1200]  
new\_list = [item for item in l if 100 < item < 1000] # 生成一个list  
＃ or  
new\_list = (item for item in l if 100 < item < 1000) # 生成一个generator

关于上面推荐的两种写法，后一种相对前一种的好处是生成了一个generator（以前的博客提到过的），它会延迟计算而不是立刻得出结果，所以在内存使用上会更优。

这样的推导语句同样适用于dict和set。

**序列切片**

list、str等对象的切片操作，可以优雅而高效的实现很多功能：

In [1]: l = range(10) # 初始化一个list

In [2]: l  
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: l[::-1] # 逆序  
Out[3]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

In [4]: l[5:] # 切片，取第5个以后  
Out[4]: [5, 6, 7, 8, 9]

In [5]: l[:5] # 切片，取第五个及以前  
Out[5]: [0, 1, 2, 3, 4]

In [6]: l[::2] # 从第0位开始，每隔一位取一个数   
Out[6]: [0, 2, 4, 6, 8]

In [7]: l[1::2] # 从第1位开始，每隔一位取一个数   
Out[7]: [1, 3, 5, 7, 9]

**switch-case**

Python没有switch-case语法，一般程序员会使用if-elif-else来模拟这样的操作，但是有经验的逼格高的程序员一般通过映射一个dict。

# Java code  
String int2en(int num) {  
 switch(num) {  
 case 1: return "one";  
 case 2: return "two";   
 case 3: return "there";   
 case 4: return "four";  
 case 5: return "five";  
 default: return "out";  
 }  
}

普通程序员的Python实现：

def int2en(num):  
 if num == 1:  
 return "one"  
 if num == 2:  
 return "two"  
 if num == 3:  
 return "there"  
 if num == 4:  
 return "four"  
 if num == 5:  
 return "five"  
 return "out"

xlzd推荐：

INT\_EN\_MAP = {  
 1: 'one',  
 2: 'two',  
 3: 'there',  
 4: 'four',  
 5: 'five'  
}

def int2en(num):  
 return INT\_EN\_MAP.get(num, 'out')

**拼接文件路径**

一般比较常见的写法是连接字符串，但是这样的写法会导致另外的平台可能不可用，也容易出现拼接字符串的错误。比如OS X和Linux上使用"/"而Windows却使用"\"来区分文件路径。通过os.path.join函数来完成这样的操作：

import os  
base = '/User/xlzd'

# this is bad  
target = base + '/workspace'

# this is good  
target = os.path.join(base, 'workspace')

**小结**

Python中一个功能可以有很多中不同的实现方式，在写代码之前最好先构思好，尽量写出美观、高效、规范的好代码。除了以上，还有很多Pythonic的代码写法，时间关系，暂不一一赘述。

来自 <[*http://xlzd.me/2015/11/29/pythonic*](http://xlzd.me/2015/11/29/pythonic)>

Tuesday, November 29, 2016

23:47

[正则表达式01：re模块](http://xlzd.me/2015/01/16/python-re-01)

作者：[xlzd](http://xlzd.me) | 时间：2015-01-16 20:01:00 | 分类：[技术](http://xlzd.me/category/tech/) | 评论：[0 评论](http://xlzd.me/2015/01/16/python-re-01)

####re模块：核心函数和方法

Python默认的正则表达式模块是re，re模块在Python1.5中被引入。下表列出了re模块最常用的函数，其中很多函数也与已编译的正则表达式对象(regex objects)和正则“匹配对象”(match objects)的方法同名且具有相同的功能。

|  |  |
| --- | --- |
|  | \*\*常用的函数和方法\*\* |
| \*\*函数\*\* | \*\*描述\*\* |
| compile(\*pattern, flags=0\*) | 对正则表达式模式pattern进行编译，flags是可选标识符，并返回一个regex对象 |
| match(\*pattern, string, flags=0\*) | 尝试用正则表达式模式pattern匹配字符串string，flags是可选标识符，如果匹配成功，则返回一个匹配对象，否则返回`None` |
| search(\*pattern, string, flags=0\*) | 在字符串string中搜索正则表达式模式pattern的第一次出现，flags是可选标识符，如果匹配成功，则返回一个匹配对象，否则返回`None` |
| findall(\*pattern, string[, flags]\*) | 在字符串string中搜索正则表达式模式pattern的所有（非重复）出现，返回一个匹配对象的列 |
| finditer(\*pattern, string[, flags]\*) | 与findall()相同，但返回一个迭代器而不是列表 |
| split(\*pattern, string, max=0\*) | 根据正则表达式pattern中的分隔符把字符串string分割为一个列表，返回成功匹配的列表，最多分割max次（默认分割所有） |
| sub(\*pattern, repl, string, max=0\*) | 把字符串string中所有匹配正则表达式pattern的地方替换成字符串repl，如果max的值没有给出，则对所有匹配的地方进行替换 |
| group(\*num=0\*) | 返回全部匹配对象（或者是num编号的子组） |
| groups() | 返回一个包含全部匹配的子组的元组（如果没有成功匹配，则返回空元组） |

####使用compile()编译正则表达式

在模式匹配之前，正则表达式模式必须先被编译成regex对象。由于正则表达式在执行过程中被多次用于比较，对它进行**预编译**可以在性能上得到一定提升。而且，既然正则表达式的编译是必须的，那么使用预编译来提升执行性能无疑是明智之举。re.compile()就是用来提供此功能的。

其实模块函数会对已编译对象进行缓存，所以不是所有使用系统正则表达式模式的search()和match()都需要编译。即使这样，你仍然节省了查询缓存和用相同的字符串反复调用函数的性能开销。缓存区可以容纳100个已编译的regex对象。

编译regex对象时给出一些可选标识符，可以得到特殊的编译对象。这些对象将允许不区别大小写的匹配，或使用系统的本地设置定义的字母表进行匹配等等。

####匹配对象和group()、groups()方法

在处理正则表达式时，除了regex对象以外，还有另外一种对象类型——**匹配对象**，这些对象是在match()或search()被成功调用以后所返回的结果。匹配对象的两个主要方法是：group()、groups()。

group()方法返回所有匹配对象或者根据要求返回某个特定子组。groups()则是返回包含所有子组的元组。

来自 <[*http://xlzd.me/2015/01/16/python-re-01*](http://xlzd.me/2015/01/16/python-re-01)>

[**Python与正则表达式02：re模块的使用<1>**](http://xlzd.me/2015/01/19/python-re-02)

作者：[xlzd](http://xlzd.me) | 时间：2015-01-19 14:20:00 | 分类：[技术](http://xlzd.me/category/tech/) | 评论：[0 评论](http://xlzd.me/2015/01/19/python-re-02)

前一篇博客记录了re模块和简单的使用，这一篇主要记录re模块的一些函数的使用，以备自己不时之需。

####用match()匹配字符串

首先从match()入手，match()是re模块的函数，也是正则表达式对象(regex object)的方法。它的作用是尝试从**字符串的开头**开始对模式进行匹配。如果匹配成功则返回一个匹配对象，否则返回None。匹配对象的group()方法可以用来显示那个成功的匹配。以下是一个简单示例：

>>> m = re.match('foo', 'foo') #匹配  
>>> m  
<\_sre.SRE\_Match object at 0x01C7D4B8>  
>>> if m is not None: #如果成功则显示  
... m.group()  
...   
'foo'

模式'foo'完全匹配字符串'foo'。

下面是一个匹配失败的例子：

>>> m = re.match('foo', 'bar') #匹配  
>>> if m is not None: #如果成功则显示  
... m.group()  
...

因为'foo'模式不能匹配字符串'bar'，所以返回值为None，于是没有任何输出。这里的if用来检测匹配是否成功，在实际运用中我一般也会写上以确保安全性，如果没有检测的话，可能会发生AttributeError异常（None没有group()方法）。

####search()在一个字符串中查找一个模式

其实，要搜索的模式出现在字符串中间的概率比出现在开头要高得多，search()就是处理这样的情况。search()和match()的工作方式几乎一样，不同之处在于search()会检查**字符串任意位置**与给定正则表达式模式的匹配情况。如果匹配成功也会返回一个匹配对象，否则返回None。以下的例子反映了search()和match()的区别：

>>> m = re.match('foo', 'this is food.') #匹配失败  
>>> if m is not None:  
... m.group  
...  
>>> #以上没有任何输出  
>>> m = re.search('foo', 'this is food.') #匹配成功  
>>> if m is not None:  
... m.group  
...  
'foo'

结果很明确，match()尝试从**字符串开头**进行匹配，也就是用'f'匹配't'，很显然是匹配失败的。虽然'*foo*'确实出现在'*this is food.*'中，但是match()会从开头匹配。search()搜索字符串中模式首次出现的位置（*从左到右匹配*），而不是尝试从开头进行匹配。

####匹配多个字符串( | )和匹配任意单个字符( . )

前面的博客已经提到了管道符号|，以及匹配任意一个字符的.，下面是一段示例代码：

>>> rex = 'bat|bit|bot' #匹配：bat, bit, bot  
>>> m = re.match(rex, 'bat') #可以匹配 bat  
>>> if n is not None:  
... m.group()  
...  
'bat'  
>>> m = re.match(rex, 'bet') #不匹配  
>>> if m is not None: m.group()  
...  
>>> m = re.match(rex, 'Baidu/Alibaba/Tencent -> bat') #不匹配  
>>> if m is not None: m.group()  
...  
>>> m = re.search(rex, 'Baidu/Alibaba/Tencent -> bat') #搜索到'bat'  
>>> if m is not None: m.group()  
...  
'bat'  
>>>   
>>>   
>>> #以下是匹配任意单个字符  
>>> rex = '.EE'  
>>> m = re.match(rex, 'SEE') #匹配字符'S'  
>>> if m is not None: m.group()  
...  
'SEE'  
>>> m = re.match(rex, 'EE') #没有字符匹配  
>>> if m is not None: m.group()  
...  
>>> m = re.match(rex, '\nEE') #不可以匹配回车符  
>>> if m is not None: m.group()  
...   
>>> m = re.search(rex, 'JavaEE') #匹配'aEE'  
>>> if m is not None: m.group()  
...  
'aEE'

下面的例子是搜索一个句点本身（小数点）的正则表达式，反斜线用来对特殊字符转义，使其失去特殊含义。

>>> rex\_dt = '3.14' #正则表达式任意单个字符   
>>> rex\_pi = '3\.14' #小数点  
>>> m = re.match(rex\_pi, '3.14') #匹配  
>>> if m is not None: m.group()  
...   
'3.14'  
>>> m = re.march(rex\_dt, '3A14') #句点匹配'A'  
>>> if m is not None: m.group()  
...   
'3A14'  
>>> m = re.march(rex\_dt, '3.14') #句点匹配'.'  
>>> if m is not None: m.group()  
...   
'3.14'

####创建字符集合( [] )

[]的意思前面已经提到过，表示匹配任一集合中表示的字符。以下是一点简单的示例：

>>> rex = '[a1][b2][c3]'  
>>> m = re.match(rex, 'a23') #匹配'a23'  
>>> if m is not None: m.group()  
...   
'a23'  
>>> m = re.match(rex, '12c') #匹配'12c'  
>>> if m is not None: m.group()  
...   
'12c'  
>>>   
>>> #与管道符号 | 不同  
>>> m = re.match('abc|123', 'a23') #不匹配  
>>> if m is not None: m.group()  
...   
>>> m = re.match('abc|123', '123') #匹配  
>>> if m is not None: m.group()  
...   
'123'

####从字符串开头、结尾匹配或在单词边界匹配

以下示例代码演示了从字符串开头、结尾匹配或在单词边界匹配，也即是介绍了*锚点性正则表达式操作符*。

>>> m = re.search('^Game', 'Game over') #匹配'Game'开头的字符串  
>>> if m is not None: m.group()  
...   
'Game'  
>>> m = re.search('^Game', 'The Game is over') #'Game'不在开头  
>>> if m is not None: m.group()  
...   
>>> m = re.search(r'\band', 'Tom and Jerry.') #单词边界  
>>> if m is not None: m.group()  
...   
'and'  
>>> m = re.search(r'\band', 'TomandJerry.') #不在单词边界  
>>> if m is not None: m.group()  
...   
>>> m = re.search(r'\Band', 'TomandJerry.') #匹配不在单词边界  
>>> if m is not None: m.group()  
...   
'and'

这个示例中出现了*原始字符串(raw strings)*，通常，在正则表达式中使用原始字符串是不错的选择，因为可以更加清晰的使阅读程序的人读懂你的正则表达式。

####用findall()找到每个出现的匹配部分

findall()用来非重叠地搜索某一个字符串中一个正则表达式模式出现的情况。findall()与search()都会搜索字符串，不同的是findall()不论搜索成功与否都会返回一个list：

* 如果没有找到匹配则返回空列表
* 如果找到匹配部分，则返回所有匹配部分的集合

>>> re.findall('foo', 'foobar')  
['foo']  
>>> re.findall('foo', 'barrrrrfo')  
[]  
>>> re.findall('foo', 'foo and bar and foo')  
['foo', 'foo']

####-TO BE CONTINUED

来自 <[*http://xlzd.me/2015/01/19/python-re-02*](http://xlzd.me/2015/01/19/python-re-02)>

2017年3月29日

19:19

**Python读书笔记**

**官方Doc运算符**

除法永远返回 float print(10/5) # 结果为 2.0

整除（地板除 floor division）print(7//3) # 2

print(7//-3) # -3

% 取模运算

如果整数int 与 浮点数float 混合运算，整数就会自动转化为float

字符串 自动连接：

text = (‘who’

‘ are you?’)

Excel读取模块 ：xlwt

**第四章 Python的对象类型**

编程，是用一些步骤（操作方式），去处理一些事情（操作的对象）。

程序处理的数据，以对象的形式出现。对象，是内存中的一部分，包含数值和相关的各种操作。

Python 程序由模块构成，模块包含语句，语句包含表达式，表达式建立并且处理对象——所以，对象是Python中最基本的概念。在Python程序中处理的每种东西，都是一种对象。

C／C++／Java，编程中很多工作都是作数据结构：定义数据类型，内存分配，对象搜索和存取等。而Python的内置数据类型非常强大。

**为什么要优先使用内置数据类型？（而不是自己构建数据类型）**

1.写起来容易，效率高

2.即使要构建自己的数据类型（比如写个class），在内置类型的基础上构建也会更简单

Python 没有类型声明，表达式的语法就决定了创建和使用的对象的类型。每个数据类型，都和某些操作一一对应。

核心数据类型（对象的类型）：

数字（包括集合）／字符串／列表／字典／元组／None／Boolean／函数／类／模块／文件

**如何查看帮助文档？**

s =‘astring’

dir(s) # 列出所有的字符串的方法名

help(s.upper) # 列出 upper 方法的使用帮助

help(s) # 列出所有字符串类型的帮助文档——不建议使用这种方式

**第五章 数字类型**

数字类型，是一组类似数据类型的归类。包括：整数int、浮点数float、复数complex、固定精度浮点数Decimal、分数Fraction、集合Set、布尔型bool。

本章还包括：官方自带的数学函数、模块

**整数int**

整数包括正负，并且有无穷的精度（不管多大都不会超出内存限额）。

在Python2.x中，如果一个整数超过内存限制，会自动转成 long int，数字后面加个L。但是Python3.x中，只有int，无穷精度。

**数字常量表示方式：二进制八进制十进制十六进制 p134、p118**

**浮点数 float**

1.0 2.3 3.14e-10

算术中，只要出现浮点数float，就自动将相关的数据转成浮点数的精度和运算法则（相当于C语言的double）

a = 1.0 + 5 # 6.0

b = 1 + 5 # 6

hex()、oct()、bin() 把一个十进制数值转成相应进制的字符串，而 int(str, base)则把字符串按指定的进制，转成十进制整数。

小数、十进制数（固定精度浮点数）Decimal

Decimal提供了固定小数位的浮点数（所谓固定精度），还可以定义舍入的方式（比如四舍五入）。

两种方式创建：整数int，字符串str

import decimal

a = decimal.Decimal(1)/decimal.Decimal(10)

from decimal import Decimal

a = Decimal('0.1')

创建时可以设置小数运算的精度

import decimal

a = decimal.Decimal(1) / decimal.Decimal(7)

print(a) #不设精度，无限循环小数0.1428571428571428571428571429

decimal.getcontext().prec = 2 # 设定精度

b = decimal.Decimal(1) / decimal.Decimal(7)

print(b) # 0.14 固定位数小数

还可以用with 环境管理器设置精度

import decimal

with decimal.localcontext() as context:

context.prec = 2

b = decimal.Decimal(1) / decimal.Decimal(7)

print(b) # 0.14

a = decimal.Decimal(1) / decimal.Decimal(7)

print(a) #离开with，精度还原 0.1428571428571428571428571429

分数 Fraction

from fractions import Fraction

x = Fraction(1,3) # 2个整数，创建分数

y = Fraction(4,6)

m = Fraction('1.25') # 用一个字符串、里面是浮点数，创建分数 5/4

接下来，可以像普通的数字一样使用分数 x+y, x-y, x/y, x\*y, x\*\*y

分数与小数，二者是近亲。浮点数因为内存空间有限，所以精度有限，而分数和小数能提供更准确、更直观的结果。

更准确：

from decimal import Decimal; from fractions import Fraction

a\_float = 0.1 + 0.1 + 0.1 - 0.3 # 本应为 0，却不为0，因为float的精度不够

a\_decimal = Decimal('0.1')+Decimal('0.1')+Decimal('0.1')-Decimal('0.3') # 结果 0.0

a\_fraction = Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10) # 结果 0

更直观：分数，是小学生的知识，简单易懂；而固定位数的小数 0.33，也比 杂乱的浮点数 0.33333333 更易读

import decimal

from fractions import Fraction

with decimal.localcontext() as context:

context.prec = 2

a\_float = 1/3 # 0.3333333333333333

a\_decimal = decimal.Decimal(1)/decimal.Decimal(3) # 0.33

a\_fraction = Fraction(1, 3) # 1/3

集合 set

一些唯一的、不可变的对象的一个无序集合/Collection。支持一般的数学集合操作。广泛应用于数学和数据库工作中。

集合set既不是序列，也不是映射。集合很像一个只有key没有value的字典：可迭代、可增减、能包含各种类型的对象。

支持len()，可迭代。但不能序列操作——索引和切片。

x = set('abcde') # 传入一个可迭代对象，构造函数自动创建一个set，结果{'a','b','c','d','e'}

a\_dict = {'d': 1, 'e': 2, 'x': 3, 'y': 2, 'z': 3}

y = set(a\_dict) # 字典的key构建了set ，结果{'e', 'd', 'x', 'y', 'z'}

print('e' in x) # 布尔运算：成员测试，'e'是集合x的成员吗？

print(x|y) # 数学求并集 {'a', 'b', 'c', 'd', 'e', 'x', 'y', 'z'}

print(x&y) # 交集 {'d', 'e'}

print(sorted(x^y)) # 并集，然后减去交集。也就是 两个集合不重合的部分 {'a', 'b', 'c', 'x', 'y', 'z'}

print(x>y) # 布尔运算，子集测试，y是x的子集吗？

P146集合操作的各种方法

创建空集合 s1 = set() # dict = {} 这是创建空字典

如果一个对象是可变的（比如列表list），那么它不可散列Unhashable，就不能作为集合set的元素，也不能作为字典dict的键。

布尔值 bool P152

连续布尔值运算：

x,y,z = 2,4,6

if x<y<z: pass # 相当于 x<y and y<z

if x<y>z:pass # 相当于 x<y and y>z

if x==y<z:pass # 相当于 x==y and y<z

占位符 None

二进制，位操作 p136

移位

或运算

与运算

**类型转换**

向上转型，自动转换：int<float<complex

不可以在数字类型和其他类型之间使用自动转换，比如 a = ‘some’+ 10

向下转型，强制转换，将会截尾：int(3.1415) 为 3

**运算符重载**

某些操作符能够根据所处理的内置对象的类型，而执行不同的操作。比如

a = 10 + 5 # 数字型，执行加法

a = ‘some’+ ‘body’ # 对于 str 或者 list，执行合并

**显示格式**

num = 1/3

r\_num = repr(num) # 源数据显示模式，可以显示额外的细节

str\_num = str(num) # 用户友好显示模式

# 两个函数都可以把任意对象转换成字符串，但是效果不太相同。

print(r\_num,'\n',str\_num)

内置数学函数：pow、abs、round、int、hex、bin

内置数学模块：random、math

pow(2, 4) # 等价于 2\*\*4 = 16

abs(-42) # 绝对值运算，|-42| = 42

sum([1, 2, 3, 4]) # 求和运算 1+2+3+4 = 10

min([1, 2, 3, 4]) # 求最小值，结果为1。max()求最大值

**import math**

p = math.pi # 3.141592653589793

s = math.sqrt(144) # 求平方根方式 1 ，结果为 12.0

s = 144 \*\* 0.5 # 求平方根方式 2

s = pow(144, 0.5) # 求平方根方式 3

**import random**

random.random() # 产生一个随机的浮点数，比如 0.9139864997839369

random.randint(1, 10) # 产生一个1-10之间的随机整数

random.choice(['a', 'b', 'c']) # 随机选择

其他扩展：

* + Numpy（Numeric Python）是第三方模块，被称为 免费易用的Matlab。支持矩阵、向量、高级计算。
  + Scipy

p120 Python 表达式操作符

数据类型

2017年3月31日

22:25

* + **六个标准数据类型：**

Number（数字）、String（字符串）、List（列表）、Tuple（元组）、Sets（集合）、Dictionary（字典）

可变数据类型：列表list[ ]、字典dict{ }

不可变数据类型：整型int、字符串str' '、元组tuple（）

**数据类型**

五、空值：空值是Python里一个特殊的值，用None表示。None不能理解为0，因为0是有意义的，而None是一个特殊的空值。

此外，Python还提供了列表、字典等多种数据类型，还允许创建自定义数据类型，我们后面会继续讲到

来自 <<http://www.imooc.com/code/3255>>

**数字类型**

整数／浮点数／复数／分数／集合／布尔值

\*\* 乘方

math 模块有很多的数学特有操作

序列：包含其他对象的有序集合

s = ‘spam’

索引

print s[0] # 正向索引index，从左至右

print s[-1] # 反向索引，从右至左，相当于 s[len(s)-1]

切片

s[ M : N ] 切片，在序列 s 中，取出偏移量为M，直到但不包括偏移量为 N 的内容，最后得到一个**新对象**。

常见切片

s[1:] # 去除首位

s[:-1] # 去除末位

s[:] # 完整复制

注意，序列绝对不能越界访问。

len(s) #计算出长度

某类序列特定的操作

astring.upper() # 通用操作与特有操作是不同的

不可变性：字符串／元组／数字创建后就不可变，而列表和字典可变。

s = ‘something’

s = ‘do ’ + s # 变量 s 的确变为 ‘do something’, 但这是形成新的字符串对象，引用赋予变量s

**字符串**

字符串是由很多’单字符串’构成的序列。Python没有 Char类型，每个字母都是字符串。

字符串支持通用的序列操作。

字符串拼接：见官方 Doc

字符串特有的函数：find replace lower upper split join strip

字符串格式化：% 与 format

模式匹配 re 模块：搜索／分割／替换，但因为它支持模式定义，所以更通用。

 序列是其他对象的有序集合，通过位置索引。字典是其他对象的无序集合，通过键索引。

**列表**

最通用的序列。

列表是一个任意类型的对象的位置相关的有序集合。大小可变。

L.append(’s’) # 在末尾添加一个对象

L.pop(2) # 指定删除一个索引值的对象

del L[2] # 指定删除一个索引值的对象

insert／remove

L.sort() # 按照 ASCII 生序排列

L.reverse() # 将原来的顺序 ，反向排列

列表可以嵌套，最常用的是组建多维数组（最常见的是矩阵）和 json 这样的复杂数据。

**列表解析式**

使用多维数组时，列表解析非常实用。

M = [ [1,2,3],

[4,5,6],

[7,8,9] ]

1.简单解析。col2 = [ row[1] for row in M ] # [2,5,8]

列表解析式 由 表达式 row[1] 和表达式 for row in M 组成，前面的表达式负责取值和运算，后面的表达式是循环迭代结构，注意两个表达式 有个变量名 row是相关联的。解析式被[ ] 包围，这说明结果是一个列表。

本质是一个 for 循环

col2 = []

for row in M:

col2.append(row[1])

2. 解析后运算 new\_col2 = [ row[1] + 10 for row in M ] # [12, 15, 18]

3. 解析后过滤 new\_col2x = [ row[1] for row in M if row[1] < 7] # [2, 5]

4. 多个列表交互解析 collect a diagonal from matrix

diag = [ M[i][i] for i in [0,1,2] ] # 结果 [1，5，9]，也可以写成 for i in range(3)

5. 创建集合和字典

col2\_set = {row[1] for row in M } # 结果{2,5,8}

col2\_dic = {i:row[1] for i in range(1,4)} # 结果{1:2, 2:5, 3:8}

6. 解析式创建生成器

Gener = (sum(row) for row in M)

next(Gener) # 结果为6

**P63-P70，模块**

序列：Str/List/Tuple

映射：dict

**字典**

字典是一种映射，也是Python中唯一的映射。映射是一个其他对象的集合，通过键值对来存储对象。大小可变。

d = {} # 创建空字典

d = {‘food’：‘spam’，‘weight’：4，‘color‘：’red‘} # 用键值对创建一个字典

d[’food‘] # 用键读取值

d[’weight‘] += 1 # 重新赋值，就是改变key对应的value

d[‘name’] = ‘apple’ # 增加一个键值对。可以创建{}，然后不断增加键值对

字典和列表一起，可以轻松地组建 json那样的复杂／高级数据结构。C／C++需要先设计／声明结构和数组，填写值，然后拼接，非常麻烦。

字典可以迭代遍历key： for key in d：print d[key]

有时候需要按照一定的顺序遍历：

方法1

keylist = list（d.keys() ） # 提取所有的key，组建一个list

for key in keylist.sort(): # list排序，然后遍历

print d[key] # 用key 提取 value

方法 2 for key in sorted(d): print d[key] # 一步完成，效果一样

字典取值，也要特别注意『越界』问题——如果key不存在，会引发异常。避免的手段：

value = d.get(‘x’,0) # 如果’x’键不存在，就使用默认值 0

value = d[‘x’] if ‘x’in d else 0 # 具有相同的效果

还可以用has\_key方法、try语句

for循环是迭代操作。for循环、列表解析（本质也是for循环），是通用的迭代工具，遵循迭代协议。迭代协议：一个内存中的序列、或一个迭代操作中，每次都产生一个元素。

列表解析、map、filter 比 for循环快很多，但Python 程序优先考虑简单和可读性，其次再考虑优化和性能，所以最广泛的还是for循环。

**元组 tuple**

元组，任意类型、任意嵌套。不可变的序列，与列表很相似。

除了序列的共有方法，list的列表修改是无法应用于tuple 的，但是tuple又有自己独特的方法

t = (‘a’,’b’,’c’,’d’)

m = t.index(‘c’) # 得出一个元素的索引值。结果为2

n = t.count(‘c’) # 得出一个元素出现的次数。结果为1

**文件 file**

文件对象不是通过常量语法（比如 l = [1,2,3] ）创建的，而是调用 open() 函数创建的。

写入操作

f = open('data.txt','w') # (文件名，处理模式)

f.write('Hello\n') # 写入一行字符

f.close() # 关闭文件

读取操作

f = open('data.txt') # 处理模式默认值为 'r'

text = f.read() # 读取文件内所有的内容（或者读取指定的字节数），存入变量

print(text) # 将内容打印

f.close() # 关闭文件

text.split() # 内容可以继续进行处理

readline() 每次读取一行。seek() 移动到一个新的文件位置。

文件读取的最好的方式，是使用迭代器 iterator

在Python3 中，文本与二进制的文件对象是截然不同的。

f = open('data.txt') # 文本文件，内容解析成字符串，默认执行Unicode编码

f = open('data.bin','rb') # 二进制文件，解析二进制

还有其他文件类工具：管道、先进先出队列FIFO、套接字Socket、数据库接口对象、通过键访问文件、对象持久化、基于描述符的文件

其他核心类型：集合set、固定精度浮点数、分数Fraction、布尔值bool、占位符None

12.None对象是一个特殊的Python对象，它总是False，一般用于占位。它有一块内存，是一个真正的对象。它不代表未定义，事实上它有定义。None是所有函数和方法的默认返回值

类型与动态类型

2017年3月31日

22:07

动态类型简介

Python 是动态类型语言。也是强类型语言，特定的对象，只能执行特定的操作。

在Python中，处理的所有事情都是某种类型的对象（内存中的数据）。任何东西都是对象，类型本身也是对象。type(x)返回变量x指向对象的类型，isinstance( x, typename )用于测试x所指对象是否是typename类型。调用类型名是对这些对象构造函数的调用，而不仅仅是类型转换。

对象不等于面向对象，在Python中，面向对象是可选的。

动态类型和多态，是Python语言简洁和灵活的根源。

Python 只有一种赋值模型，所以理解了动态类型，就随处可用了。

**初始化赋值：变量、对象、引用**

初始化赋值语句

a = 3

* + 对象：等式右边，常量 3 创建了一个新的对象（一块内存空间）去表示这个值。

    - 每次运行一个表达式（运算或者常量赋值）生成一个新的值，Python都创建一个新的对象（内存空间）去表示这个值。Python对象缓存了不变的对象并对其进行复用，如小整数和小字符串。但是逻辑上看，每个表达式的结果值都是不同的对象，占用不同的内存。
    - 当变量出现在表达式中时，它会马上被当前引用的对象所代替。
    - 每个对象都有2个标准的头部信息：数据类型标识符、引用计数器。

类型标志符：标识了该对象的类型 。查看对象的类型用type()函数

引用计数器：决定了是否可以回收这个对象。sys模块的getrefcount函数会返回对象的引用次数。Python对象缓存了不变的对象并对其进行复用。引用计数器记录了当前指向该对象的引用的数目，一旦它为0， 则该对象的内存空间就会自动回收（垃圾收集）。比如 a = "something" ，重新赋值，对象 3 就垃圾回收。

* + 如果一个变量重新赋值，并不是替换原始的对象，而是让这个变量去引用完全不同的一个对象。之前那个对象如果没有变量引用了，引用计数器就置零。
  + 变量：等式左边，创建一个变量名 a
    - ，
    - 任何变量名在使用前必须明确地赋值。用未赋值的变量会产生错误。 变量在初始化赋值（第一次赋值）时创建 ，再次赋值，就会改变它的值。
    - 变量名是一个系统表对象，拥有指向对象内存空间的连接（引用）。在Python内部，变量实际上是指向对象内存空间的一个指针。
    - 类型的概念是存在于对象中而不是变量名中。变量名是通用的（一个变量可以引用任何对象），只是在某个特定的时间点引用了一个特定的对象。变量没有类型，所以不需要变量类型声明。类型由对象决定，而不是通过声明来决定的 ，并且是在运行的过程中自动跟踪变量引用的对象的类型。
    - 变量使用时，马上会被引用的对象所代替，比如 c = a 将 a 引用的对象 3 ，再次赋值，引用给 c 。未初始化赋值、创建的变量不准使用。
  + 引用：赋值语句的等式，创建一个变量 a 与对象 3 的引用。变量和对象存储在内存的不同区域，并通过引用相连。
    - 引用就是从变量到对象的内存指针，相当于 C 语言的 void 指针。

**共享引用**

a = 3 # 初始化赋值，创建对象 3、变量 a 、以及相互之间的引用。

b = a # 变量a马上被替换成对象 3，所以b也引用了对象 3——a与b共同引用一块内存区域

a = a + 2 # 变量a 替换成对象 3 ，与对象 2 进行运算得到全新的对象 5，然后创建变量a 与对象 5的引用。此语句对 变量 b 没有任何影响。

a = "something" # 变量a 引用新的对象 "something"

对一个变量赋值，仅仅会影响变量，不会影响对象。

因为，在有些语言中，变量是可改变的内存区域的地址，重新赋值，会替换这个内存区域的值。而在Python中，变量是一个指向内存区域的指针，重新赋值，是让这个变量去引用一个新的对象（指向新的内存空间）。

**共享对象与数值相等**

L = [1, 2, 3]

M = [1, 2, 3]

print L == M # True ，因为 == 操作符比较两个对象的数值，数值的确相等。

print L is M # False ，因为 is 操作符比较L与M是不是引用同一个对象（指向同一块内存）

但是，有时候又会出现“意外情况”：

L = 42

M = 42

print L == M # True ，因为 == 操作符比较两个对象的数值，数值的确相等。

print L is M # True ，很奇怪。

因为Python把小的整数缓存，并且复用了，所以L与M的确引用了同一个对象。这是Python决定的，不是代码能够改变的。

查询一个对象被引用的次数：数字对象 1 被引用了 607 次

>>> import sys

>>> sys.getrefcount(1)

607

**共享引用、原处修改可变的对象**

多个变量名引用同一个对象时，称为共享引用。在共享引用中，  
==操作符比较的是两个变量引用的对象是否具有相同的值  
is操作符比较的是两个变量是否引用的是同一个对象

有些数据类型的对象，本身就是对其他对象的引用，所以可以原地修改。比如列表、字典、class。对象的原地修改操作会影响到所有的共享引用变量 。

L1 = [1, 2, 3, 4]

L2 = L1 # 共享引用。L2 也引用了list 对象 [1, 2, 3, 4]

L1 = 89 # L1 引用了全新的对象 89。不影响 L2

但是，下面就会相互影响了

L1 = [1, 2, 3, 4]

L2 = L1 # L1与L2共享引用

L1[0] = 32 # 结果L2 也受到了影响，变为 [32, 2, 3, 4]

避免共享对象带来的影响，可以直接拷贝对象。让不同的变量，引用不同的对象（指向不同的内存区域）

L2 = L1[:] # 列表拷贝最简单的办法——从头到尾切片。

其他的对象，应该使用copy方法

import copy

x = copy.copy(y) # make a shallow copy of y 创建对象 y 的影子拷贝

x = copy.deepcopy(y) # make a deep copy of y 创建对象 y 的深度拷贝

影子拷贝和深度拷贝，差异就在复合对象上（包含其他对象的对象，比如列表和实例）。

影子拷贝是创建一个复合对象，然后内含的对象，还是上一个对象的引用。

深度拷贝，创建一个复合对象，内含的对象，也是新建的。

**检查类型的3种方法**

a\_list = [1,2]

if type(a\_list) == type([]): print('it is a list')

if type(a\_list) == list:print('it is a list')

if isinstance(a\_list,list): print('it is a list')

如上所示，Python完全有能力做到类型检测，将代码限制在某种类型上，但大多数时候是不必要的。这破坏了代码的灵活性，限制了代码的类型，违背了Python的动态类型和多态思想。动态类型是Python多态的基础，因为没有类型约束。

Python的多态是x.method的方法运行时，method的意义取决于x的类型，属性总是在运行期解析。

一个操作的意义取决于被操作对象的类型。同样的操作对不同的对象来说意义可能不同（前提是该对象支持该操作，若对象不支持某种操作，则Python会在运行时检测到错误并自动抛出一个异常 ）。Python代码不关心特定的数据类型，只关心接口，这就是Python的多态设计。

15.Python可以在旧版本中开启新版本的特性，只需用在模块文件中使用： from \_\_future\_\_ import 特性名

笔记：模块

2017年3月29日

21:22

Python的程序结构：

Python程序 >> 模块 >> 语句（构建基本工具，函数、类等等）>>表达式（建立并处理对象）

一个Python程序，是由很多模块组成的系统。一个模块就是一个.py文件（或者是C/C++、Java、C#写的扩展）。

* 唯一的顶层文件（或叫顶层脚本），包含了程序的主要控制流程，启动就是运行整个程序（相当于main入口函数）。
* 其他的模块文件，是工具库，放置各种支撑的组件（函数、类等等）。

模块+导入，大程序切分为不同的软件组件，不同模块之间的代码相互隔离，运行时通过导入操作连接在一起，就是Python程序架构的核心。

模块的3个作用：

* 复用代码和共享。交互模式，输入就运行，退出就没有了。保存后的.py文件，可以无数次载入和运行，还可以拷贝给别人。
* 作为一个命名空间。模块是一个定义变量名的空间，可以避免变量名的冲突。
* 组建工程。模块是Python程序中最高级别的组织单元。将各个工具部件组成一个系统（将各种函数和类组成一个.py文件）。

导入链要多深，就有多深：顶层文件>>模块>>第三方库>>标准库（权重依次递减）

# 模块.py文件

建立一个.py文件，就是一个模块。如果需要被导入，必须以.py结尾。顶层文件，仅运行而不被导入，可以不用.py结尾，但最好用.py结尾，编辑的时候代码高亮。

包和模块，导入后都成为一个变量名，所以遵循普通变量名的命名法则。

|  |  |
| --- | --- |
| 模块内部的通常布局 | 范例 demo.py |
| 1.Shebang (也称Hashbang)，指定Python解析器路径。  2. 指定文件编码。  3.模块文档字符串。  4.导入。  5.全局变量。    6.类定义。文档字符串。      7.函数定义。文档字符串。    8.主程序。一般用来写单元测试。 | **#!/usr/bin/env Python3**  **# -\*- coding: utf-8 -\*-**  **"""This is just a test module"""**  **from platform import \***  **string = 'module'**  **class Test():**  """\_\_Doc\_\_"""  def \_\_init\_\_(self):  print('run \_\_init\_\_', end='\n')  **def foo():**  """\_\_Doc\_\_"""  print('run foo()',python\_version())  **if \_\_name\_\_ == '\_\_main\_\_':**  Test()  foo()  print('\_\_main\_\_',python\_version()) |

#!/usr/bin/python指定一个路径里的Python解析器，系统Python解析器（#!/usr/bin/python指定Python3）。

#!/usr/local/bin/python 指定一个路径里的Python解析器，系统Python解析器

**#!/usr/bin/env python 指出要Python2，但让系统搜寻路径（#!/usr/bin/env python3 指出要Python3）——建议。**

指定解析器的检验方法：写出平台特有的语法，然后直接运行py文件。如上面的例子。

chmod u+x demo.py # 更改文件的模式，变为可执行模式

./demo.py # 运行这个文件

Python 2的 print 2无法在Python3运行，而Python3 的 print('run \_\_init\_\_',end='\n') 无法在2运行。

python hello.py是指定python2运行，python3 hello.py是指定python3运行，都不是直接运行模块。

# import 语句——导入整个模块

这与C语言的 #include语句有点类似，但不一样。#include是将一个文件的语句插入另一个文件。

import 的 3个步骤：

* 识别这个文件，编译成.pyc文件（只在必要时），加载这个模块

识别的路径：sys.path包括：程序的主目录，PythonPATH，标准库，路径文件.pth 定义的路径（权重依次递减）。主目录和标准库是自定义的，无需设置；PythonPATH 和 .pth 用于扩展路径，作为补充。

1. 程序主目录：交互模式，就是当前目录。创建一个工程或者运行程序时，就是顶层脚本所在的目录。最先搜索，权重也最高，可能会覆盖其他地方的同名模块。
2. PythonPATH：当主目录与支撑模块所在的目录不同时，才会用到这个方式。比如，控制面板-系统与安全-系统-高级系统设置-环境变量，建立一个 PythonPATH，填入C:\pycode\utilities;D:\pycode\package
3. 标准库：Python自带的官方库，无需设置，直接import
4. 路径文件.pth ：是PythonPATH的替代品，设置更简单，并且设置一次可以让所有用户使用。Windows的，C:\Python30\myconfig.pth 或 C:\Python30\Lib\sitepackages\myconfig.pth（常用来定义第三方库）；Unix的，usr/local/lib/site-python 或 usr/local/lib/python30/site-packages。比如，在 C:/Python30/pydirs.pth 文本文件加入

C:\pycode\utilities

D:\pycode\package

print(sys.path)来检查自己的设置是否正确

语句 import b 可能并不是加载 .py文件，它可能是：

* b.py文件或b.pyc文件，或者.pyo优化文件。只在必要的时候才编译，比如.pyc不存在，第一次编译，比如.py文件修改后再次运行程序需要重新编译。只有被导入的文件才会留下.pyc。如果不存在.py，会直接加载.pyc。
* 目录b——包导入
* C/C++模块：动态链接导入扩展模块（b.so、b.dll、b.pyd之类），或者静态链接导入已经编译好的模块。
* .zip压缩包。导入时自动解压。
* frozen 二进制文件，导入的是内存中的模块。
* java类，Jython。.net组件，IronPython

从sys.path中，从左至右一个个路径开始搜寻b，找到一个b.x文件（后缀必须合法）就导入——这就是覆盖原理。最好不要在同一目录下出现 b.py 和 b.dll这样的情况。

* 将变量名赋值给这个模块对象。隐性赋值，效果和赋值语句一样，都是将一个变量名指向内存中的一块区域。
* 运行模块中的语句，建立其中的顶层代码的内容（也就是模块里面零级缩进代码所定义的对象，比如类、函数、变量）。之后，就能通过变量名获取其中的所有的属性了。

一个文件导入了一个模块，就获得了其中的各种工具的读取访问权。导入模块，获取它的属性，使用它的工具。导入后，一个模块文件的顶层定义的（零级缩进的）所有变量名都成了被导入的模块对象的属性。

对象属性语法：object.attribute

* 调用函数：random.random()——调用模块random的random函数；self.run()——调用一个对象的run()函数
* 取得数据：math.pi——调用模块math的pi变量值；self.name——调用一个对象的name属性

导入的模块，存在sys.modules的字典里，每次导入都会检查（用print(list(sys.modules.keys()))查看）。首次导入才会运行这3步，之后如果导入相同的模块，就在内存中提取。这样可以节省系统开销。

# from … import a, b——导入特定的变量名

# imp.reload——不停止程序时，重新载入模块

目录

2017年3月31日

22:43

对象类型

类型概述，动态类型

数字类型

字符串

列表与元组

字典

文件

语句与语法

语句概述，编程普遍规范（变量名规则，注释，docstring，）

赋值

表达式，打印

If

While

迭代器和解析

函数

函数概述

作用域

参数

函数高级话题

迭代器和解析

模块

模块基础

类和OOP

异常和工具