

# 目錄

Python基础	0
认识python和基础知识	1
认识python(了解)	1.1
第一个python程序	1.2
注释	1.3
变量以及类型	1.4
标示符和关键字	1.5
输出	1.6
输入	1.7
运算符	1.8
数据类型转换	1.9
判断语句介绍	1.10
if语句	1.11
比较、关系运算符	1.12
作业	1.13
附录-推荐的python电子书	1.14
判断语句和循环语句	2
if-else	2.1
elif	2.2
if嵌套	2.3
if应用:猜拳游戏	2.4
循环语句介绍	2.5
while循环	2.6
while循环应用	2.7

while循环的嵌套以及应用	2.8
for循环	2.9
break和continue	2.10
总结	2.11
作业	2.12
字符串、列表、元组、字典	3
字符串介绍	3.1
字符串输出	3.2
字符串输入	3.3
下标和切片	3.4
字符串常见操作	3.5
列表介绍	3.6
列表的循环遍历	3.7
列表的常见操作	3.8
列表的嵌套	3.9
元组	3.10
字典介绍	3.11
字典的常见操作1	3.12
字典的常见操作2	3.13
字典的遍历	3.14
公共方法	3.15
引用	3.16
作业	3.17
函数	4
函数介绍	4.1
函数定义、调用	4.2
函数的文档说明	4.3

函数参数(一)	4.4
函数返回值(一)	4.5
4种函数的类型	4.6
函数的嵌套调用	4.7
函数应用：打印图形和数学计算	4.8
局部变量	4.9
全局变量	4.10
函数应用：学生管理系统	4.11
函数返回值(二)	4.12
函数参数(二)	4.13
递归函数	4.14
匿名函数	4.15
函数使用注意事项	4.16
作业	4.17
文件操作、综合应用	5
文件操作介绍	5.1
文件的打开与关闭	5.2
文件的读写	5.3
应用1:制作文件的备份	5.4
文件的定位读写	5.5
文件的重命名、删除	5.6
文件夹的相关操作	5.7
应用2:批量修改文件名	5.8
综合应用:学生管理系统(文件版)	5.9
作业	5.10
面向对象1	6
面向对象编程介绍	6.1

类和对象	6.2
定义类	6.3
创建对象	6.4
__init__方法	6.5
应用:创建多个对象	6.6
"魔法"方法	6.7
self	6.8
应用:烤地瓜	6.9
隐藏数据	6.10
应用:存放家具	6.11
面向对象2	7
应用:老王开枪	7.1
保护对象的属性	7.2
__del__方法	7.3
单继承	7.4
多继承	7.5
重写父类方法与调用父类方法	7.6
多态	7.7
类属性、实例属性	7.8
静态方法和类方法	7.9
面向对象3、异常、模块	8
练习：设计类	8.1
工厂模式	8.2
__new__方法	8.3
单例模式	8.4
异常介绍	8.5
捕获异常	8.6

异常的传递	8.7
抛出自定义的异常	8.8
异常处理中抛出异常	8.9
模块介绍	8.10
模块制作	8.11
模块中的__all__	8.12
python中的包	8.13
模块发布	8.14
模块安装、使用	8.15
 强化练习	9
给程序传参数	9.1
列表推导式	9.2
set、list、tuple	9.3
面试题1	9.4
面试题2	9.5
 应用:打飞机	10
打飞机代码：搭建界面	10.1
打飞机代码：检测键盘	10.2
打飞机代码：显示、控制玩具飞机-面向过程	10.3
打飞机代码：显示、控制玩具飞机-面向对象	10.4
打飞机代码：玩家飞机发射子弹	10.5
打飞机代码：显示敌机	10.6
打飞机代码：优化代码	10.7
打飞机代码：让敌机移动	10.8
打飞机代码：敌机发射子弹	10.9
打飞机代码：代码优化-抽象出基类	10.10







# 认识python(了解)

## 1. Python发展历史

- 起源

Python的作者， Guido von Rossum， 荷兰人。1982年， Guido从阿姆斯特丹大学获得了数学和计算机硕士学位。然而， 尽管他算得上是一位数学家， 但他更加享受计算机带来的乐趣。用他的话说， 尽管拥有数学和计算机双料资质， 他总趋向于做计算机相关的工作，并热衷于做任何和编程相关的活儿。

那个时候， Guido接触并使用过诸如Pascal、 C、 Fortran等语言。这些语言的基本设计原则是让机器能更快运行。在80年代， 虽然IBM和苹果已经掀起了个人电脑浪潮， 但这些个人电脑的配置很低。比如早期的Macintosh， 只有8MHz的CPU主频和128KB的RAM， 一个大的数组就能占满内存。所有的编译器的核心是做优化， 以便让程序能够运行。为了增进效率， 语言也迫使程序员像计算机一样思考， 以便能写出更符合机器口味的程序。在那个时代， 程序员恨不得用手榨取计算机每一寸的能力。有人甚至认为C语言的指针是在浪费内存。至于动态类型， 内存自动管理， 面向对象…… 别想了， 那会让你的电脑陷入瘫痪。

这种编程方式让Guido感到苦恼。Guido知道如何用C语言写出一个功能， 但整个编写过程需要耗费大量的时间， 即使他已经准确的知道了如何实现。他的另一个选择是shell。Bourne Shell作为UNIX系统的解释器已经长期存在。UNIX的管理员们常常用shell去写一些简单的脚本， 以进行一些系统维护的工作， 比如定期备份、 文件系统管理等等。shell可以像胶水一样， 将UNIX下的许多功能连接在一起。许多C语言下上百行的程序，在shell下只用几行就可以完成。然而， shell的本质是调用命令。它并不是一个真正的语言。比如说， shell没有数值型的数据类型， 加法运算都很复杂。总之， shell不能全面的调动计算机的功能。

Guido希望有一种语言，这种语言能够像C语言那样，能够全面调用计算机的功能接口，又可以像shell那样，可以轻松的编程。ABC语言让Guido看到希望。ABC是由荷兰的数学和计算机研究所开发的。Guido在该研究所工作，并参与到ABC语言的开发。ABC语言以教学为目的。与当时的大部分语言不同，ABC语言的目标是“让用户感觉更好”。ABC语言希望让语言变得容易阅读，容易使用，容易记忆，容易学习，并以此来激发人们学习编程的兴趣。比如下面是一段来自Wikipedia的ABC程序，这个程序用于统计文本中出现的词的总数：

```
HOW TO RETURN words document:  
PUT {} IN collection  
FOR line IN document:  
    FOR word IN split line:  
        IF word not.in collection:  
            INSERT word IN collection  
RETURN collection
```

HOW TO用于定义一个函数。一个Python程序员应该很容易理解这段程序。ABC语言使用冒号和缩进来表示程序块。行尾没有分号。for和if结构中也没有括号()。赋值采用的是PUT，而不是更常见的等号。这些改动让ABC程序读起来像一段文字。尽管已经具备了良好的可读性和易用性，ABC语言最终没有流行起来。在当时，ABC语言编译器需要比较高配置的电脑才能运行。而这些电脑的使用者通常精通计算机，他们更多考虑程序的效率，而非它的学习难度。除了硬件上的困难外，ABC语言的设计也存在一些致命的问题：可拓展性差。ABC语言不是模块化语言。如果想在ABC语言中增加功能，比如对图形化的支持，就必须改动很多地方。不能直接进行IO。ABC语言不能直接操作文件系统。尽管你可以通过诸如文本流的方式导入数据，但ABC无法直接读写文件。输入输出的困难对于计算机语言来说是致命的。你能想像一个打不开车门的跑车么？过度革新。ABC用自然语言的方式来表达程序的意义，比如上面程序中的HOW TO。然而对于程序员来说，他们更习惯用function或者define来定义一个函数。同样，程序员更习惯用等号来分配变量。尽管ABC语言很特别，但学习难度也很大。传播困难。ABC编译器很大，

必须被保存在磁带上。当时Guido在访问的时候，就必须有一个大磁带来给别人安装ABC编译器。这样，ABC语言就很难快速传播。1989年，为了打发圣诞节假期，Guido开始写Python语言的编译器。Python这个名字，来自Guido所挚爱的电视剧Monty Python's Flying Circus。他希望这个新的叫做Python的语言，能符合他的理想：创造一种C和shell之间，功能全面，易学易用，可拓展的语言。Guido作为一个语言设计爱好者，已经有过设计语言的尝试。这一次，也不过是一次纯粹的hacking行为。

- 一门语言的诞生

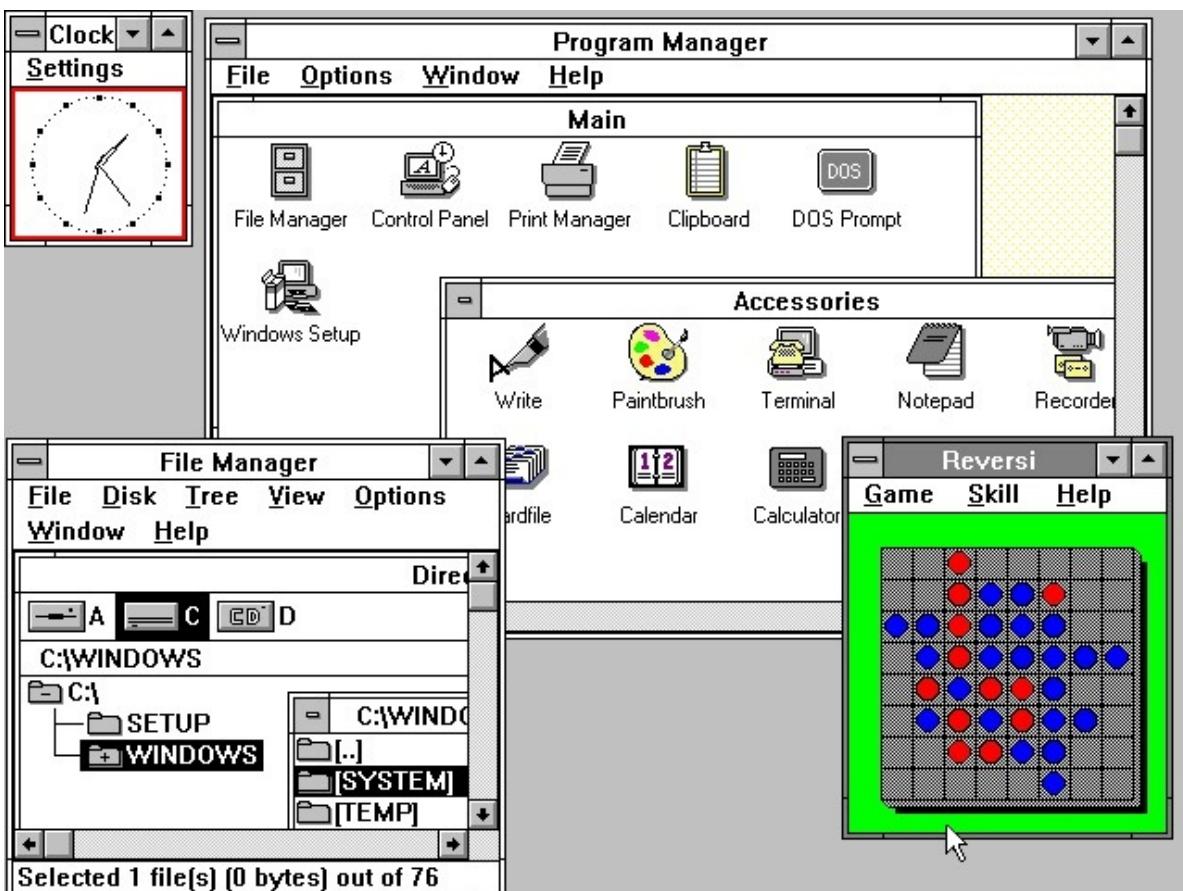
1991年，第一个Python编译器诞生。它是用C语言实现的，并能够调用C语言的库文件。从一出生，Python已经具有了：类，函数，异常处理，包含表和词典在内的核心数据类型，以及模块为基础的拓展系统。Python语法很多来自C，但又受到ABC语言的强烈影响。来自ABC语言的一些规定直到今天还富有争议，比如强制缩进。但这些语法规规定让Python容易读。另一方面，Python聪明的选择服从一些惯例，特别是C语言的惯例，比如回归等号赋值。Guido认为，如果“常识”上确立的东西，没有必要过度纠结。Python从一开始就特别在意可拓展性。Python可以在多个层次上拓展。从高层上，你可以直接引入.py文件。在底层，你可以引用C语言的库。Python程序员可以快速的使用Python写.py文件作为拓展模块。但当性能是考虑的重要因素时，Python程序员可以深入底层，写C程序，编译为.so文件引入到Python中使用。Python就好像是使用钢构建房一样，先规定好大的框架。而程序员可以在此框架下相当自由的拓展或更改。最初的Python完全由Guido本人开发。Python得到Guido同事的欢迎。他们迅速的反馈使用意见，并参与到Python的改进。Guido和一些同事构成Python的核心团队。他们将自己大部分的业余时间用于hack Python。随后，Python拓展到研究所之外。Python将许多机器层面上的细节隐藏，交给编译器处理，并凸显出逻辑层面的编程思考。Python程序员可以花更多的时间用于思考程序的逻辑，而不是具体的实现细节。这一特征吸引了广大的程序员。Python开始流行。



人生苦短，我用python

- 时势造英雄

我们不得不暂停我们的Python时间，转而看一看瞬息万变的计算机行业。1990年代初，个人计算机开始进入普通家庭。Intel发布了486处理器，windows发布window 3.0开始的一系列视窗系统。计算机的性能大大提高。程序员开始关注计算机的易用性，比如图形化界面。



## Windows 3.0

由于计算机性能的提高，软件的世界也开始随之改变。硬件足以满足许多个人电脑的需要。硬件厂商甚至渴望高需求软件的出现，以带动硬件的更新换代。C++和Java相继流行。C++和Java提供了面向对象的编程范式，以及丰富的对象库。在牺牲了一定的性能的代价下，C++和Java大大提高了程序的产量。语言的易用性被提到一个新的高度。我们还记得，ABC失败的一个重要原因是硬件的性能限制。从这方面说，Python要比ABC幸运许多。另一个悄然发生的改变是Internet。1990年代还是个人电脑的时代，windows和Intel挟PC以令天下，盛极一时。尽管Internet为主体的信息革命尚未到来，但许多程序员以及资深计算机用户已经在频繁使用Internet进行交流，比如使用email和newsgroup。Internet让信息交流成本大大下降。一种新的软件开发模式开始流行：开源。程序员利用业余时间进行软件开发，并开放源代码。1991年，Linus在comp.os.minix新闻组上发布了Linux内核源代码，吸引大批hacker的加入。Linux和GNU相互合作，最终构成了一个充满活力的开源平台。硬件性能不是瓶颈，Python又容易使用，所以许多人开始转向Python。Guido维护了一个maillist，Python用户就通过邮件进行交流。

Python用户来自许多领域，有不同的背景，对Python也有不同的需求。Python相当的开放，又容易拓展，所以当用户不满足于现有功能，很容易对Python进行拓展或改造。随后，这些用户将改动发给Guido，并由Guido决定是否将新的特征加入到Python或者标准库中。如果代码能被纳入Python自身或者标准库，这将极大的荣誉。由于Guido至高无上的决定权，他因此被称为“终身的仁慈独裁者”。Python被称为“Battery Included”，是说它以及其标准库的功能强大。这些是整个社区的贡献。Python的开发者来自不同领域，他们将不同领域的优点带给Python。比如Python标准库中的正则表达是参考Perl，而lambda, map, filter, reduce等函数参考了Lisp。Python本身的一些功能以及大部分的标准库来自于社区。Python的社区不断扩大，进而拥有了自己的newsgroup，网站，以及基金。从Python 2.0开始，Python也从maillist的开发方式，转为完全开源的开发方式。社区气氛已经形成，工作被整个社区分担，Python也获得了更加高速的发展。到今天，Python的框架已经确立。Python语言以对象为核心组织代码，支持多种编程范式，采用动态类型，自动进行内存回收。Python支持解释运行，并能调用C库进行拓展。Python有强大的标准库。由于标准库的体系已经稳定，所以Python的生态系统开始拓展到第三方包。这些包，如Django、web.py、wxpython、numpy、matplotlib、PIL，将Python升级成了物种丰富的热带雨林。

- 启示录

Python崇尚优美、清晰、简单，是一个优秀并广泛使用的语言。Python在TIOBE排行榜中排行第八，它是Google的第三大开发语言，Dropbox的基础语言，豆瓣的服务器语言。Python的发展史可以作为一个代表，带给我许多启示。在Python的开发过程中，社区起到了重要的作用。Guido自认为自己不是全能型的程序员，所以他只负责制订框架。如果问题太复杂，他会选择绕过去，也就是cut the corner。这些问题最终由社区中的其他人解决。社区中的人才是异常丰富的，就连创建网站，筹集基金这样与开发稍远的事情，也有人乐于处理。如今的项目开发越来越复杂，越来越庞大，合作以及开放的心态成为项目最终成功的关键。Python从其他语言中学到了很多，无论是已经进入历史的ABC，还是依然在使用的C和Perl，以及许多没有列出的其他语言。可以说，Python的成功代表了它所有借鉴的语言的成功。同样，Ruby借鉴了Python，它

的成功也代表了Python某些方面的成功。每个语言都是混合体，都有它优秀的地方，但也有各种各样的缺陷。同时，一个语言“好与不好”的评判，往往受制于平台、硬件、时代等等外部原因。程序员经历过许多语言之争。其实，以开放的心态来接受各个语言，说不定哪一天，程序员也可以如Guido那样，混合出自己的语言。

## 关键点常识

- Python的发音与拼写
- Python的意思是蟒蛇，源于作者喜欢的一部电视剧 (C呢？ )
- Python的作者是Guido van Rossum (龟叔)
- Python是龟叔在1989年圣诞节期间，为了打发无聊的圣诞节而用C编写的一个编程语言
- Python正式诞生于1991年
- Python的解释器如今有多个语言实现，我们常用的是CPython（官方版本的C语言实现），其他还有Jython（可以运行在Java平台）、IronPython（可以运行在.NET和Mono平台）、PyPy（Python实现的，支持JIT即时编译）
- Python目前有两个版本，Python2和Python3，最新版分别为2.7.12和3.5.2，现阶段大部分公司用的是Python2
- Life is short, you need Python. 人生苦短，我用Python。
- 2017年1月份 编程语言流行排行榜

Jan 2017	Jan 2016	Change	Programming Language	Ratings	Change
1	1		Java	17.278%	-4.19%
2	2		C	9.349%	-6.69%
3	3		C++	6.301%	-0.61%
4	4		C#	4.039%	-0.67%
5	5		Python	3.465%	-0.39%
6	7	▲	Visual Basic .NET	2.960%	+0.38%
7	8	▲	JavaScript	2.850%	+0.29%
8	11	▲	Perl	2.750%	+0.91%
9	9		Assembly language	2.701%	+0.61%
10	6	▼	PHP	2.564%	-0.14%
11	12	▲	Delphi/Object Pascal	2.561%	+0.78%
12	10	▼	Ruby	2.546%	+0.50%
13	54	▲	Go	2.325%	+2.16%
14	14		Swift	1.932%	+0.57%
15	13	▼	Visual Basic	1.912%	+0.23%
16	19	▲	R	1.787%	+0.73%
17	26	▲	Dart	1.720%	+0.95%
18	18		Objective-C	1.617%	+0.54%
19	15	▼	MATLAB	1.578%	+0.35%
20	20		PL/SQL	1.539%	+0.52%

<http://blog.csdn.net/qi123>

## 2. Python优缺点

### 优点

- 简单——Python是一种代表简单主义思想的语言。阅读一个良好的Python程序就感觉像是在读英语一样，尽管这个英语的要求非常严格！Python的这种伪代码本质是它最大的优点之一。它使你能够专注于解决问题而不是去搞明白语言本身。
- 易学——就如同你即将看到的一样，Python极其容易上手。前面已经提到了，Python有极其简单的语法。
- 免费、开源——Python是FLOSS（自由/开放源码软件）之一。简单地说，你可以自由地发布这个软件的拷贝、阅读它的源代码、对它做改动、把它的一部分用于新的自由软件中。FLOSS是基于一个团体分享知

识的概念。这是为什么Python如此优秀的原因之一——它是由一群希望看到一个更加优秀的Python的人创造并经常改进着的。

- **高层语言**——当你用Python语言编写程序的时候，你无需考虑诸如如何管理你的程序使用的内存一类的底层细节。
- **可移植性**——由于它的开源本质，Python已经被移植在许多平台上（经过改动使它能够工作在不同平台上）。如果你小心地避免使用依赖于系统的特性，那么你的所有Python程序无需修改就可以在下述任何平台上面运行。这些平台包括Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2、Amiga、AROS、AS/400、BeOS、OS/390、z/OS、Palm OS、QNX、VMS、Psion、Acom RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE甚至还有PocketPC、Symbian以及Google基于linux开发的Android平台！
- **解释性**——这一点需要一些解释。一个用编译性语言比如C或C++写的程序可以从源文件（即C或C++语言）转换到一个你的计算机使用的语言（二进制代码，即0和1）。这个过程通过编译器和不同的标记、选项完成。当你运行你的程序的时候，连接/转载器软件把你的程序从硬盘复制到内存中并且运行。而Python语言写的程序不需要编译成二进制代码。你可以直接从源代码运行程序。在计算机内部，Python解释器把源代码转换成称为字节码的中间形式，然后再把它翻译成计算机使用的机器语言并运行。事实上，由于你不再需要担心如何编译程序，如何确保连接转载正确的库等等，所有这一切使得使用Python更加简单。由于你只需要把你的Python程序拷贝到另外一台计算机上，它就可以工作了，这也使得你的Python程序更加易于移植。
- **面向对象**——Python既支持面向过程的编程也支持面向对象的编程。在“面向过程”的语言中，程序是由过程或仅仅是可重用代码的函数构建起来的。在“面向对象”的语言中，程序是由数据和功能组合而成的对象构建起来的。与其他主要的语言如C++和Java相比，Python以一种非常强大又简单的方式实现面向对象编程。

- 可扩展性——如果你需要你的一段关键代码运行得更快或者希望某些算法不公开，你可以把你的部分程序用C或C++编写，然后在你的Python程序中使用它们。
- 丰富的库——Python标准库确实很庞大。它可以帮助你处理各种工作，包括正则表达式、文档生成、单元测试、线程、数据库、网页浏览器、CGI、FTP、电子邮件、XML、XML-RPC、HTML、WAV文件、密码系统、GUI（图形用户界面）、Tk和其他与系统有关的操作。记住，只要安装了Python，所有这些功能都是可用的。这被称作Python的“功能齐全”理念。除了标准库以外，还有许多其他高质量的库，如wxPython、Twisted和Python图像库等等。
- 规范的代码——Python采用强制缩进的方式使得代码具有极佳的可读性。

## 缺点

1. 运行速度，有速度要求的话，用C++改写关键部分吧。
2. 国内市场较小（国内以python来做主要开发的，目前只有一些web2.0公司）。但时间推移，目前很多国内软件公司，尤其是游戏公司，也开始规模使用他。
3. 中文资料匮乏（好的python中文资料屈指可数）。托社区的福，有几本优秀的教材已经被翻译了，但入门级教材多，高级内容还是只能看英语版。
4. 构架选择太多（没有像C#这样的官方.net构架，也没有像ruby由于历史较短，构架开发的相对集中。Ruby on Rails 构架开发中小型web程序天下无敌）。不过这也从另一个侧面说明，python比较优秀，吸引的人才多，项目也多。

## 3. Python应用场景

- Web应用开发

Python经常被用于Web开发。比如，通过mod\_wsgi模块，Apache可以运行用Python编写的Web程序。Python定义了WSGI标准应用接口来协调Http服务器与基于Python的Web程序之间的通信。一些Web框架，如Django,TurboGears,web2py,Zope等，可以让程序员轻松地开发和管理复杂的Web程序。

- **操作系统管理、服务器运维的自动化脚本**

在很多操作系统里，Python是标准的系统组件。大多数Linux发行版以及NetBSD、OpenBSD和Mac OS X都集成了Python，可以在终端下直接运行Python。有一些Linux发行版的安装器使用Python语言编写，比如Ubuntu的Ubiquity安装器,Red Hat Linux和Fedora的Anaconda安装器。Gentoo Linux使用Python来编写它的Portage包管理系统。Python标准库包含了多个调用操作系统功能的库。通过pywin32这个第三方软件包，Python能够访问Windows的COM服务及其它Windows API。使用IronPython，Python程序能够直接调用.NET Framework。一般说来，Python编写的系统管理脚本在可读性、性能、代码重用度、扩展性几方面都优于普通的shell脚本。

- **科学计算**

NumPy,SciPy,Matplotlib可以让Python程序员编写科学计算程序。

- **桌面软件**

PyQt、PySide、wxPython、PyGTK是Python快速开发桌面应用程序的利器。

- **服务器软件（网络软件）**

Python对于各种网络协议的支持很完善，因此经常被用于编写服务器软件、网络爬虫。第三方库Twisted支持异步网络编程和多数标准的网络协议(包含客户端和服务器)，并且提供了多种工具，被广泛用于编写高性能的服务器软件。

- **游戏**

很多游戏使用C++编写图形显示等高性能模块，而使用Python或者Lua编写游戏的逻辑、服务器。相较于Python，Lua的功能更简单、体积更小；而Python则支持更多的特性和数据类型。

- 构思实现，产品早期原型和迭代

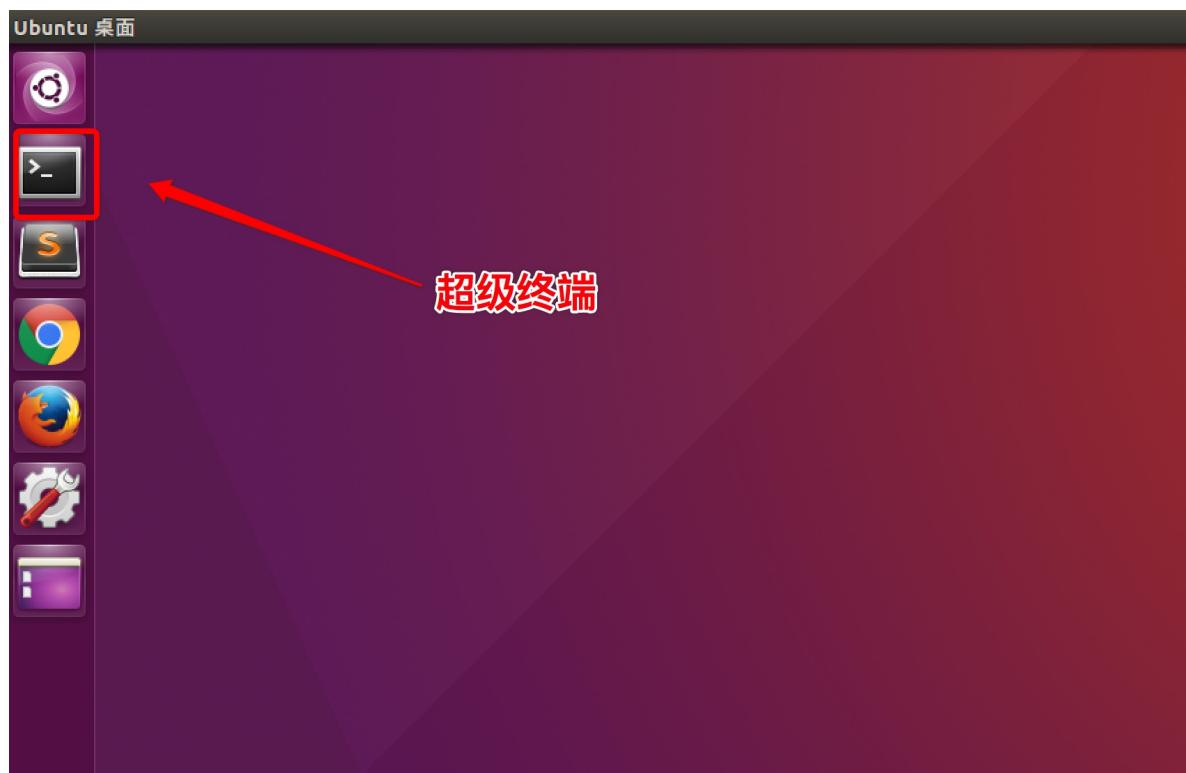
YouTube、Google、Yahoo!、NASA都在内部大量地使用Python。

```
1, python test.py运行程序  
2, python 打开python的交互模式是为了自己不懂的地方进行测试  
3, python python3 运行python 2.0或者3.0  
4, i python 不仅仅可以输入python命令也是可以进行linux命令的  
5, 退出 exit 或者exit() 自己试试就知道了  
6, 综上所述请使用 i python3 测试
```

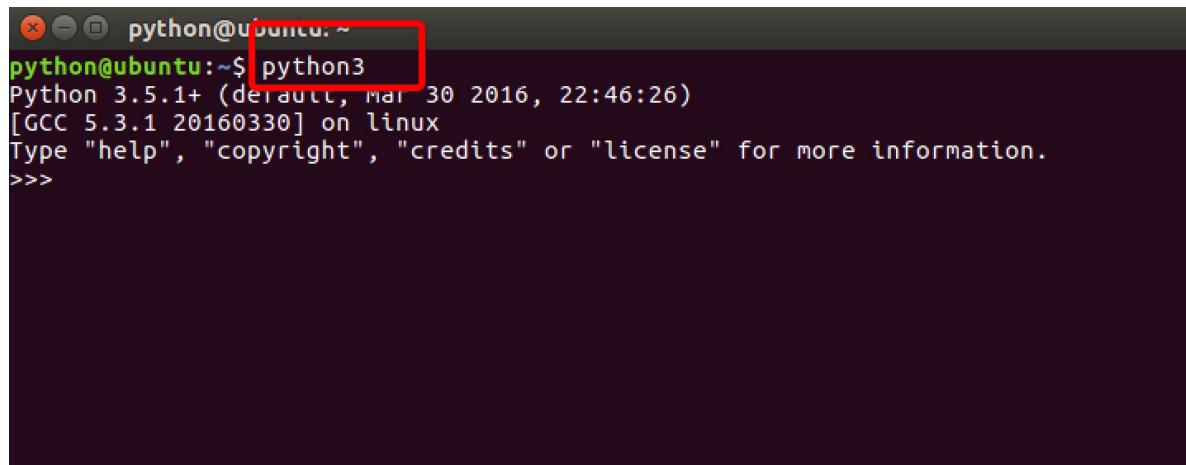
# 编写第一个python程序

## <1>编写python程序方法1

### 1. 打开“超级终端”



### 2. 输入 `python3` , 输入python3表示用的python这门编程语言的第3个版本, 如果只输入python的话表示用的是python的第2个版本



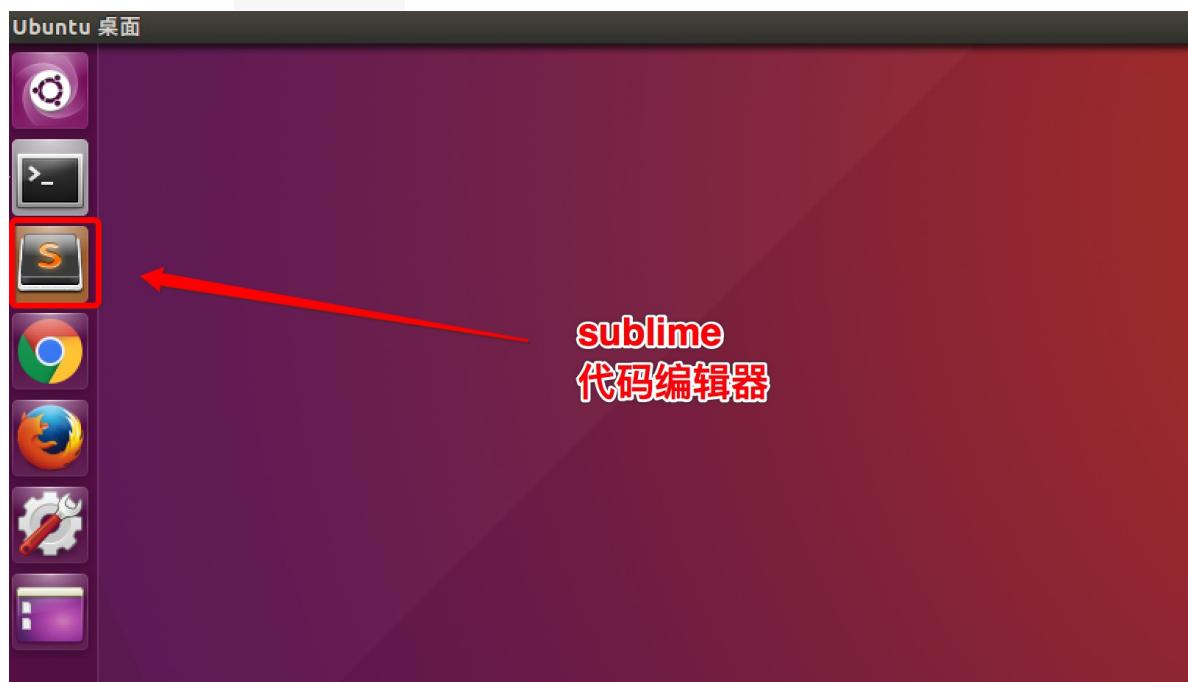
### 3. 输入以下代码

```
print('hello world')
```

```
python@ubuntu:~$ python3
Python 3.5.1+ (default, Mar 30 2016, 22:46:26)
[GCC 5.3.1 20160330] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('hello world')
hello world
>>> 
```

## <2>编写python程序方法2

- 打开编辑软件 sublime



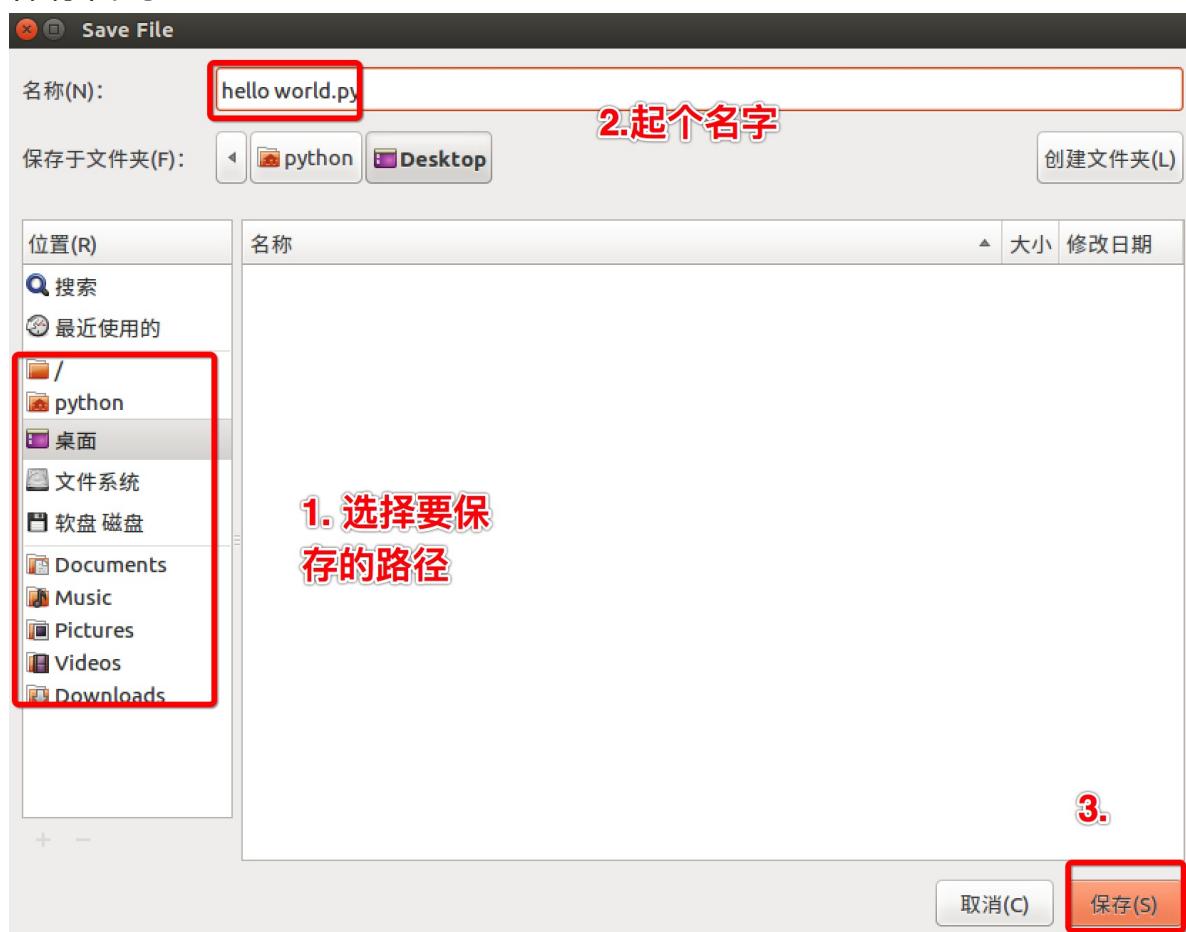
- 把以下代码，编写如下代码

```
print('hello world')
```

1 print('hello world')

在这里编写  
代码

- 保存代码



- 运行程序

The screenshot shows a terminal window with the following text:

```
python@ubuntu:~/Desktop$ ls
hello world.py
python@ubuntu:~/Desktop$
```

A red box highlights the file name "hello world.py". To the right of the terminal, the text "找到刚刚编写" (Found the just written) and "的代码文件" (code file) is displayed in red.

At the bottom of the terminal, the command "python3 hello\ world.py" is shown being typed, with a red underline under it. Below the terminal, the text "运行结果" (Execution result) is displayed in red, with two red arrows pointing from the text to the command line and the output line respectively.

```
python@ubuntu:~/Desktop$ ls
hello world.py
python@ubuntu:~/Desktop$ python3 hello\ world.py
hello world
python@ubuntu:~/Desktop$
```

运行结果

运行 xxx.py

### <3>另外一种运行python的程序的方法

- 在代码第一行写入执行时的python解释器路径，编辑完后需要对此python文件添加'x'权限

```
1 #!/usr/bin/python
2 #coding=utf-8
3 print("helloworld哈哈d")
```

The screenshot shows a terminal window with the following text:

```
python@ubuntu:~/Desktop$ ./hello.py
helloworld哈哈d
python@ubuntu:~/Desktop$
```

### <4>练一练

要求：编写一个程序，输出 `itcast.cn`

## <5>小总结

- 对于编写python程序，上面有3种方法，那到实际开发中哪一种用的比较多呢？一般是用第2或者第3种，即保存在xxx.py文件中，这样可以直接下一次执行运行；而如果用第一种方法的话，每一次运行程序都需要重新进行输入，费时费力

# 1. 注释的引入

<1> 看以下程序示例（未使用注释）

```
2048.py *  
1 #-*- coding:utf-8 -*-  
2  
3 import curses  
4 from random import randrange, choice # generate and place new tile  
5 from collections import defaultdict  
6  
7 letter_codes = [ord(ch) for ch in 'WASDRQwasdrq']  
8 actions = ['Up', 'Left', 'Down', 'Right', 'Restart', 'Exit']  
9 actions_dict = dict(zip(letter_codes, actions * 2))  
10  
11 def get_user_action(keyboard):  
12     char = "N"  
13     while char not in actions_dict:  
14         char = keyboard.getch()  
15     return actions_dict[char]  
16  
17 def transpose(field):  
18     return [list(row) for row in zip(*field)]  
19  
20 def invert(field):  
21     return [row[::-1] for row in field]  
22  
23 class GameField(object):  
24     def __init__(self, height=4, width=4, win=2048):  
25         self.height = height  
26         self.width = width  
27         self.win_value = 2048  
28         self.score = 0  
29         self.highscore = 0  
30         self.reset()  
31  
32     def reset(self):
```

<2> 看以下程序示例（使用注释）

```
2048.py
158 def main(stdscr):
159     def init():
160         #重置游戏棋盘
161         game_field.reset()
162         return 'Game'
163
164     def not_game(state):
165         #画出 GameOver 或者 Win 的界面
166         game_field.draw(stdscr)
167         #读取用户输入得到action, 判断是重启游戏还是结束游戏
168         action = get_user_action(stdscr)
169         responses = defaultdict(lambda: state) #默认是当前状态, 没有行为就会一直在当前界面循环
170         responses['Restart'], responses['Exit'] = 'Init', 'Exit' #对应不同的行为转换到不同的状态
171         return responses[action]
172
173     def game():
174         #画出当前棋盘状态
175         game_field.draw(stdscr)
176         #读取用户输入得到action
177         action = get_user_action(stdscr)
178
179         if action == 'Restart':
180             return 'Init'
181         if action == 'Exit':
182             return 'Exit'
183         if game_field.move(action): # move successful
184             if game_field.is_win():
185                 return 'Win'
186             if game_field.is_gameover():
187                 return 'Gameover'
188         return 'Game'
```

## <3> 小总结（注释的作用）

- 通过用自己熟悉的语言，在程序中对某些代码进行标注说明，这就是注释的作用，能够大大增强程序的可读性

## 2. 注释的分类

### <1> 单行注释

以#开头，#右边的所有东西当做说明，而不是真正要执行的程序，起辅助说明作用

```
# 我是注释，可以在里写一些功能说明之类的哦
print('hello world')
```

## <2> 多行注释

'''我是多行注释，可以写很多很多行的功能说明

这就是我牛x指出

哈哈哈哈。 . .

1

1

下面的代码完成：打印一首诗

名字叫做：春江花月夜

作者：忘了

1

### 3. python程序中，中文支持

如果直接在程序中用到了中文，比如

```
print('你好')
```

第一行#coding=utf-8 指定所有的编码格式是utf-8  
推荐的方式# -\*- coding:utf-8 -\*-

如果直接运行输出，程序会出错：

```
MacBook-Pro 01-python基础班-资料 $ python 01-第2天测试代码.py
File "01-第2天测试代码.py", line 3
SyntaxError: Non-ASCII character '\xe4' in file 01-第2天测试代码.py on line 3, but no encoding declared; see http://python.org/dev/peps/pep-0263/
for details
```

解决的办法为：在程序的开头写入如下代码，这就是中文注释

```
#coding=utf-8
```

修改之后的程序：

```
#coding=utf-8
print('你好')
```

运行结果：

```
你好
```

注意：

在python的语法规范中推荐使用的方式：

```
# -*- coding: utf-8 -*-
```

# 变量以及类型

## <1>变量的定义

在程序中，有时我们需要对2个数据进行求和，那么该怎样做呢？

大家类比一下现实生活中，比如去超市买东西，往往咱们需要一个菜篮子，用来进行存储物品，等到所有的物品都购买完成后，在收银台进行结账即可

如果在程序中，需要把2个数据，或者多个数据进行求和的话，那么就需要把这些数据先存储起来，然后把它们累加起来即可

在Python中，存储一个数据，需要一个叫做 变量 的东西，如下示例：

```
num1 = 100 #num1就是一个变量，就好一个小菜篮子  
num2 = 87 #num2也是一个变量  
  
result = num1 + num2 #把num1和num2这两个"菜篮子"中的数据进行累加,
```

- 说明：

- 所谓变量，可以理解为 菜篮子，如果需要存储多个数据，最简单的方式是有多个变量，当然了也可以使用一个
- 程序就是用来处理数据的，而变量就是用来存储数据的

想一想：我们应该让变量占用多大的空间，保存什么样的数据？

## <2>变量的类型

- 生活中的“类型”的例子：



- 程序中:

为了更充分的利用内存空间以及更有效率的管理内存，变量是有不同的类型的，如下所示：



- 怎样知道一个变量的类型呢?

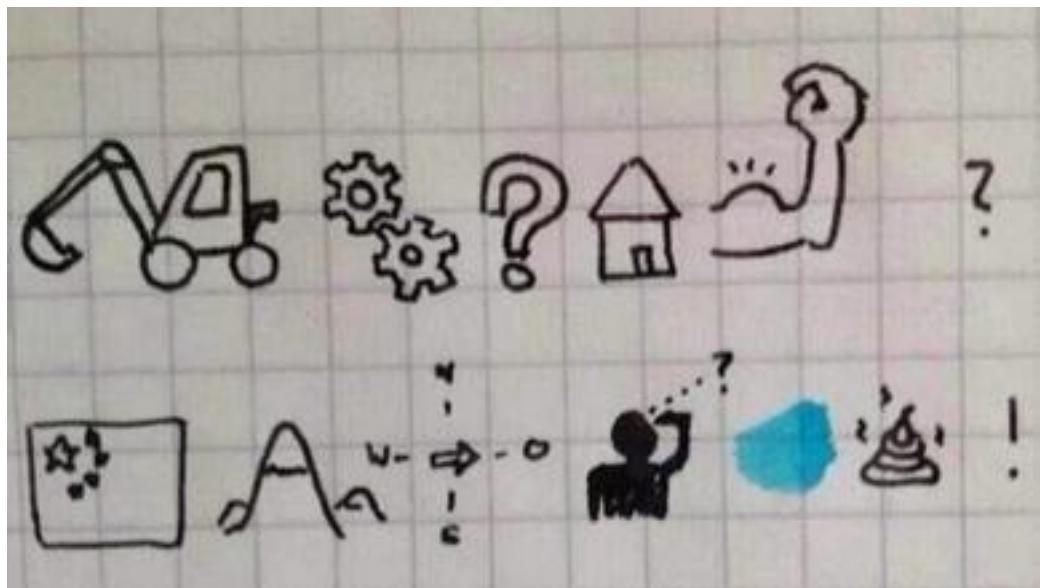
- 在python中，只要定义了一个变量，而且它有数据，那么它的类型就已经确定了，不需要咱们开发者主动的去说明它的类型，系统会自动辨别
- 可以使用`type(变量的名字)`，来查看变量的类型

[参见数据类型转换](#)

## 标示符和关键字

### <1>标示符

- 什么是标示符，看下图：



开发人员在程序中自定义的一些符号和名称

标示符是自己定义的,如变量名、函数名等

### <2>标示符的规则

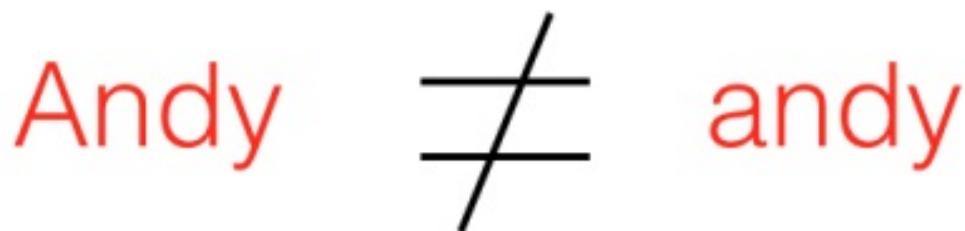
- 标示符由字母、下划线和数字组成，且数字不能开头

思考：下面的标示符哪些是正确的，哪些不正确为什么

```
fromNo12  
from#12  
my_Boolean  
my-Boolean  
Obj2  
2ndObj  
myInt
```

```
test1  
Mike2jack  
My_tExt  
_test  
test!32  
haha(da)tt  
int  
jack_rose  
jack&rose  
GUI  
G.U.I
```

- python中的标识符是区分大小写的

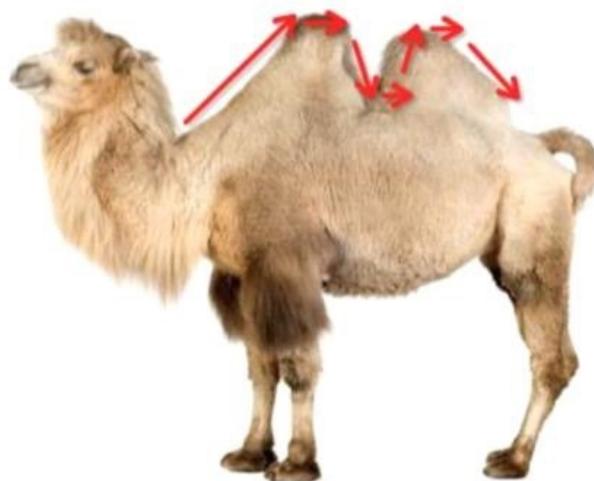


### <3>命名规则

- 见名知意

起一个有意义的名字，尽量做到看一眼就知道是什么意思(提高代码可读性) 比如：名字就定义为 name，定义学生用 student

- 驼峰命名法



如： userName      userLoginFlag

小驼峰式命名法 (lower camel case) : 第一个单词以小写字母开始; 第二个单词的首字母大写, 例如: myName、aDog

大驼峰式命名法 (upper camel case) : 每一个单字的首字母都采用大写字母, 例如: FirstName、LastName

不过在程序员中还有一种命名法比较流行, 就是用下划线“\_”来连接所有的单词, 比如send\_buf

## <4>关键字

- 什么是关键字

python一些具有特殊功能的标示符, 这就是所谓的关键字

关键字, 是python已经使用的了, 所以不允许开发者自己定义和关键字相同的名字的标示符

- 查看关键字:

and	as	assert	break	class	contin
elif	else	except	exec	finally	for
if	in	import	is	lambda	not
print	raise	return	try	while	with

可以通过以下命令进行查看当前系统中python的关键字

```
root@ubuntu:~/ftp/share# python3
Python 3.5.2 (default, Jul  5 2016, 12:43:10)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.    查看关键字
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except',
 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise',
 'return', 'try', 'while', 'with', 'yield']
>>> 
```

关键字的学习以及使用，咱们会在后面的课程中依依进行学习

import keyword 导包  
keyword.kwlist 用其中的list工作

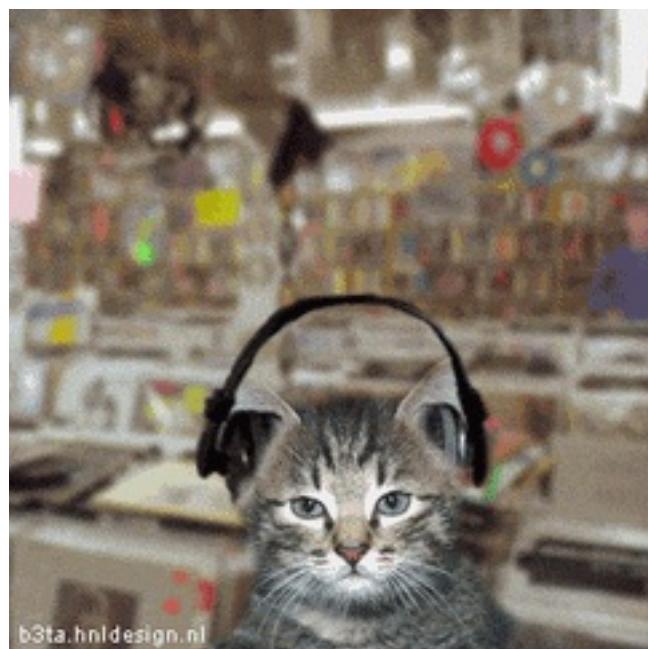
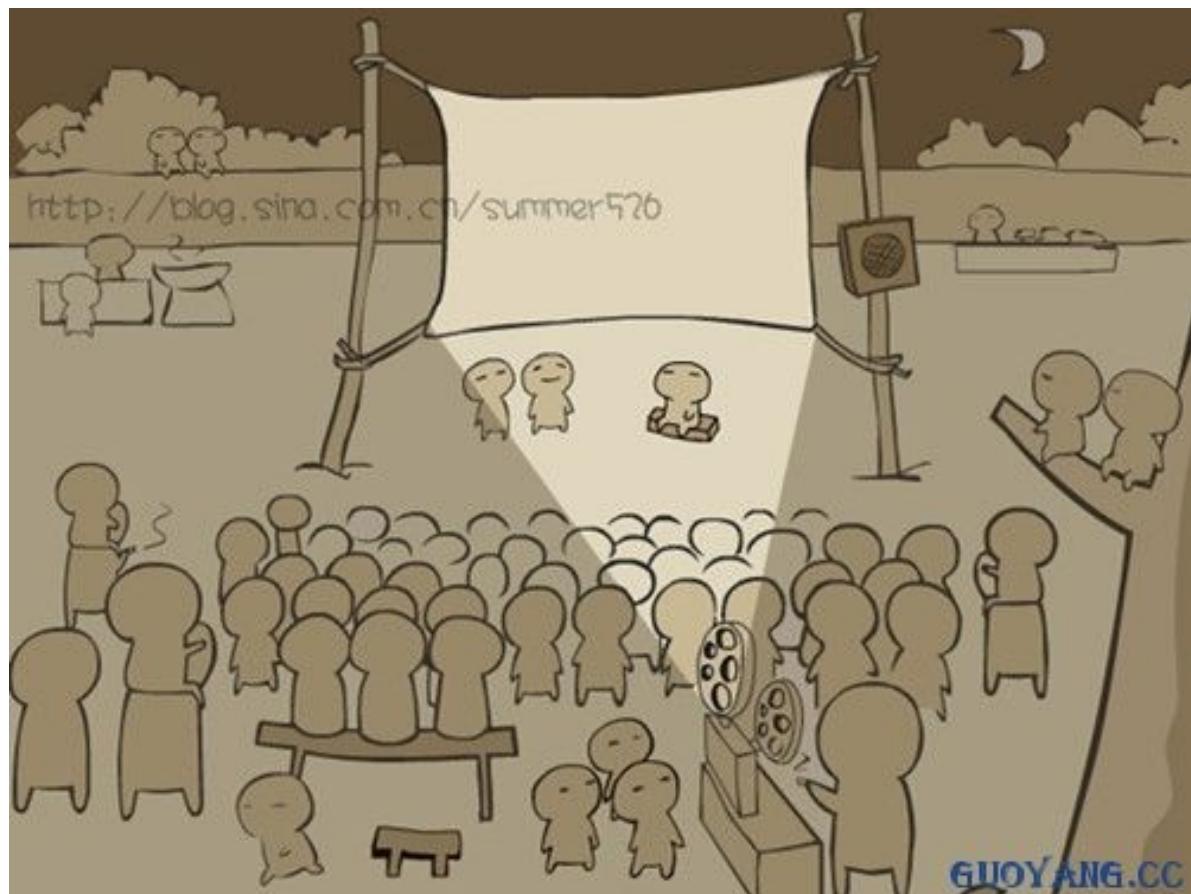
# 输出

```
print("嘻嘻嘻嘻嘻")
print("name=%s, age=%d"%(name, age)) 一一对应
```

## 1. 普通的输出

- 生活中的“输出”





- 软件中的“输出”



- python中变量的输出

```
# 打印提示
print('hello world')
print('给我的卡---印度语，你好的意思')
```

## 2. 格式化输出

### <1>格式化操作的目的

比如有以下代码:

```
pirnt("我今年10岁")
pirnt("我今年11岁")
pirnt("我今年12岁")
...
```

- 想一想:

在输出年龄的时候，用了多次“我今年xx岁”，能否简化一下程序呢？？？

- 答:

字符串格式化

## <2>什么是格式化

看如下代码:

```
age = 10
print("我今年%d岁"%age)

age += 1
print("我今年%d岁"%age)

age += 1
print("我今年%d岁"%age)

...

```

在程序中，看到了 % 这样的操作符，这就是Python中格式化输出。

```
age = 18
name = "xiaohua"
print("我的姓名是%s, 年龄是%d"%(name, age))
```

## <3>常用的格式符号

下面是完整的，它可以与%符号使用列表:

格式符号	转换
%c	字符
%s	通过str() 字符串转换来格式化
%i	有符号十进制整数

%d	有符号十进制整数
%u	无符号十进制整数
%o	八进制整数
%x	十六进制整数 (小写字母)
%X	十六进制整数 (大写字母)
%e	索引符号 (小写'e')
%E	索引符号 (大写"E")
%f	浮点实数
%g	%f和%e 的简写
%G	%f和%E的简写

### 3. 换行输出

在输出的时候，如果有 \n 那么，此时 \n 后的内容会在另外一行显示

```
print("1234567890-----") # 会在一行显示

print("1234567890\n-----") # 一行显示1234567890, 另外一行显示
```

### 4. 练一练

- 编写代码完成以下名片的显示

```
=====
姓名: dongGe
QQ:xxxxxxx
手机号:131xxxxxx
公司地址:北京市xxxx
```

```
=====
```

python2中 `raw_input("1+4")` 输出的是1+4和python3中的`input("1+4")`是一样的, 但是在python2中`input("1+4")`输出的是5, 也就是说2中input把你输入的当做代码使用, 3中是当做一块来进行的使用(就是说你输入什么人家就输出什么内容, 也就是string类型啦)

# 输入

## 1. python2版本中



咱们在银行ATM机器前取钱时, 肯定需要输入密码, 对不?

那么怎样才能让程序知道咱们刚刚输入的是什么呢? ?

大家应该知道了, 如果要完成ATM机取钱这件事情, 需要先从键盘中输入一个数据, 然后用一个变量来保存, 是不是很好理解啊

### 1.1 `raw_input()`

在Python中, 获取键盘输入的数据的方法是采用 `raw_input` 函数 (至于什么是函数, 咱们以后的章节中讲解), 那么这个 `raw_input` 怎么用呢?

看如下示例:

```
password = raw_input("请输入密码:")
print '您刚刚输入的密码是:', password
```

运行结果:

注意:

- raw\_input()的小括号中放入的是，提示信息，用来在获取数据之前给用户的一个简单提示
- raw\_input()在从键盘获取了数据以后，会存放到等号右边的变量中
- raw\_input()会把用户输入的任何值都作为字符串来对待

## 1.2 input()

input()函数与raw\_input()类似，但其接受的输入必须是表达式。

```
>>> a = input()
123
>>> a
123
>>> type(a)
<type 'int'>
>>> a = input()
abc
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<string>", line 1, in <module>
NameError: name 'abc' is not defined
>>> a = input()
"abc"
>>> a
'abc'
```

```
>>> type(a)
<type 'str'>
>>> a = input()
1+3
>>> a
4
>>> a = input()
"abc"+"def"
>>> a
'abcdef'
>>> value = 100
>>> a = input()
value
>>> a
100
```

input()接受表达式输入，并把表达式的结果赋值给等号左边的变量

## 2. python3版本中

没有raw\_input()函数，只有input()

并且 python3中的input与python2中的raw\_input()功能一样

# 运算符

python支持以下几种运算符

- 算术运算符

下面以 $a=10$ , $b=20$ 为例进行计算

运算符	描述	实例
+	加	两个对象相加 $a + b$ 输出结果 30
-	减	得到负数或是一个数减去另一个数 $a - b$ 输出结果 -10
*	乘	两个数相乘或是返回一个被重复若干次的字符串 $a * b$ 输出结果 200 " <i>ni hao</i> "*10   输出10个连在一起的 <i>ni hao</i> 字符串
/	除	$x$ 除以 $y$ $b / a$ 输出结果 2
//	取整除	返回商的整数部分 $9//2$ 输出结果 4, $9.0//2.0$ 输出结果 4.0
%	取余	返回除法的余数 $b \% a$ 输出结果 0
**	幂	返回 $x$ 的 $y$ 次幂 $a**b$ 为10的20次方,   输出结果 10000000000000000000000000

```
>>> 9/2.0
4.5
>>> 9//2.0
4.0
```

- 赋值运算符

运算符	描述	实例

=

赋值运算符

把=号右边的结果给左边的变量 num=1+2\*3 结果 num的值为7

```
>>> a, b = 1, 2
>>> a
1
>>> b
2
```

- 复合赋值运算符

运算符	描述	实例
<code>+=</code>	加法赋值运算符	<code>c += a</code> 等效于 <code>c = c + a</code>
<code>-=</code>	减法赋值运算符	<code>c -= a</code> 等效于 <code>c = c - a</code>
<code>*=</code>	乘法赋值运算符	<code>c *= a</code> 等效于 <code>c = c * a</code>
<code>/=</code>	除法赋值运算符	<code>c /= a</code> 等效于 <code>c = c / a</code>
<code>%=</code>	取模赋值运算符	<code>c %= a</code> 等效于 <code>c = c % a</code>
<code>**=</code>	幂赋值运算符	<code>c **= a</code> 等效于 <code>c = c ** a</code>
<code>//=</code>	取整除赋值运算符	<code>c //= a</code> 等效于 <code>c = c // a</code>

a\*8-5 其实表示的是a\*(8-5), 建议加上括号

# 常用的数据类型转换

函数	说明
int(x [,base ])	将x转换为一个整数
long(x [,base ])	将x转换为一个长整数
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
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

## 举例

```
a = '100' # 此时a的类型是一个字符串, 里面存放了100这3个字符
b = int(a) # 此时b的类型是整型, 里面存放的是数字100

print("a=%d"%b)
```



## 判断语句介绍

### <1>生活中的判断场景

火车站安检

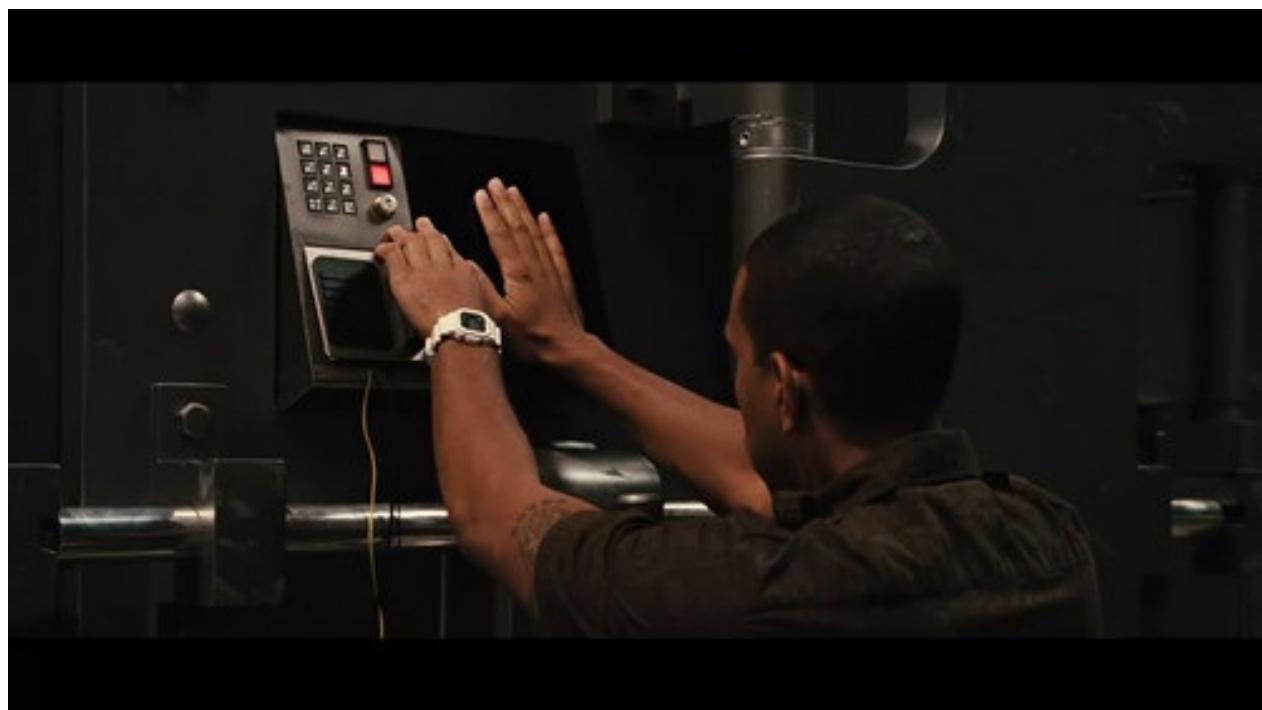


网吧



## <2>开发中的判断场景

密码判断



重要日期判断

```
if 今天是周六或者周日：  
    约妹子
```

```
if 今天是情人节：  
    买玫瑰
```

```
if 今天发工资：
```

```
    先还信用卡的钱
```

```
if 有剩余：
```

```
    又可以happy了， 0(∩_∩)0哈哈~
```

```
else：
```

```
    噢， no。。。还的等30天
```

## 小总结：

- 如果某些条件满足，才能做某件事情，而不满足时不允许做，这就是所谓的判断
- 不仅生活中有，在软件开发中“判断”功能也经常会用到

# if判断语句

## <1>if判断语句介绍

- if语句是用来进行判断的，其使用格式如下：

```
if 要判断的条件:  
    条件成立时，要做的事情
```

- demo1:

```
age = 30  
  
print "-----if判断开始-----"  
  
if age>=18:  
    print "我已经成年了"  
  
print "-----if判断结束-----"
```

- 运行结果：

```
-----if判断开始-----  
我已经成年了  
-----if判断结束-----
```

- demo2:

```
age = 16  
  
print "-----if判断开始-----"  
  
if age>=18:
```

```
print "我已经成年了"  
  
print "-----if判断结束-----"
```

- 运行结果:

```
-----if判断开始-----  
-----if判断结束-----
```

小总结:

- 以上2个demo仅仅是age变量的值不一样，结果却不同；能够看得出if判断语句的作用：就是当满足一定条件时才会执行那块代码，否则就不执行那块代码

注意：

- 代码的缩进为一个tab键，或者4个空格

## <2>练一练

要求：从键盘获取自己的年龄，判断是否大于或者等于18岁，如果满足就输出“哥，已成年，网吧可以去了”

1. 使用input从键盘中获取数据，并且存入到一个变量中
2. 使用if语句，来判断  $age \geq 18$  是否成立

## <3>想一想

- 判断age大于或者等于18岁，使用的是  $\geq$ ，还有哪些呢？

## <1> 比较(即关系)运算符

python中的比较运算符如下表

运算符	描述	示例
<code>==</code>	检查两个操作数的值是否相等，如果是则条件变为真。	如 <code>a=3,b=3</code> 则 <code>(a == b)</code> 为 <code>true</code> .
<code>!=</code>	检查两个操作数的值是否相等，如果值不相等，则条件变为真。	如 <code>a=1,b=3</code> 则 <code>(a != b)</code> 为 <code>true</code> .
<code>&lt;&gt;</code>	检查两个操作数的值是否相等，如果值不相等，则条件变为真。	如 <code>a=1,b=3</code> 则 <code>(a &lt;&gt; b)</code> 为 <code>true</code> 。这个类似于 <code>!=</code> 运算符
<code>&gt;</code>	检查左操作数的值是否大于右操作数的值，如果是，则条件成立。	如 <code>a=7,b=3</code> 则 <code>(a &gt; b)</code> 为 <code>true</code> .
<code>&lt;</code>	检查左操作数的值是否小于右操作数的值，如果是，则条件成立。	如 <code>a=7,b=3</code> 则 <code>(a &lt; b)</code> 为 <code>false</code> .
<code>&gt;=</code>	检查左操作数的值是否大于或等于右操作数的值，如果是，则条件成立。	如 <code>a=3,b=3</code> 则 <code>(a &gt;= b)</code> 为 <code>true</code> .
<code>&lt;=</code>	检查左操作数的值是否小于或等于右操作数的值，如果是，则条件成立。	如 <code>a=3,b=3</code> 则 <code>(a &lt;= b)</code> 为 <code>true</code> .

## <2> 逻辑运算符

运算符	逻辑表达式	描述	实例
	X		

and	and y	回 False, 否则它返回 y 的计算值。	返回 20。
or	x or y	布尔"或" - 如果 x 是 True, 它返回 True, 否则它返回 y 的计算值。	(a or b) 返回 10。
not	not x	布尔"非" - 如果 x 为 True, 返回 False 。如果 x 为 False, 它返回 True。	not(a and b) 返回 False

并且

或者

取反

# 作业

## 必做题：

1. 说出变量名字，可以由哪些字符组成

2. 写出变量命名时的规则

3. 说出什么是驼峰法（大驼峰、小驼峰）

4. 编写程序，完成以下要求：

- 提示用户进行输入数据
- 获取用户的数据数据（需要获取2个）
- 对获取的两个数字进行求和运行，并输出相应结果

5. 编写程序，完成以下要求：

- 提示用户进行输入数据
- 获取用户的数据数据（需要获取2个）
- 对获取的两个数字进行减法运行，并输出相应结果

6. 编写程序，完成以下信息的显示：

```
=====
=          欢迎进入到身份认证系统V1.0
= 1. 登录
= 2. 退出
= 3. 认证
= 4. 修改密码
=====
```

## 7. 编写程序，从键盘获取一个人的信息，然后按照下面格式显示

```
=====
姓名: dongGe
QQ:xxxxxxx
手机号:131xxxxxx
公司地址:北京市xxxx
=====
```

## 8. 编写程序，从键盘获取用户名和密码，然后判断，如果正确就输出以下信息

```
亲爱的xxx, 欢迎登陆 爱学习管理系统
```

# 附录-推荐的python电子书

python学习路线图

优先级

入门:python核心编程

提高:python cookbook

其他

The screenshot shows a file sharing interface with the following details:

Folder name: 给学生推荐python书籍

Share time: 2016-06-13 19:05

Buttons: 保存至网盘, 下载, 举报, 分享至

Links: 返回上一级 | 全部文件 > 给学生推荐python书...

文件名	大小	修改日期
Python核心编程(第二版).pdf	141.9M	2016-06-13 19:04
python标准库.pdf	651KB	2016-06-13 19:04
python340.chm	6.9M	2016-06-13 19:04
Python.Cookbook(第2版).pdf	68.7M	2016-06-13 19:04
PYTHON WEB开发学习实录.pdf	62.1M	2016-06-13 19:04

电子书下载地址:

链接: <http://pan.baidu.com/s/1dFI4p7V> 密码: 7es5



程序三种顺序执行: 顺序, 选择, 循环

## if-else

想一想: 在使用if的时候, 它只能做到满足条件时要做的事情。那万一需要在不满足条件的时候, 做某些事, 该怎么办呢?

答: else

### <1>if-else的使用格式

`if` 条件:

    满足条件时要做的事情1

    满足条件时要做的事情2

    满足条件时要做的事情3

    ... (省略) ...

`else`:

    不满足条件时要做的事情1

    不满足条件时要做的事情2

    不满足条件时要做的事情3

    ... (省略) ...

tab键位类似于  
java中的{}  


demo1

前边已经有定义了, `ctrl + n` 提示, 类似Tab

```
chePiao = 1 # 用1代表有车票, 0代表没有车票
```

```
if chePiao == 1:
    print("有车票, 可以上火车")
    print("终于可以见到Ta了, 美滋滋~~~")
else:
    print("没有车票, 不能上车")
    print("亲爱的, 那就下次见了, 一票难求啊~~~~(>_<)~~~~")
```

注意: 条件后边受tab键位的限制, 没有的话那么就不受限制, 但是在不受限制之后又是tab的想从上边的条件进行是不行的会报错.  
`python3 1.py +11` 调到报错行11行

结果1: 有车票的情况

有车票, 可以上火车

终于可以见到Ta了，美滋滋~~~

结果2：没有车票的情况

没有车票，不能上课

亲爱的，那就下次见了，一票难求啊~~~~(>\_<)~~~~

## <2>练一练

要求：从键盘输入刀子的长度，如果刀子长度没有超过10cm，则允许上火车，否则不允许上火车

## elif

- 想一想:

if能完成当xxx时做事情

if-else能完成当xxx时做事情1, 否则做事情2

如果有这样一种情况: 当xxx1时做事情1, 当xxx2时做事情2, 当xxx3时做事情3, 那该怎么实现呢?

- 答:

elif

### <1> elif的功能

elif的使用格式如下:



说明:

- 当xxx1满足时, 执行事情1, 然后整个if结束
- 当xxx1不满足时, 那么判断xxx2, 如果xxx2满足, 则执行事情2, 然后整个if结束
- 当xxx1不满足时, xxx2也不满足, 如果xxx3满足, 则执行事情3, 然后整个if结束

demo:

```
score = 77

if score>=90 and score<=100:
    print('本次考试, 等级为A')
elif score>=80 and score<90:
    print('本次考试, 等级为B')
elif score>=70 and score<80:
    print('本次考试, 等级为C')
elif score>=60 and score<70:
    print('本次考试, 等级为D')
elif score>=0 and score<60:
    print('本次考试, 等级为E')
```

## <2> 注意点

- 可以和else一起使用

```
if 性别为男性:
    输出男性的特征
    ...
elif 性别为女性:
    输出女性的特征
    ...
else:
    第三种性别的特征
    ...
```

说明:

- 当“性别为男性”满足时，执行“输出男性的特征”的相关代码
- 当“性别为男性”不满足时，如果“性别为女性”满足，则执行“输出女

性的特征”的相关代码

- 当“性别为男性”不满足，“性别为女性”也不满足，那么久默认执行  
else后面的代码，即“第三种性别的特征”相关代码
- elif必须和if一起使用，否则出错

# if嵌套

通过学习if的基本用法，已经知道了

- 当需要满足条件去做事情的这种情况需要使用if
- 当满足条件时做事情A，不满足条件做事情B的这种情况使用if-else

想一想：

坐火车或者地铁的实际情况是：先进行安检如果安检通过才会判断是否有车票，或者是先检查是否有车票之后才会进行安检，即实际的情况某个判断是再另外一个判断成立的基础上进行的，这样的情况该怎样解决呢？

答：

if嵌套

## <1>if嵌套的格式

if 条件1:

    满足条件1 做的事情1

    满足条件1 做的事情2

    ... (省略) ...

if 条件2:

    满足条件2 做的事情1

    满足条件2 做的事情2

    ... (省略) ...

- 说明
  - 外层的if判断，也可以是if-else
  - 内层的if判断，也可以是if-else

- 根据实际开发的情况，进行选择

## <2>if嵌套的应用

demo:

```
chePiao = 1      # 用1代表有车票，0代表没有车票
daoLenght = 9    # 刀子的长度，单位为cm

if chePiao == 1:
    print("有车票，可以进站")
    if daoLenght < 10:
        print("通过安检")
        print("终于可以见到Ta了，美滋滋~~~")
    else:
        print("没有通过安检")
        print("刀子的长度超过规定，等待警察处理...")

else:
    print("没有车票，不能进站")
    print("亲爱的，那就下次见了，一票难求啊~~~~(>_<)~~~~")
```

结果1: chePiao = 1;daoLenght = 9

```
有车票，可以进站
通过安检
终于可以见到Ta了，美滋滋~~~
```

结果2: chePiao = 1;daoLenght = 20

```
有车票，可以进站
没有通过安检
刀子的长度超过规定，等待警察处理...
```

结果3: chePiao = 0;daoLenght = 9

没有车票，不能进站

亲爱的，那就下次见了，一票难求啊~~~~(>\_<)~~~~

结果4：chePiao = 0;daoLenght = 20

没有车票，不能进站

亲爱的，那就下次见了，一票难求啊~~~~(>\_<)~~~~

想一想：为什么结果3和结果4相同？？？

### <3>练一练

情节描述：上公交车，并且可以有座位坐下

要求：输入公交卡当前的余额，只要超过2元，就可以上公交车；如果空座位的数量大于0，就可以坐下

# 应用:猜拳游戏

## <1>运行效果:



```
01-python基础班 — python@ubuntu: ~/Desktop — ssh python@172.16.138.135 — 133x34
python@ubuntu:~/Desktop$ python test.py
请输入：剪刀(0) 石头(1) 布(2):
```

## <2>参考代码:

```
import random

player = input('请输入：剪刀(0) 石头(1) 布(2):')

player = int(player)

computer = random.randint(0, 2)

# 用来进行测试
#print('player=%d, computer=%d', (player, computer))

if ((player == 0) and (computer == 2)) or ((player == 1) and
```

```
    print('获胜， 哈哈， 你太厉害了')
elif player == computer:
    print('平局， 要不再来一局')
else:
    print('输了， 不要走， 洗洗手接着来， 决战到天亮')
```

# 循环介绍

## <1>生活中的循环场景

跑道



风扇



CF加特林



## <2>软件开发中循环的使用场景

跟媳妇承认错误，说一万遍“媳妇儿，我错了”

```
print("媳妇儿，我错了")
print("媳妇儿，我错了")
print("媳妇儿，我错了")
....(还有99997遍)....
```

使用循环语句一句话搞定

```
i = 0
while i<10000:
    print("媳妇儿，我错了")
    i+=1
```

## <3>小总结

- 一般情况下，需要多次重复执行的代码，都可以用循环的方式来完成
- 循环不是必须要使用的，但是为了提高代码的重复使用率，所以有经验的开发者都会采用循环

# while循环

## <1>while循环的格式

```
while 条件:  
    条件满足时, 做的事情1  
    条件满足时, 做的事情2  
    条件满足时, 做的事情3  
    ... (省略) ...
```

demo

while True : 后边的True和False 注意大写

```
i = 0  
while i<5:  
    print("当前是第%d次执行循环"%(i+1))  
    print("i=%d"%i)  
    i+=1
```

结果:

```
当前是第1次执行循环  
i=0  
当前是第2次执行循环  
i=1  
当前是第3次执行循环  
i=2  
当前是第4次执行循环  
i=3  
当前是第5次执行循环  
i=4
```



## while循环应用

### 1. 计算1~100的累积和（包含1和100）

参考代码如下：

```
#encoding=utf-8

i = 1
sum = 0
while i<=100:
    sum = sum + i
    i += 1

print("1~100的累积和为:%d"%sum)
```

### 2. 计算1~100之间偶数的累积和（包含1和100）

参考代码如下：

```
#encoding=utf-8

i = 1
sum = 0
while i<=100:
    if i%2 == 0:
        sum = sum + i
    i+=1

print("1~100的累积和为:%d"%sum)
```



## while循环嵌套

- 前面学习过if的嵌套了，想一想if嵌套是什么样子的？
- 类似if的嵌套，while嵌套就是：while里面还有while

### <1>while嵌套的格式

```
while 条件1:
```

    条件1满足时，做的事情1

    条件1满足时，做的事情2

    条件1满足时，做的事情3

    ... (省略) ...

```
while 条件2:
```

    条件2满足时，做的事情1

    条件2满足时，做的事情2

    条件2满足时，做的事情3

    ... (省略) ...

### <2>while嵌套应用一

要求：打印如下图形：

```
*  
* *  
* * *  
* * * *  
* * * * *
```

参考代码：

```
i = 1
while i<=5:

    j = 1
    while j<=i:
        print("* ", end=' ')
        j+=1

    print("\n")
    i+=1
```

## <3>while嵌套应用二：九九乘法表

```
1*1=1
1*2=2  2*2=4
1*3=3  2*3=6  3*3=9
1*4=4  2*4=8  3*4=12 4*4=16
1*5=5  2*5=10 3*5=15 4*5=20 5*5=25
1*6=6  2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7=7  2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8=8  2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
1*9=9  2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81
```

参考代码：

```
i = 1
while i<=9:
    j=1
    while j<=i:
        print("%d * %d = %-2d "%(j,i,i*j),end=' ')
        j+=1
    print('\n')
```

i+=1

\t 制表位

%-2d 表示以int类型格式输出结果，而且保留  
2位整数（若结果位数大于2位，则按实际结果输出）

两者效果一样的

```
FILE EDIT VIEW SEARCH TERMINAL HELP
# -*- coding = utf-8 -*-
i = 1
while i <= 9:
    j = 1
    while j <= i:
        print("%d * %d = %d \t" % (j, i, i*j), end = ' ')
        j += 1
    print("\n")
    i += 1
```

## for循环

像while循环一样， for可以完成循环的功能。

在Python中 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等。

## for循环的格式

```
for 临时变量 in 列表或者字符串等:  
    循环满足条件时执行的代码  
else:  
    循环不满足条件时执行的代码
```

## demo1

```
name = 'dongGe'  
  
for x in name:  
    print(x)
```

运行结果如下：

```
>>> name = 'dongGe'  
>>>  
>>> for x in name:  
...     print(x)  
...  
d  
o  
n  
g  
G  
e  
>>> █
```

## demo2

```
name = ''  
  
for x in name:  
    print(x)  
else:  
    print("没有数据")
```

运行结果如下：

```
>>> name = ''  
>>>  
>>> for x in name:  
...     print(x)  
... else:  
...     print("没有数据")  
...  
没有数据
```



# break和continue

## 1. break

### <1> for循环

- 普通的循环示例如下：

```
name = 'dongGe'

for x in name:
    print('---')
    print(x)
```

运行结果：

```
>>> name = 'dongGe'  
>>>  
>>> for x in name:  
...     print('----')  
[...     print(x)  
[...  
----  
d  
----  
o  
----  
n  
----  
g  
----  
G  
----  
e  
[>>>
```

- 带有break的循环示例如下:

```
name = 'dongGe'  
  
for x in name:  
    print('----')  
    if x == 'g':  
        break  
    print(x)
```

运行结果:

```
>>> name = 'dongGe'  
>>>  
>>> for x in name:  
...     print('----')  
...     if x == 'g':  
...         break  
[...     print(x)  
[...  
----  
d  
----  
o  
----  
n  
----  
>>> █
```

## <2> while循环

- 普通的循环示例如下：

```
i = 0  
  
while i<10:  
    i = i+1  
    print('----')  
    print(i)
```

运行结果：

```
>>> while i<10:  
...     i = i+1  
...     print('----')  
...     print(i)  
...  
----  
1  
----  
2  
----  
3  
----  
4  
----  
5  
----  
6  
----  
7  
----  
8  
----  
9  
----  
10  
>>>
```

- 带有break的循环示例如下：

```
i = 0  
  
while i<10:  
    i = i+1
```

```
print('----')
if i==5:
    break
print(i)
```

运行结果：

```
>>> i = 0
>>>
>>> while i<10:
...     i = i+1
...     print('----')
...     if i==5:
...         break
...     print(i)
...
-----
1
-----
2
-----
3
-----
4
-----
>>> █
```

## 小总结：

- break的作用：用来结束整个循环

## 2. continue

## <1> for循环

- 带有continue的循环示例如下：

```
name = 'dongGe'

for x in name:
    print('----')
    if x == 'g':
        continue
    print(x)
```

运行结果：

```
>>> name = 'dongGe'  
>>>  
>>> for x in name:  
...     print('----')  
...     if x == 'g':  
...         continue  
...     print(x)  
...  
----  
d  
----  
o  
----  
n  
----  
----  
G  
----  
e  
>>> █
```

## <2> while循环

- 带有continue的循环示例如下：

```
i = 0  
  
while i<10:  
    i = i+1  
    print('----')  
    if i==5:  
        continue  
    print(i)
```

运行结果：

```
>>> i = 0
>>>
>>> while i<10:
...     i = i+1
...     print('----')
...     if i==5:
...         continue
[...     print(i)
[...
-----
1
-----
2
-----
3
-----
4
-----
-----
6
-----
7
-----
8
-----
9
-----
10
>>> █
```

- 小总结：

- continue的作用：用来结束本次循环，紧接着执行下一次的循环

### 3. 注意点

- break/continue只能用在循环中，除此以外不能单独使用
- break/continue在嵌套循环中，只对最近的一层循环起作用

# 总结

- if往往用来对条件是否满足进行判断
- if有4中基本的使用方法：

## 1. 基本方法

```
if 条件:  
    满足时做的事情
```

## 2. 满足与否执行不同的事情

```
if 条件:  
    满足时做的事情  
else:  
    不满足时做的事情
```

## 3. 多个条件的判断

```
if 条件:  
    满足时做的事情  
elif 条件2:  
    满足条件2时做的事情  
elif 条件3:  
    满足条件3时做的事情  
else:  
    条件都不满足时做的事情
```

## 4. 嵌套

```
if 条件:  
    满足时做的事情  
  
    这里还可以放入其他任何形式的if判断语句
```

- while循环一般通过数值是否满足来确定循环的条件

```
i = 0
while i<10:
    print("hello")
    i+=1
```

- for循环一般是对能保存多个数据的变量，进行便利

```
name = 'dongGe'

for x in name:
    print(x)
```

- if、while、for等其他语句可以随意组合，这样往往就完成了复杂的功能

# 作业

## 必做题：

### 1. 使用if，编写程序，实现以下功能：

- 从键盘获取用户名、密码
- 如果用户名和密码都正确（预先设定一个用户名和密码），那么就显示“欢迎进入xxx的世界”，否则提示密码或者用户名错误

### 2. 使用while，完成以下图形的输出

```
*  
* *  
* * *  
* * * *  
* * * * *  
* * * *  
* * *  
* *  
*
```

## 选做题：

### 1. 根据以下信息提示，请帮小明计算，他每月乘坐地铁支出的总费用

#### 提示信息：

北京公交地铁新票价确定

据北京市发改委网站消息称，北京市将从2015年12月28起实施公共交通新票价：地铁6公里(含)内3元，公交车10公里(含)内2元，使用市政交通一卡通刷卡乘公交车普通卡5折，学生卡2.5折。

具体实施方案如下：

一、城市公共电汽车价格调整为：10公里(含)内2元，10公里以上部分，每增加1元可乘坐5公里。使用市政交通一卡通刷卡乘坐城市公共电汽车，市域内路段给予普通卡5折，学生卡2.5折优惠；市域外路段维持现行折扣优惠不变。享受公交政策的郊区客运价格，由各区、县政府按照城市公共电汽车价格制定。

二、轨道交通价格调整为：6公里(含)内3元；6公里至12公里(含)4元；12公里至22公里(含)5元；22公里至32公里(含)6元；32公里以上部分，每增加1元可乘坐20公里。使用市政交通一卡通刷卡乘坐轨道交通，每自然月内每张卡支出累计满100元以后的乘次，价格给予8折优惠；满150元以后的乘次，价格给予5折优惠；支出累计达到400元以后的乘次，不再享受打折优惠。

## 要求：

假设每个月，小明都需要上20天班，每次上班需要来回1次，即每天需要乘坐2次同样路线的地铁；每月月初小明第一次刷公交卡时，扣款5元；编写程序，帮小明完成每月乘坐地铁需要的总费用



cpu 计算快存储小 而 硬盘 存贮大速度慢 所以产生了中间东西 内存.

## 字符串介绍

- 想一想：

当打来浏览器登录某些网站的时候，需要输入密码，浏览器把密码传送到服务器后，服务器会对密码进行验证，其验证过程是把之前保存的密码与本次传递过去的密码进行对比，如果相等，那么就认为密码正确，否则就认为不对；服务器既然想要存储这些密码可以用数据库（比如MySQL），当然为了简单起见，咱们可以先找个变量把密码存储起来即可；那么怎样存储带有字母的密码呢？

- 答：

字符串

### <1>python中字符串的格式

''   ''' 单双引号括起来的都是字符串  
+ 方式和java一样，区分string和int等

如下定义的变量a，存储的是数字类型的值

```
a = 100
```

如下定义的变量b，存储的是字符串类型的值

```
b = "hello itcast.cn"  
或者  
b = 'hello itcast.cn'
```

小总结：

- 双引号或者单引号中的数据，就是字符串

# 字符串输出

demo

```
name = 'xiaoming'  
position = '讲师'  
address = '北京市昌平区建材城西路金燕龙办公楼1层'  
  
print('-----')  
print("姓名: %s"%name)  
print("职位: %s"%position)  
print("公司地址: %s"%address)  
print('-----')
```

结果:

```
-----  
姓名: xiaoming  
职位: 讲师  
公司地址: 北京市昌平区建材城西路金燕龙办公楼1层  
-----
```

## 字符串输入

之前在学习input的时候，通过它能够完成从键盘获取数据，然后保存到指定的变量中；

注意：input获取的数据，都以字符串的方式进行保存，即使输入的是数字，那么也是以字符串方式保存

demo:

```
userName = input('请输入用户名：')
print("用户名为：%s"%userName)

password = input('请输入密码：')
print("密码为：%s"%password)
```

结果：（根据输入的不同结果也不同）

```
请输入用户名:dongGe
用户名为: dongGe
请输入密码:haohaoxuexitiantianxiangshang
密码为: haohaoxuexitiantianxiangshang
```

# 下标和切片

## 1. 下标索引

所谓“下标”，就是编号，就好比超市中的存储柜的编号，通过这个编号就能找到相应的存储空间

- 生活中的“下标”

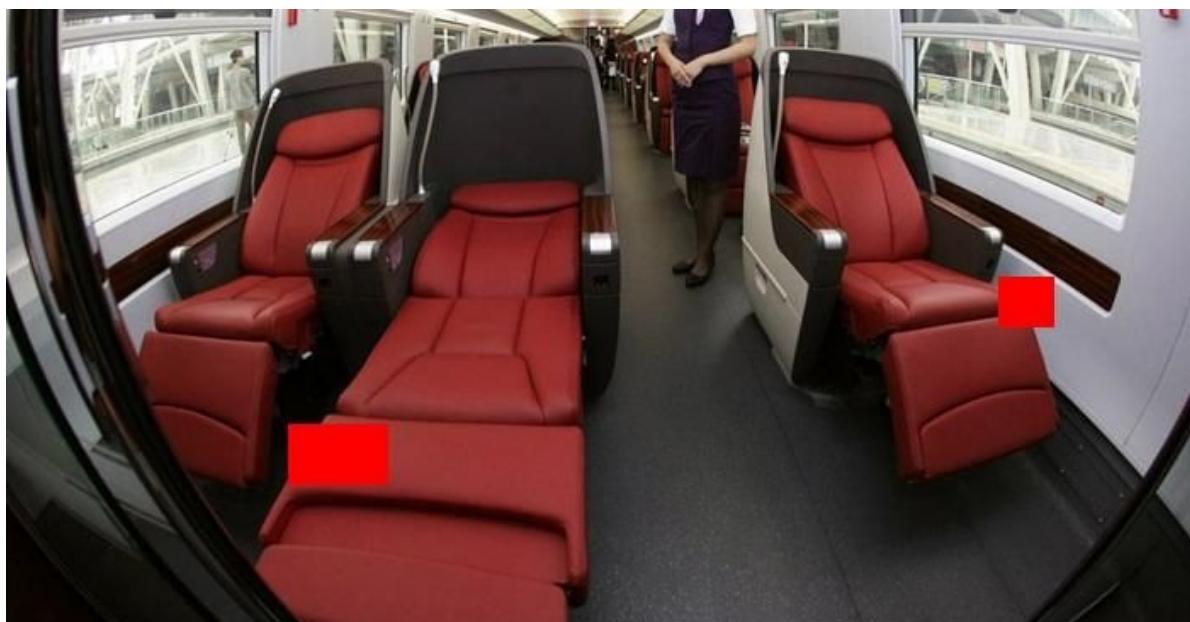
超市储物柜



高铁二等座



高铁一等座



绿皮车



- 字符串中"下标"的使用

列表与元组支持下标索引好理解，字符串实际上就是字符的数组，所以也支持下标索引。

如果有字符串: `name = 'abcdef'`，在内存中的实际存储如下:



如果想取出部分字符，那么可以通过 下标 的方法，（注意python中下标从 0 开始）

```

name = 'abcdef'

print(name[0])
print(name[1])
print(name[2])

```

运行结果:

The screenshot shows a terminal window with two panes. The left pane displays the code in `test.py`:

```

1 #coding=utf-8
2
3
4 name = 'abcdef'
5
6 print(name[0])
7 print(name[1])
8 print(name[2])
9

```

The right pane shows the terminal output:

```

python@ubuntu:~/Desktop$ python3 test.py
a
b
c
python@ubuntu:~/Desktop$ █

```

## 2. 切片

切片是指对操作的对象截取其中一部分的操作。**字符串、列表、元组**都支持切片操作。

**切片的语法: [起始:结束:步长]**

**注意: 选取的区间属于左闭右开型, 即从"起始"位开始, 到"结束"位的前一位结束 (不包含结束位本身)。**

我们以字符串为例讲解。

`name[2:]` 2到最后  
`name[2:0]` 啥都没有  
`name[2:-1:2]` 跳着取值, 步长为2  
`name[::-1]` 字符串逆序

如果取出一部分, 则可以在中括号[]中, 使用:

```

name = 'abcdef'

print(name[0:3]) # 取 下标0~2 的字符

```

## 运行结果:

```
test.py
1 #coding=utf-8
2
3
4 name = 'abcdef'
5
6 print(name[0:3]) # 取下标0~2 的字符
7
```

```
python@ubuntu:~/Desktop$ python3 test.py
abc
python@ubuntu:~/Desktop$
```

```
name = 'abcdef'

print(name[0:5]) # 取下标为0~4 的字符
```

## 运行结果:

```
test.py
1 #coding=utf-8
2
3
4
5 name = 'abcdef'
6
7 print(name[0:5]) # 取下标为0~4 的字符
8
```

```
python@ubuntu:~/Desktop$ python3 test.py
abcde
python@ubuntu:~/Desktop$
```

```
name = 'abcdef'

print(name[3:5]) # 取下标为3、4 的字符
```

## 运行结果:

```
test.py
1 #coding=utf-8
2
3 name = 'abcdef'
4
5 print(name[3:5]) # 取下标为3、4 的字符
6
```

```
python@ubuntu:~/Desktop$ python3 test.py
de
python@ubuntu:~/Desktop$
```

```
name = 'abcdef'

print(name[2:]) # 取下标为2开始到最后的字符
```

运行结果：

```
test.py
1 #coding=utf-8
2
3
4 name = 'abcdef'
5
6 print(name[2:]) # 取 下标为2开始到最后的字符
7
```

```
python@ubuntu:~/Desktop$ python3 test.py
cdef
python@ubuntu:~/Desktop$
```

```
name = 'abcdef'

print(name[1:-1]) # 取 下标为1开始 到 最后第2个 之间的字符
```

运行结果：

```
test.py
1 #coding=utf-8
2
3 name = 'abcdef'
4
5 print(name[1:-1]) # 取 下标为1开始 最后第2个之间的字符串
6
```

```
python@ubuntu:~/Desktop$ python3 test.py
bcde
python@ubuntu:~/Desktop$
```

```
>>> a = "abcdef"
>>> a[:3]
'abc'
>>> a[::-2]
'ace'
>>> a[5:1:-2]
 ''
>>> a[1:5:-2]
'bd'
>>> a[::-2]
'fdb'
>>> a[5:1:-2]
'fd'
```

## 想一想

- (面试题) 给定一个字符串aStr, 请反转字符串



# 字符串常见操作

如有字符串 `mystr = 'hello world itcast and itcastcpp'`，以下是常见的操作

## <1>find

检测 str 是否包含在 mystr 中，如果是返回开始的索引值，否则返回-1

```
mystr.find(str, start=0, end=len(mystr))
```

```
>>> mystr = 'hello world itcast and itcastcpp'  
>>> mystr.find("itcast")  
12  
>>> █
```

```
>>> mystr = 'hello world itcast and itcastcpp'  
>>> mystr.find("itcast",0,10)  
-1  
>>> █
```

## <2>index

跟find()方法一样，只不过如果str不在 mystr中会报一个异常.

```
mystr.index(str, start=0, end=len(mystr))
```

```
>>> mystr = 'hello world itcast and itcastcpp'  
>>> mystr.find("itcast",0,10)  
-1  
>>> mystr.index("itcast",0,10)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: substring not found  
>>> █
```

### <3>count

返回 str在start和end之间 在 mystr里面出现的次数

```
mystr.count(str, start=0, end=len(mystr))
```

```
>>> mystr = 'hello world itcast and itcastcpp'  
>>> mystr.count("itcast")  
2  
>>> █
```

### <4>replace

把 mystr 中的 str1 替换成 str2,如果 count 指定, 则替换不超过 count 次.

```
mystr.replace(str1, str2, mystr.count(str1))
```

```
>>> name="hello world ha ha"  
>>> name.replace("ha", "Ha")  
'hello world Ha Ha'  
>>> name.replace("ha", "Ha", 1)  
'hello world Ha ha'  
>>> █
```

### <5>split

以 str 为分隔符切片 mystr, 如果 maxsplit有指定值, 则仅分隔 maxsplit 个子字符串

```
mystr.split(str="", 2)
```

```
>>> name="hello world ha ha"  
>>> name.split(" ")  
['hello', 'world', 'ha', 'ha']  
>>> name.split(" ", 2)  
['hello', 'world', 'ha ha']  
>>> [ ]
```

## <6>capitalize

把字符串的第一个字符大写

```
mystr.capitalize()
```

```
>>> mystr = 'hello world itcast and itcastcpp'  
>>> mystr.capitalize()  
'Hello world itcast and itcastcpp'  
>>>
```

## <7>title

把字符串的每个单词首字母大写

```
>>> a = "hello itcast"  
>>> a.title()  
'Hello Itcast'
```

## <8>startswith

检查字符串是否是以 obj 开头, 是则返回 True, 否则返回 False

```
mystr.startswith(obj)
```

```
>>> mystr = 'hello world itcast and itcastcpp'  
>>> mystr.startswith("hello")  
True  
>>> mystr.startswith("Hello")  
False  
>>> █
```

## <9>endswith

检查字符串是否以obj结束，如果是返回True,否则返回 False.

```
mystr.endswith(obj)
```

```
>>> mystr = 'hello world itcast and itcastcpp'  
>>> mystr.endswith('cpp')  
True  
>>> mystr.endswith('app')  
False  
>>> █
```

## <10>lower

转换 mystr 中所有大写字符为小写

```
mystr.lower()
```

```
>>> mystr = 'HELLO world itcast and itcastcpp'  
>>> mystr.lower()  
'hello world itcast and itcastcpp'  
>>> █
```

## <11>upper

转换 mystr 中的小写字符为大写

```
mystr.upper()
```

```
>>> mystr = 'HELLO world itcast and itcastcpp'  
>>> mystr.upper()  
'HELLO WORLD ITCAST AND ITCASTCPP'  
>>> █
```

## <12>ljust

返回一个原字符串左对齐,并使用空格填充至长度 width 的新字符串

```
mystr.ljust(width)
```

```
>>> mystr="hello"  
>>> mystr.ljust(10)  
'hello      '  
>>> █
```

## <13>rjust

返回一个原字符串右对齐,并使用空格填充至长度 width 的新字符串

```
mystr.rjust(width)
```

```
>>> mystr="hello"  
>>> mystr.rjust(10)  
'      hello'  
>>> █
```

## <14>center

返回一个原字符串居中,并使用空格填充至长度 width 的新字符串

```
mystr.center(width)
```

```
>>> mystr = 'hello world itcast and itcastcpp'
>>> mystr.center(50)
'          hello world itcast and itcastcpp          '
>>>
```

### <15>lstrip

删除 mystr 左边的空白字符

```
mystr.lstrip()
```

```
>>> mystr="      hello"
>>> mystr.lstrip()
'hello'
>>> mystr="      hello      "
>>> mystr.lstrip()
'hello      '
>>>
```

### <16>rstrip

删除 mystr 字符串末尾的空白字符

```
mystr.rstrip()
```

```
>>> mystr="      hello      "
>>> mystr.rstrip()
'      hello'
>>>
```

### <17>strip

删除mystr字符串两端的空白字符

```
>>> a = "\n\t itcast \t\n"
>>> a.strip()
'itcast'
```

## <18>rfind

类似于 find()函数，不过是从右边开始查找.

```
mystr.rfind(str, start=0, end=len(mystr) )
```

```
>>> mystr = 'hello world itcast and itcastcpp'
>>> mystr.rfind("itcast")
23
>>> █
```

## <19>rindex

类似于 index(), 不过是从右边开始.

```
mystr.rindex( str, start=0, end=len(mystr))
```

```
>>> mystr = 'hello world itcast and itcastcpp'
>>> mystr.rindex("IT")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> █
```

## <20>partition

把mystr以str分割成三部分,str前, str和str后

```
mystr.partition(str)
```

```
>>> mystr = 'hello world itcast and itcastcpp'  
>>> mystr.partition("itcast")  
('hello world ', 'itcast', ' and itcastcpp')  
>>> █
```

## <21>rpartition

类似于 partition()函数,不过是从右边开始.

```
mystr.rpartition(str)
```

```
>>> mystr = 'hello world itcast and itcastcpp'  
>>> mystr.partition("itcast")  
('hello world ', 'itcast', ' and itcastcpp')  
>>> mystr.rpartition("itcast")  
('hello world itcast and ', 'itcast', 'cpp')  
>>> █
```

## <22>splitlines

按照行分隔, 返回一个包含各行作为元素的列表

```
mystr.splitlines()
```

```
>>> mystr="hello\nworld"  
>>> print mystr  
hello  
world  
>>> mystr.splitlines()  
['hello', 'world']  
>>>
```

## <23>isalpha

如果 mystr 所有字符都是字母 则返回 True,否则返回 False

```
mystr.isalpha()
```

```
>>> mystr = 'abc'  
>>> mystr.isalpha()  
True  
>>> mystr = '123'  
>>> mystr.isalpha()  
False  
>>> mystr = 'abc 123'  
>>> mystr.isalpha()  
False  
>>> █
```

## <24>isdigit

如果 mystr 只包含数字则返回 True 否则返回 False.

```
mystr.isdigit()
```

```
>>> mystr = 'abc'  
>>> mystr.isdigit()  
False  
>>> mystr = '123'  
>>> mystr.isdigit()  
True  
>>> mystr = 'abc123'  
>>> mystr.isdigit()  
False  
>>> █
```

## <25>isalnum

如果 mystr 所有字符都是字母或数字则返回 True,否则返回 False

```
mystr.isalnum()
```

```
>>> mystr = '123'  
>>> mystr.isalnum()  
True  
>>> mystr = 'abc'  
>>> mystr.isalnum()  
True  
>>> mystr = 'abc123'  
>>> mystr.isalnum()  
True  
>>> mystr = 'abc 123'  
>>> mystr.isalnum()  
False  
>>> █
```

## <26>isspace

如果 mystr 中只包含空格，则返回 True，否则返回 False.

```
mystr.isspace()
```

```
>>> mystr = 'abc123'  
>>> mystr.isspace()  
False  
>>> mystr = ''  
>>> mystr.isspace()  
False  
>>> mystr = ' '  
>>> mystr.isspace()  
True  
>>> mystr = ' ' '  
>>> mystr.isspace()  
True  
>>> 
```

## <27>join

mystr 中每个字符后面插入str,构造出一个新的字符串

```
mystr.join(str)
```

```
>>> str = " "  
>>> li = ["my", "name", "is", "dongGe"]  
>>> str.join(li)  
'my name is dongGe'  
>>> str = "_"  
>>> str.join(li)  
'my_name_is_dongGe'  
>>> 
```

## 想一想

- （面试题）给定一个字符串aStr，返回使用空格或者'\t'分割后的倒数第二个子串

```
In [36]: testStr = "haha nihao a \t heihei \t woshi nide \t hao \npengyou"
In [37]: testStr.split("\t\n")
Out[37]: ['haha nihao a \t heihei \t woshi nide \t hao \npengyou']错误的处理方式
In [38]: testStr.split()
Out[38]: ['haha', 'nihao', 'a', 'heihei', 'woshi', 'nide', 'hao', 'pengyou']正确的处理方式
```

## 列表介绍

想一想：

这里写法类似java中的数据但是功能和list一样，因为其中的存储可以是不同的类型。

列表和字典都是可以存储不同元素的，列表相同的元素，字段是一个元素的不同属性

前面学习的字符串可以用来存储一串信息，那么想一想，怎样存储咱们班所有同学的名字呢？

定义100个变量，每个变量存放一个学生的姓名可行吗？有更好的办法吗？

答：

列表

### <1>列表的格式

变量A的类型为列表

```
namesList = ['xiaoWang', 'xiaoZhang', 'xiaoHua']
```

比C语言的数组强大的地方在于列表中的元素可以是不同类型的

```
testList = [1, 'a']
```

### <2>打印列表

demo:

```
namesList = ['xiaoWang', 'xiaoZhang', 'xiaoHua']
print(namesList[0])
print(namesList[1])
print(namesList[2])
```

结果：

```
xiaowang  
xiaoZhang  
xiaoHua
```

# 列表的循环遍历

## 1. 使用for循环

为了更有效率的输出列表的每个数据，可以使用循环来完成

demo:

```
namesList = ['xiaoWang', 'xiaoZhang', 'xiaoHua']
for name in namesList:
    print(name)
```

else:

print('-----')      循环完事之后会打印，其他语言都没有啊，奇葩。这个其实是和循环(不论是for还是while)成为了一体，就是说循环中有break的话就不会执行了，哈哈，一般都不怎么这么用

```
xiaoWang
xiaoZhang
xiaoHua
```

## 2. 使用while循环

为了更有效率的输出列表的每个数据，可以使用循环来完成

demo:

```
namesList = ['xiaoWang', 'xiaoZhang', 'xiaoHua']

length = len(namesList)

i = 0

while i < length:
    print(namesList[i])
    i+=1
```

结果：

```
xiaowang  
xiaoZhang  
xiaoHua
```

# 列表的相关操作

列表中存放的数据是可以进行修改的，比如“增”、“删”、“改”

## <1>添加元素("增"append, extend, insert)

### append

通过append可以向列表添加元素

```
a = ['aa', 'bb']
a = a.append('cc')
print(a) 输出为空
```

demo:

```
#定义变量A, 默认有3个元素
A = ['xiaoWang', 'xiaoZhang', 'xiaoHua']

print("----添加之前, 列表A的数据----")
for tempName in A:
    print(tempName)

#提示、并添加元素
temp = input('请输入要添加的学生姓名:')
A.append(temp)

print("----添加之后, 列表A的数据----")
for tempName in A:
    print(tempName)
```

结果：

```
----添加之前, 列表A的数据----
xiaoWang
xiaoZhang
xiaoHua
请输入要添加的学生姓名:■
```

## extend

通过extend可以将另一个集合中的元素逐一添加到列表中

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> a.append(b)
>>> a
[1, 2, [3, 4]]
>>> a.extend(b)
>>> a
[1, 2, [3, 4], 3, 4]
```

## insert

insert(index, object) 在指定位置index前插入元素object

```
>>> a = [0, 1, 2]
>>> a.insert(1, 3)
>>> a
[0, 3, 1, 2]
```

## <2>修改元素("改")

修改元素的时候，要通过下标来确定要修改的是哪个元素，然后才能进行修改

demo:

```
#定义变量A, 默认有3个元素
A = ['xiaoWang', 'xiaoZhang', 'xiaoHua']

print("-----修改之前, 列表A的数据-----")
for tempName in A:
    print(tempName)
```

```
#修改元素  
A[1] = 'xiaoLu'  
  
print("----修改之后，列表A的数据----")  
for tempName in A:  
    print(tempName)
```

结果：

```
----修改之前，列表A的数据----  
xiaoWang  
xiaoZhang  
xiaoHua  
----修改之后，列表A的数据----  
xiaoWang  
xiaoLu  
xiaoHua
```

### <3>查找元素("查"in, not in, index, count)

所谓的查找，就是看看指定的元素是否存在

#### in, not in

python中查找的常用方法为：

- in（存在），如果存在那么结果为true，否则为false
- not in（不存在），如果不存在那么结果为true，否则false

demo

```
#待查找的列表  
nameList = ['xiaoWang', 'xiaoZhang', 'xiaoHua']  
  
#获取用户要查找的名字  
findName = input('请输入要查找的姓名：')
```

```
#查找是否存在
if findName in nameList:
    print('在字典中找到了相同的名字')
else:
    print('没有找到')
```

结果1：(找到)

结果2：(没有找到)

说明：

| in的方法只要会用了，那么not in也是同样的用法，只不过not in判断的是不存在

## index, count

index和count与字符串中的用法相同

```
>>> a = ['a', 'b', 'c', 'a', 'b']
>>> a.index('a', 1, 3) # 注意是左闭右开区间
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'a' is not in list
>>> a.index('a', 1, 4)
3
>>> a.count('b')
2
>>> a.count('d')
```

## <4>删除元素("删"del, pop, remove)

类比现实生活中，如果某位同学调班了，那么就应该把这个条走后的学生的姓名删除掉；在开发中经常会用到删除这种功能。

列表元素的常用删除方法有：

- del：根据下标进行删除
- pop：删除最后一个元素
- remove：根据元素的值进行删除

del name[0]  
name.pop()  
name.remove('xxx') 只是删除第一个

demo:(del)

```
movieName = ['加勒比海盗', '骇客帝国', '第一滴血', '指环王', '霍比特人'

print('-----删除之前-----')
for tempName in movieName:
    print(tempName)

del movieName[2]

print('-----删除之后-----')
for tempName in movieName:
    print(tempName)
```

结果：

```
-----删除之前-----
加勒比海盗
骇客帝国
第一滴血
指环王
霍比特人
```

速度与激情

-----删除之后-----

加勒比海盗

骇客帝国

指环王

霍比特人

速度与激情

demo:(pop)

```
movieName = ['加勒比海盗', '骇客帝国', '第一滴血', '指环王', '霍比特人'

print('-----删除之前-----')
for tempName in movieName:
    print(tempName)

movieName.pop()

print('-----删除之后-----')
for tempName in movieName:
    print(tempName)
```

结果:

-----删除之前-----

加勒比海盗

骇客帝国

第一滴血

指环王

霍比特人

速度与激情

-----删除之后-----

加勒比海盗

骇客帝国

第一滴血

指环王

## 霍比特人

demo:(remove)

```
movieName = ['加勒比海盗', '骇客帝国', '第一滴血', '指环王', '霍比特人'

print('-----删除之前-----')
for tempName in movieName:
    print(tempName)

movieName.remove('指环王')

print('-----删除之后-----')
for tempName in movieName:
    print(tempName)
```

结果:

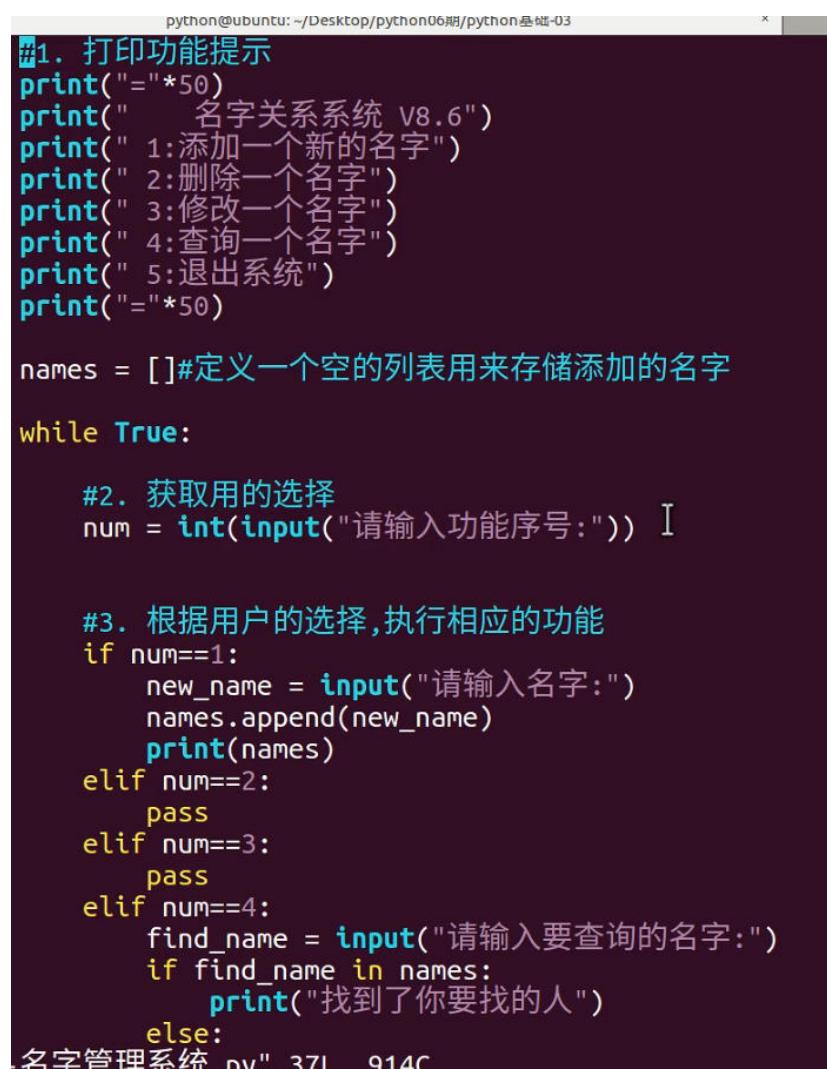
```
-----删除之前-----
加勒比海盗
骇客帝国
第一滴血
指环王
霍比特人
速度与激情
-----删除之后-----
加勒比海盗
骇客帝国
第一滴血
霍比特人
速度与激情
```

## <5>排序(sort, reverse)

sort方法是将list按特定顺序重新排列， 默认认为由小到大， 参数reverse=True可改为倒序， 由大到小。

reverse方法是将list逆置。

```
>>> a = [1, 4, 2, 3]
>>> a
[1, 4, 2, 3]
>>> a.reverse()
>>> a
[3, 2, 4, 1]
>>> a.sort()
>>> a
[1, 2, 3, 4]
>>> a.sort(reverse=True)
>>> a
[4, 3, 2, 1]
```



The screenshot shows a terminal window titled "python@ubuntu: ~/Desktop/python06期/python基础-03". The code is a script for managing names:

```
#1. 打印功能提示
print("*50")
print("    名字关系系统 V8.6")
print(" 1:添加一个新的名字")
print(" 2:删除一个名字")
print(" 3:修改一个名字")
print(" 4:查询一个名字")
print(" 5:退出系统")
print("*50)

names = []#定义一个空的列表用来存储添加的名字

while True:

    #2. 获取用的选择
    num = int(input("请输入功能序号:"))  I

    #3. 根据用户的选择,执行相应功能
    if num==1:
        new_name = input("请输入名字:")
        names.append(new_name)
        print(names)
    elif num==2:
        pass
    elif num==3:
        pass
    elif num==4:
        find_name = input("请输入要查询的名字:")
        if find_name in names:
            print("找到了你要找的人")
        else:
            print("没有找到该名字")
```

# 列表的嵌套

## 1. 列表嵌套

类似while循环的嵌套，列表也是支持嵌套的

一个列表中的元素又是一个列表，那么这就是列表的嵌套

```
schoolNames = [[['北京大学', '清华大学'],
                 ['南开大学', '天津大学', '天津师范大学'],
                 ['山东大学', '中国海洋大学']]]
```

## 2. 应用

一个学校，有3个办公室，现在有8位老师等待工位的分配，请编写程序，完成随机的分配

```
#encoding=utf-8

import random

# 定义一个列表用来保存3个办公室
offices = [[], [], []]

# 定义一个列表用来存储8位老师的名字
names = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']

i = 0
for name in names:
    index = random.randint(0, 2)
    offices[index].append(name)

i = 1
```

```
for tempNames in offices:  
    print('办公室%d的人数为:%d'%(i,len(tempNames)))  
    i+=1  
    for name in tempNames:  
        print("%s"%name,end=' ')  
    print("\n")  
    print("-"*20)
```

运行结果如下：

办公室 1 的人数为 :4  
ABCDE

-----  
办公室 2 的人数为 :3  
DGH

-----  
办公室 3 的人数为 :1  
F

# 元组

元祖的方法和列表都一样, 只是只能查, 不能增删改

Python的元组与列表类似, 不同之处在于元组的元素不能修改。元组使用小括号, 列表使用方括号。

```
>>> aTuple = ('et', 77, 99.9)
>>> aTuple
('et', 77, 99.9)
```

type(aTuple) 获取什么东西的类型, tuple元祖

## <1>访问元组

```
[>>> tuple=('hello',100,3.14)
[>>> tuple[0]
'hello'
[>>> tuple[1]
100
[>>> tuple[2]
3.14
>>> ]
```

## <2>修改元组

```
[>>> tuple[2]=188
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> ]
```

说明: python中不允许修改元组的数据, 包括不能删除其中的元素。

## <3>元组的内置函数count, index

index和count与字符串和列表中的用法相同

```
>>> a = ('a', 'b', 'c', 'a', 'b')
>>> a.index('a', 1, 3) # 注意是左闭右开区间
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
>>> a.index('a', 1, 4)
3
>>> a.count('b')
2
>>> a.count('d')
0
```

# 字典介绍

想一想：

如果有列表

```
nameList = ['xiaoZhang', 'xiaoWang', 'xiaoLi'];
```

需要对"xiaoWang"这个名字写错了，通过代码修改：

```
nameList[1] = 'xiaoxiaоРang'
```

如果列表的顺序发生了变化，如下

```
nameList = ['xiaoWang', 'xiaoZhang', 'xiaoLi'];
```

此时就需要修改下标，才能完成名字的修改

```
nameList[0] = 'xiaoxiaоРang'
```

有没有方法，既能存储多个数据，还能在访问元素的很方便就能够定位到需要的那个元素呢？

答：

字典

另一个场景：

学生信息列表，每个学生信息包括学号、姓名、年龄等，如何从中找到某个学生的信息？

```
>>> studens = [[1001, "王宝强", 24], [1002, "马蓉", 23], [1005, "
```

循环遍历? No!

## <1>生活中的字典



## <2>软件开发中的字典

变量info为字典类型：

```
info = {'name': '班长', 'id': 100, 'sex': 'f', 'address': '地球亚}
```

key为string, value为任何东西

说明：

- 字典和列表一样，也能够存储多个数据
- 列表中找某个元素时，是根据下标进行的
- 字典中找某个元素时，是根据'名字'（就是冒号前面的那个值，例如上面代码中的'name'、'id'、'sex'）
- 字典的每个元素由2部分组成，键:值。例如 'name':'班长', 'name'为键，'班长'为值

### <3>根据键访问值

还有get获取方式

```
info = {'name': '班长', 'id': 100, 'sex': 'f', 'address': '地球亚洲'}  
print(info['name'])  
print(info['address'])
```

结果：

```
班长  
地球亚洲中国北京
```

若访问不存在的键，则会报错：

```
>>> info['age']  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'age'
```

在我们不确定字典中是否存在某个键而又想获取其值时，可以使用get方法，还可以设置默认值：

```
>>> age = info.get('age')
>>> age # 'age'键不存在, 所以age为None
>>> type(age)
<type 'NoneType'>
>>> age = info.get('age', 18) # 若info中不存在'age'这个键, 就返回默认值
>>> age
18
```

# 字典的常见操作1

## <1>修改元素

字典的每个元素中的数据是可以修改的，只要通过key找到，即可修改

demo:

```
info = {'name': '班长', 'id': 100, 'sex': 'f', 'address': '地球亚洲'}

newId = input('请输入新的学号')

info['id'] = int(newId)

print('修改之后的id为%d: %s' % info['id'])
```

结果：

增加	<code>name[new_key] = value</code>
删除	<code>del name[old_key]</code>
改	<code>name[old_key] = new_value</code>
查	<code>name.get('key', 19)</code> : 不一定有给了一个默认值 <code>name['key']</code> key不存在的话就会造成creash

## <2>添加元素

demo: 访问不存在的元素

```
info = {'name': '班长', 'sex': 'f', 'address': '地球亚洲中国北京'}

print('id为:%d' % info['id'])
```

结果：

```
MacBook-Pro 01-python基础班 - 资料 $ python test.py
File "test.py", line 10
    print 'id为：' info['id']
          ^
SyntaxError: invalid syntax
```

如果在使用 **变量名['键'] = 数据** 时，这个“键”在字典中，不存在，那么就会新增这个元素

demo:添加新的元素

```
info = {'name': '班长', 'sex': 'f', 'address': '地球亚洲中国北京'}

# print('id为:%d'%info['id'])#程序会终端运行，因为访问了不存在的键

newId = input('请输入新的学号')

info['id'] = newId

print('添加之后的id为:%d'%info['id'])
```

结果：

```
请输入新的学号188
添加之后的id为： 188
```

## <3>删除元素

对字典进行删除操作，有一下几种：

- del
- clear()

demo:del删除指定的元素

```
info = {'name': '班长', 'sex': 'f', 'address': '地球亚洲中国北京'}

print('删除前,%s'%info['name'])

del info['name']

print('删除后,%s'%info['name'])
```

## 结果

MacBook-Pro 01-python基础班-资料 \$ python test.py  
删除前，班长  
删除后，  
Traceback (most recent call last):  
 File "test.py", line 9, in <module>  
 print '删除后,',info['name']  
KeyError: 'name'

删除后不能访问

## demo:del删除整个字典

```
info = {'name': 'monitor', 'sex': 'f', 'address': 'China'}

print('删除前,%s'%info)

del info

print('删除后,%s'%info)
```

## 结果

MacBook-Pro 01-python基础班-资料 \$ python test.py  
删除前，{'address': 'China', 'name': 'monitor', 'sex': 'f'}  
删除后，  
Traceback (most recent call last):  
 File "test.py", line 10, in <module>  
 print '删除后,',info  
NameError: name 'info' is not defined

## demo:clear清空整个字典

```
info = {'name':'monitor', 'sex':'f', 'address':'China'}  
  
print('清空前,%s'%info)  
  
info.clear()  
  
print('清空后,%s'%info)
```

## 结果

```
MacBook-Pro 01-python基础班-资料$ python test.py  
清空前, {'address': 'China', 'name': 'monitor', 'sex': 'f'}  
清空后, {}
```

元祖拆包

## 字典的常见操作2

### <1>len()

测量字典中，键值对的个数

```
[>>> dict = {"name":'zhangsan', 'sex':'m'}
[>>> len(dict)
2
>>> ]
```

```
[1]: a = (11,22)
[2]: b = a
[3]: b
[3]: (11, 22)
[4]: c,d = a = (11,22)
[5]: c
[5]: 11
[6]: d
[6]: 22
```

### <2>keys

返回一个包含字典所有KEY的列表

```
[>>> dict = {"name":'zhangsan', 'sex':'m'}
[>>> dict.keys()
['name', 'sex']
>>> ]
```

### <3>values

返回一个包含字典所有value的列表

```
[>>> dict = {"name":'zhangsan', 'sex':'m'}
[>>> dict.values()
['zhangsan', 'm']
>>> ]
```

### <4>items

返回一个包含所有（键， 值）元祖的列表

```
[>>> dict = {"name":'zhangsan', 'sex':'m'}  
[>>> dict.items()  
[('name', 'zhangsan'), ('sex', 'm')]  
>>> █
```

## <5>has\_key

dict.has\_key(key)如果key在字典中，返回True，否则返回False

```
[>>> dict = {"name":'zhangsan', 'sex':'m'}  
[>>> dict.has_key('name')  
True  
[>>> dict.has_key('phone')  
False  
>>> █
```

# 遍历

通过for ... in ...的语法结构，我们可以遍历字符串、列表、元组、字典等数据结构。

注意python语法的缩进

## 字符串遍历

```
>>> a_str = "hello itcast"
>>> for char in a_str:
...     print(char, end=' ')
...
h e l l o    i t c a s t
```

## 列表遍历

```
>>> a_list = [1, 2, 3, 4, 5]
>>> for num in a_list:
...     print(num, end=' ')
...
1 2 3 4 5
```

## 元组遍历

```
>>> a_turple = (1, 2, 3, 4, 5)
>>> for num in a_turple:
...     print(num, end=" ")
1 2 3 4 5
```

## 字典遍历

### <1> 遍历字典的key（键）

```
[>>> dict = {"name":'zhangsan', 'sex':'m'}
[>>> for key in dict.keys():
[...     print key
[...
name
sex
>>> ]]
```

### <2> 遍历字典的value（值）

```
[>>> dict = {"name":'zhangsan', 'sex':'m'}
[>>> for value in dict.values():
[...     print value
[...
zhangsan
m
>>> ]]
```

### <3> 遍历字典的项（元素）

```
[>>> dict = {"name":'zhangsan', 'sex':'m'}
[>>> for item in dict.items():
[...     print item
[...
('name', 'zhangsan')
('sex', 'm')
>>> ]]
```

### <4> 遍历字典的key-value（键值对）

```
[>>> dict = {"name":'zhangsan', 'sex':'m'}
[>>> for key,value in dict.items():
[...     print("key=%s,value=%s"%(key,value))
[...
key=name,value=zhangsan
key=sex,value=m
>>> ]]
```

# 想一想，如何实现带下标索引的遍历

```
>>> chars = ['a', 'b', 'c', 'd']
>>> i = 0
>>> for chr in chars:
...     print("%d %s"%(i, chr))
...     i += 1
...
0 a
1 b
2 c
3 d
```

## enumerate()

```
>>> chars = ['a', 'b', 'c', 'd']
>>> for i, chr in enumerate(chars):
...     print i, chr
...
0 a
1 b
2 c
3 d
```

# 公共方法

## 运算符

运算符	Python 表达式	结果	描述	支持的数据类型
+	[1, 2] + [3, 4]	[1, 2, 3, 4]	合并	字符串、列表、元组
*	'Hi!' * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	复制	字符串、列表、元组
in	3 in (1, 2, 3)	True	元素是否存在	字符串、列表、元组、字典
not in	4 not in (1, 2, 3)	True	元素是否不存在	字符串、列表、元组、字典

+

```
>>> "hello " + "itcast"
'hello itcast'
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
>>> ('a', 'b') + ('c', 'd')
('a', 'b', 'c', 'd')
```

\*

```
>>> 'ab'*4
'ababab'
>>> [1, 2]*4
[1, 2, 1, 2, 1, 2, 1, 2]
>>> ('a', 'b')*4
```

```
('a', 'b', 'a', 'b', 'a', 'b', 'a', 'b')
```

## in

```
>>> 'itc' in 'hello itcast'  
True  
>>> 3 in [1, 2]  
False  
>>> 4 in (1, 2, 3, 4)  
True  
>>> "name" in {"name": "Delron", "age": 24}  
True
```

注意，in在对字典操作时，判断的是字典的键

## python内置函数

Python包含了以下内置函数

序号	方法	描述
1	cmp(item1, item2)	比较两个值
2	len(item)	计算容器中元素个数
3	max(item)	返回容器中元素最大值
4	min(item)	返回容器中元素最小值
5	del(item)	删除变量

## cmp

```
>>> cmp("hello", "itcast")  
-1  
>>> cmp("itcast", "hello")  
1
```

```
>>> cmp("itcast", "itcast")
0
>>> cmp([1, 2], [3, 4])
-1
>>> cmp([1, 2], [1, 1])
1
>>> cmp([1, 2], [1, 2, 3])
-1
>>> cmp({"a":1}, {"b":1})
-1
>>> cmp({"a":2}, {"a":1})
1
>>> cmp({"a":2}, {"a":2, "b":1})
-1
```

注意： **cmp**在比较字典数据时，先比较键，再比较值。

## len

```
>>> len("hello itcast")
12
>>> len([1, 2, 3, 4])
4
>>> len((3,4))
2
>>> len({"a":1, "b":2})
2
```

注意： **len**在操作字典数据时，返回的是键值对个数。

## max

```
>>> max("hello itcast")
't'
>>> max([1, 4, 522, 3, 4])
522
```

```
>>> max({"a":1, "b":2})  
'b'  
>>> max({"a":10, "b":2})  
'b'  
>>> max({"c":10, "b":2})  
'c'
```

## del

del有两种用法，一种是del加空格，另一种是del()

```
>>> a = 1  
>>> a  
1  
>>> del a  
>>> a  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'a' is not defined  
>>> a = ['a', 'b']  
>>> del a[0]  
>>> a  
['b']  
>>> del(a)  
>>> a  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'a' is not defined
```

## 多维列表/元祖访问的示例

```
>>> tuple1 = [(2,3),(4,5)]  
>>> tuple1[0]  
(2, 3)  
>>> tuple1[0][0]
```

```
2
>>> tuple1[0][2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>> tuple1[0][1]
3
>>> tuple1[2][2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> tuple2 = tuple1+[(3)]
>>> tuple2
[(2, 3), (4, 5), 3]
>>> tuple2[2]
3
>>> tuple2[2][0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not subscriptable
```

# 引用

## 想一想

```
>>> a = 1  
>>> b = a  
>>> b  
1  
>>> a = 2  
>>> a  
2
```

说就是1

请问此时b的值为多少?

```
>>> a = [1, 2]  
>>> b = a  
>>> b  
[1, 2]  
>>> a.append(3)  
>>> a  
[1, 2, 3]
```

b也是123, 因为是列表, 有点类似于java中的地址值

请问此时b的值又是多少?

# 引用

在python中，值是靠引用来传递来的。

我们可以用id()来判断两个变量是否为同一个值的引用。我们可以将id值理解为那块内存的地址标示。

```
>>> a = 1
```

type(a) 看类型

```
>>> b = a
>>> id(a)
13033816
>>> id(b) # 注意两个变量的id值相同
```

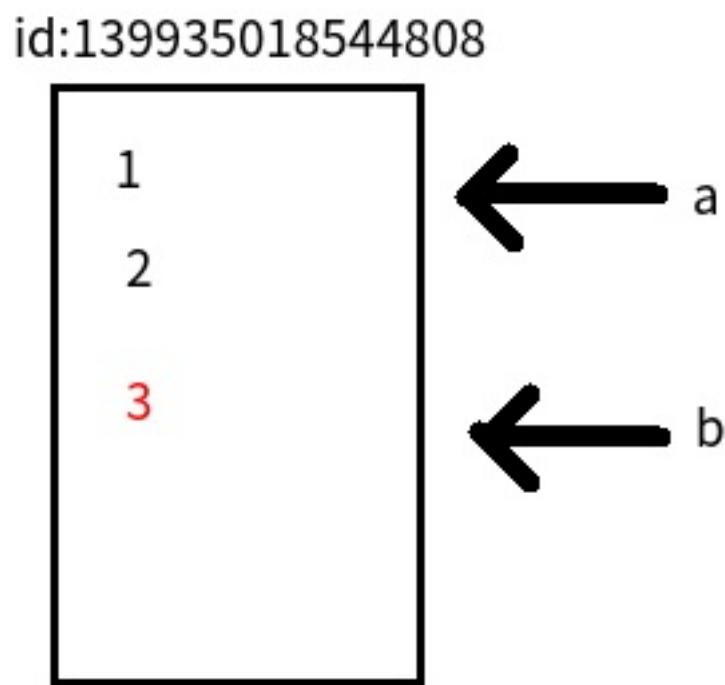
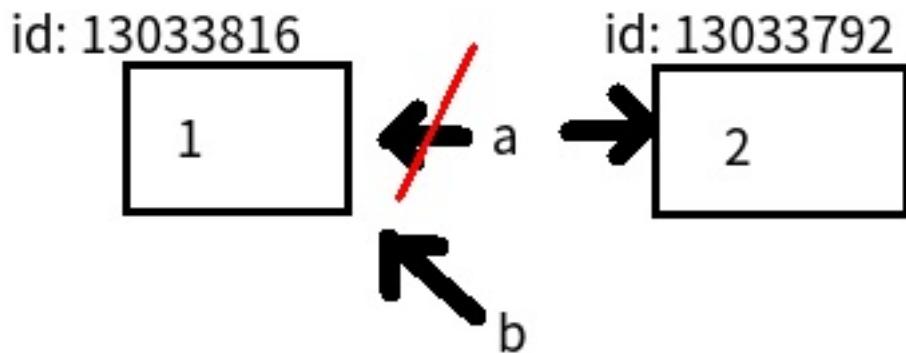
```
13033816
>>> a = 2
>>> id(a) # 注意a的id值已经变了
13033792
>>> id(b) # b的id值依旧
```

13033816

数字, 字符串, 元祖: 三者是不可修改的.  
 列表字典可变, 字典key除了列表和字典不可其他都  
 行, 因为是将key进行hash算法, 可变类型不能进行hash  
 算法

```
>>> a = [1, 2]
>>> b = a
>>> id(a)
139935018544808
>>> id(b)
139935018544808
>>> a.append(3)
>>> a
[1, 2, 3]
>>> id(a)
139935018544808
>>> id(b) # 注意a与b始终指向同一个地址
139935018544808
```

a = '你好啊' a[0] = 'h' 错误 a = '得到的' 这是  
 重新指向, 原来的字符串并没有被修改是被抛弃了.



## 可变类型与不可变类型

可变类型，值可以改变：

- 列表 list
- 字典 dict

不可变类型，值不可以改变：

- 数值类型 int, long, bool, float
- 字符串 str

- 元组 tuple

怎样交换两个变量的值？

# 作业

**1. 编程实现对一个元素全为数字的列表，求最大值、最小值**

**2. 编写程序，完成以下要求：**

- 统计字符串中，各个字符的个数
- 比如：“hello world” 字符串统计的结果为： h:1 e:1 l:3 o:2 d:1 r:1 w:1

**3. 编写程序，完成以下要求：**

- 完成一个路径的组装
- 先提示用户多次输入路径，最后显示一个完成的路径，比如 /home/python/ftp/share

**4. 编写程序，完成“名片管理器”项目**

- 需要完成的基本功能：
  1. 添加名片
  2. 删除名片
  3. 修改名片
  4. 查询名片
  5. 退出系统
- 程序运行后，除非选择退出系统，否则重复执行功能

单列map集合一样

```
pass
elif num==4:
    find_name = input("请输入要查找的姓名:")
    find_flag = 0#默认表示没有找到
    for temp in card_infor:
        if find_name == temp["name"]:
            print("%s\t%s\t%s\t%s"%(temp['name'], temp['qq'], temp['weixin'], temp['addr']))
            find_flag=1#表示找到了
            break
    #判断是否找到了
    if find_flag == 0:
        print("查无此人....")
elif num==5:
    print("姓名\tQQ\t微信\t住址")
    for temp in card_infor:
        print("%s\t%s\t%s\t%s"%(temp['name'], temp['qq'], temp['weixin'], temp['addr']))
elif num==6:
    break
```

# 函数介绍

## <1>什么是函数

请看如下代码：

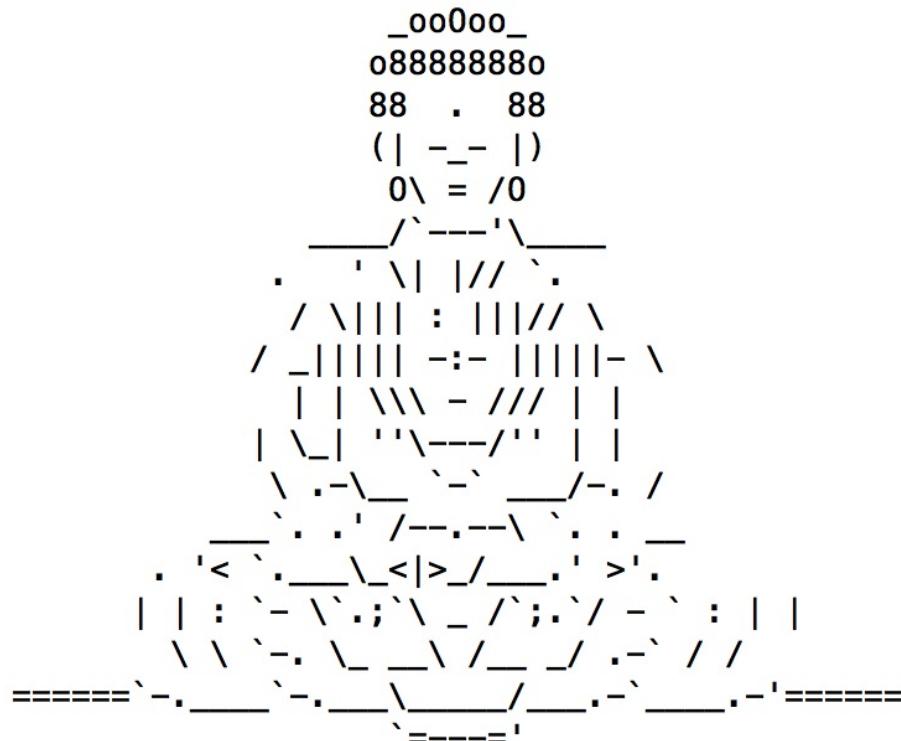
```

print "          _000oo_   "
print "          o8888888o   "
print "          88 . 88   "
print "          (| -_- |)   "
print "          0\\ \\ = /0   "
print "          ____/_`---'\\\\____   "
print "          . . ' \\ \\| | // ` . .   "
print "          / \\ \\||| : ||| // \\ \\   "
print "          / _||| | -:- ||| | - \\ \\   "
print "          | | \\ \\ \\ \\ \\ - // / | |   "
print "          | \\ \\| | ' \\ \\ --- / ' | |   "
print "          \\ \\ . - \\ \\ _ ` -` _ / - . /   "
print "          _ _ ` . . ' / - - - \\ \\ ` . . _ _   "
print "          .''' ' < ` . __ \\ \\ < | > _ / __ . ' > '''' .   "
print "          | | : ^ - \\ \\ ^ ; ` \\ \\ _ / ^ ; . ^ / - ^ : | |   "
print "          \\ \\ \\ \\ ^ - . \\ \\ _ __ \\ \\ / _ _ / . - ^ / / '   "
print "          =====` - . __ ` - . __ \\ \\ _ / _ / . - ^ _ / . - ' =====
print "          "
print "          ..... "
print "          佛祖镇楼           BUG辟易   "
print "          佛曰：   "
print "          写字楼里写字间，写字间里程序员；   "
print "          程序人员写程序，又拿程序换酒钱。   "
print "          酒醒只在网上坐，酒醉还来网下眠；   "
print "          酒醉酒醒日复日，网上网下年复年。   "
print "          但愿老死电脑间，不愿鞠躬老板前；   "
print "          奔驰宝马贵者趣，公交自行程序员。   "
print "          别人笑我忒疯癫，我笑自己命太贱；   "

```

```
print "不见满街漂亮妹，哪个归得程序员？"
```

运行后的现象：



佛祖镇楼

BUG辟易

佛曰：

写字楼里写字间，写字间里程序员；  
程序人员写程序，又拿程序换酒钱。  
酒醒只在网上坐，酒醉还来网下眠；  
酒醉酒醒日复日，网上网下年复年。  
但愿老死电脑间，不愿鞠躬老板前；  
奔驰宝马贵者趣，公交自行程序员。  
别人笑我忒疯癫，我笑自己命太贱；  
不见满街漂亮妹，哪个归得程序员？

想一想：

如果一个程序在不同的地方需要输出“佛祖镇楼”，程序应该怎样设计？

if 条件1:

    输出‘佛祖镇楼’

... (省略) ...

```
if 条件2:  
    输出‘佛祖镇楼’  
  
    ... (省略) ...
```

如果需要输出多次，是否意味着要编写这块代码多次呢？

## 小总结：

- 如果在开发程序时，需要某块代码多次，但是为了提高编写的效率以及代码的重用，所以把具有独立功能的代码块组织为一个小模块，这就是函数

# 函数定义和调用

## <1> 定义函数

定义函数的格式如下：

```
def 函数名():
    代码
```

demo:

```
# 定义一个函数，能够完成打印信息的功能
def printInfo():
    print '-----'
    print '        人生苦短，我用Python'
    print '-----'
```

## <2> 调用函数

定义了函数之后，就相当于有了一个具有某些功能的代码，想要让这些代码能够执行，需要调用它

调用函数很简单的，通过 **函数名()** 即可完成调用

demo:

```
# 定义完函数后，函数是不会自动执行的，需要调用它才可以
printInfo()
```

## <3>练一练

要求：定义一个函数，能够输出自己的姓名和年龄，并且调用这个函数让它执行

- 使用**def**定义函数
- 编写完函数之后，通过**函数名()**进行调用

## 函数的文档说明

```
>>> def test(a,b):  
...     "用来完成对2个数求和"  
...     print("%d"%(a+b))  
...  
>>>  
>>> test(11,22)  
33
```

如果执行，以下代码

```
>>> help(test)
```

能够看到test函数的相关说明

```
Help on function test in module __main__:  
  
test(a, b)  
    用来完成对2个数求和  
(END)
```

# 函数参数(一)

思考一个问题，如下：

现在需要定义一个函数，这个函数能够完成2个数的加法运算，并且把结果打印出来，该怎样设计？下面的代码可以吗？有什么缺陷吗？

```
def add2num():
    a = 11
    b = 22
    c = a+b
    print c
```

为了让一个函数更通用，即想让它计算哪两个数的和，就让它计算哪两个数的和，在定义函数的时候可以让函数接收数据，就解决了这个问题，这就是函数的参数

## <1> 定义带有参数的函数

示例如下：

```
def add2num(a, b):
    c = a+b
    print c
```

## <2> 调用带有参数的函数

以调用上面的add2num(a, b)函数为例：

```
def add2num(a, b):
    c = a+b
```

```
print c  
  
add2num(11, 22) #调用带有参数的函数时，需要在小括号中，传递数据
```

调用带有参数函数的运行过程：

→ #定义接收2个参数的函数  
def add2num(a, b):  
 c = a+b  
 print c

#调用带有参数的函数  
add2num(110, 22)

终端

## <3> 练一练

要求：定义一个函数，完成前2个数完成加法运算，然后对第3个数，进行减法；然后调用这个函数

- 使用def定义函数，要注意有3个参数
- 调用的时候，这个函数定义时有几个参数，那么就需要传递几个参数

## <4> 调用函数时参数的顺序

```
>>> def test(a, b):  
...     print(a, b)  
...  
>>> test(1, 2)  
1 2
```

```
>>> test(b=1, a=2)
2 1
>>>
>>> test(b=1, 2)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
>>>
>>>
```

## <4> 小总结

- 定义时小括号中的参数，用来接收参数用的，称为“形参”
- 调用时小括号中的参数，用来传递给函数用的，称为“实参”

## 函数返回值(一)

### <1>“返回值”介绍

现实生活中的场景:

我给儿子10块钱，让他给我买包烟。这个例子中，10块钱是我给儿子的，就相当于调用函数时传递到参数，让儿子买烟这个事情最终的目标是，让他把烟给你带回来然后给你对么，，，此时烟就是返回值

开发中的场景:

定义了一个函数，完成了获取室内温度，想一想是不是应该把这个结果给调用者，只有调用者拥有了这个返回值，才能够根据当前的温度做适当的调整

综上所述:

- 所谓“返回值”，就是程序中函数完成一件事情后，最后给调用者的结果

### <2>带有返回值的函数

想要在函数中把结果返回给调用者，需要在函数中使用return

如下示例:

```
def add2num(a, b):  
    c = a+b  
    return c
```

或者

```
def add2num(a, b):  
    return a+b
```

## <3>保存函数的返回值

在本小节刚开始的时候，说过的“买烟”的例子中，最后儿子给你烟时，你一定是从儿子手中接过来 对么，程序也是如此，如果一个函数返回了一个数据，那么想要用这个数据，那么就需要保存

保存函数的返回值示例如下：

```
#定义函数
def add2num(a, b):
    return a+b

#调用函数，顺便保存函数的返回值
result = add2num(100, 98)

#因为result已经保存了add2num的返回值，所以接下来就可以使用了
print result
```

结果：

```
198
```

# 4种函数的类型

函数根据有没有参数，有没有返回值，可以相互组合，一共有4种

- 无参数，无返回值
- 无参数，有返回值
- 有参数，无返回值
- 有参数，有返回值

## <1>无参数，无返回值的函数

此类函数，不能接收参数，也没有返回值，一般情况下，打印提示灯类似的功能，使用这类的函数

```
def printMenu():
    print('-----')
    print('      xx涮涮锅 点菜系统')
    print('')
    print('  1. 羊肉涮涮锅')
    print('  2. 牛肉涮涮锅')
    print('  3. 猪肉涮涮锅')
    print('-----')
```

结果：

```
-----  
      xx涮涮锅 点菜系统  
  
  1. 羊肉涮涮锅  
  2. 牛肉涮涮锅  
  3. 猪肉涮涮锅  
-----
```

## <2>无参数，有返回值的函数

此类函数，不能接收参数，但是可以返回某个数据，一般情况下，像采集数据，用此类函数

```
# 获取温度
def getTemperature():

    #这里是获取温度的一些处理过程

    #为了简单起见，先模拟返回一个数据
    return 24

temperature = getTemperature()
print('当前的温度为：%d' % temperature)
```

结果：

```
当前的温度为： 24
```

## <3>有参数，无返回值的函数

此类函数，能接收参数，但不可以返回数据，一般情况下，对某些变量设置数据而不需结果时，用此类函数

## <4>有参数，有返回值的函数

此类函数，不仅能接收参数，还可以返回某个数据，一般情况下，像数据处理并需要结果的应用，用此类函数

```
# 计算1~num的累积和
def calculateNum(num):
```

```
result = 0
i = 1
while i<=num:

    result = result + i

    i+=1

return result

result = calculateNum(100)
print('1~100的累积和为：%d'%result)
```

结果：

```
1~100的累积和为： 5050
```

## <5>小总结

- 函数根据有没有参数，有没有返回值可以相互组合
- 定义函数时，是根据实际的功能需求来设计的，所以不同开发人员编写的函数类型各不相同

# 函数的嵌套调用

```
def testB():
    print('---- testB start----')
    print('这里是testB函数执行的代码... (省略)... ')
    print('---- testB end----')

def testA():
    print('---- testA start----')

    testB()

    print('---- testA end----')

testA()
```

结果：

```
---- testA start----
---- testB start----
这里是testB函数执行的代码... (省略)...
---- testB end----
---- testA end----
```

## 小总结：

- 一个函数里面又调用了另外一个函数，这就是所谓的函数嵌套调用

```
def testB():
    print('--- testB start----')
    print('这里是 testB函数执行的代码...(省略)...')
    print('---- testB end----')

def testA():
    print('---- testA start----')
    testB()
    print('---- testA end----')

testA()
```

- 如果函数A中，调用了另外一个函数B，那么先把函数B中的任务都执行完毕之后才会回到上次 函数A执行的位置

# 函数应用：打印图形和数学计算

## 目标

- 感受函数的嵌套调用
- 感受程序设计的思路,复杂问题分解为简单问题

## 思考&实现1

1. 写一个函数打印一条横线
2. 打印自定义行数的横线

## 参考代码1

```
# 打印一条横线
def printOneLine():
    print("-"*30)

# 打印多条横线
def printNumLine(num):
    i=0

        # 因为printOneLine函数已经完成了打印横线的功能,
        # 只需要多次调用此函数即可
    while i<num:
        printOneLine()
        i+=1

printNumLine(3)
```

## 思考&实现2

1. 写一个函数求三个数的和
2. 写一个函数求三个数的平均值

## 参考代码2

```
# 求3个数的和
def sum3Number(a,b,c):
    return a+b+c # return 的后面可以是数值, 也可是一个表达式

# 完成对3个数求平均值
def average3Number(a,b,c):

    # 因为sum3Number函数已经完成了3个数的就和, 所以只需调用即可
    # 即把接收到的3个数, 当做实参传递即可
    sumResult = sum3Number(a,b,c)
    aveResult = sumResult/3.0
    return aveResult

# 调用函数, 完成对3个数求平均值
result = average3Number(11,2,55)
print("average is %d"%result)
```

# 局部变量

## <1>什么是局部变量

如下图所示：

```

test.py - Sublime Text
test.py
1 #coding=utf-8
2
3 def test1():
4     a = 300
5     print('----test1--修改前--a=%d'%a)
6     a = 200
7     print('----test1--修改后--a=%d'%a)
8
9 def test2():
10    a = 400
11    print('----test3----a=%d'%a)
12
13
14 # 调用函数
15 test1()
16 test2()
17

```

局部变量

```

python@ubuntu:~/Desktop$ python3 test.py
----test1--修改前--a=300
----test1--修改后--a=200
----test3----a=400
python@ubuntu:~/Desktop$
```

## <2>小总结

- 局部变量，就是在函数内部定义的变量
- 不同的函数，可以定义相同的名字的局部变量，但是各用个的不会产生影响
- 局部变量的作用，为了临时保存数据需要在函数中定义变量来进行存储，这就是它的作用

# 全局变量

## <1>什么是全局变量

如果一个变量，既能在一个函数中使用，也能在其他的函数中使用，这样的变量就是 全局变量

demo如下：

```
# 定义全局变量
a = 100

def test1():
    print(a)

def test2():
    print(a)

# 调用函数
test1()
test2()
```

运行结果：

```
python@ubuntu:~/Desktop$ python3 test.py
100
100
```

## <2>全局变量和局部变量名字相同问题

看如下代码：

The screenshot shows a Sublime Text editor with a file named 'test.py'. The code defines a global variable 'a' (line 3) and two functions: 'test1' (lines 5-11) and 'test2' (lines 13-14). Inside 'test1', there is a local variable 'a' (line 7) and a print statement (line 9). Inside 'test2', there is another print statement (line 14). A red box highlights the line 'a = 100' with the text '全局变量'. Another red box highlights the line 'a = 300' with the text '局部变量'. To the right, a terminal window shows the output of running the script: it prints 'a=300' before the modification in test1, changes it to 'a=200', and then prints 'a=100' again.

## <3>修改全局变量

既然全局变量，就是能够在所有的函数中进行使用，那么可否进行修改呢？

代码如下：

重不重名都建议加上 global . 但是列表和字典是不需要的

The screenshot shows a Sublime Text editor with the same 'test.py' code as the previous example. A red box highlights the 'global a' declaration in the 'test1' function (line 7). An arrow points from this box to a blue text box containing the note: '重不重名都建议加上 global . 但是列表和字典是不需要的'. To the right, a terminal window shows the output of running the script: it prints 'a=100' before the modification in test1, changes it to 'a=200', and then prints 'a=200' again. A red box highlights the word 'ok' in the terminal output.

先调用了函数而在后边定义全部变量是不可以的, 和 java 不一样, 一般 python 顺序: 先全局变量, 后函数, 再调用

## <4>总结1:

- 在函数外边定义的变量叫做 全局变量
- 全局变量能够在所有的函数中进行访问
- 如果在函数中修改全局变量，那么就需要使用 `global` 进行声明，否则

出错

- 如果全局变量的名字和局部变量的名字相同，那么使用的是局部变量的，小技巧 强龙不压地头蛇

## <5>可变类型的全局变量

```

>>> a = 1
>>> def f():
...     a += 1
...     print a
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 'a' referenced before assignment
>>>
>>>
>>> li = [1,]
>>> def f2():
...     li.append(1)
...     print li
...
>>> f2()
[1, 1]
>>> li
[1, 1]

```

## <5>总结2：

- 在函数中不使用global声明全局变量时不能修改全局变量的本质是不能修改全局变量的指向，即不能将全局变量指向新的数据。
- 对于不可变类型的全局变量来说，因其指向的数据不能修改，所以不使

用global时无法修改全局变量。

- 对于可变类型的全局变量来说，因其指向的数据可以修改，所以不使用global时也可修改全局变量。

# 函数应用：学生管理系统

## 函数返回值(二)

在python中我们可不可以返回多个值？

```
>>> def divid(a, b):
...     shang = a//b
...     yushu = a%b
...     return shang, yushu
...
>>> sh, yu = divid(5, 2)
>>> sh
5
>>> yu
1
```

本质是利用了元组

# 函数参数(二)

## 1. 缺省参数

调用函数时，缺省参数的值如果没有传入，则被认为是默认值。下例会打印默认的age，如果age没有被传入：

```
def printinfo( name, age = 35 ):
    # 打印任何传入的字符串
    print "Name: ", name
    print "Age ", age

# 调用printinfo函数
printinfo(name="miki" )
printinfo( age=9, name="miki" )
```

命名参数 不写的话就是从前往后赋值，和java有区别

以上实例输出结果：

```
Name: miki
Age 35
Name: miki
Age 9
```

**注意：带有默认值的参数一定要位于参数列表的最后面。**

```
>>> def printinfo(name, age=35, sex):
...     print name
...
File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

## 2. 不定长参数

有时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，声明时不会命名。

基本语法如下：

```
def functionname([formal_args], *args, **kwargs):
    "函数_文档字符串"
    function_suite
    return [expression]
```

用的时候可都没有\*

加了星号 (\*) 的变量args会存放所有未命名的变量参数，args为元组；而加\*\*的变量kwargs会存放命名参数，即形如key=value的参数，kwargs为字典。

```
>>> def fun(a, b, *args, **kwargs):
...     """可变参数演示示例"""
...     print "a =", a
...     print "b =", b
...     print "args =", args
...     print "kwargs: "
...     for key, value in kwargs.items():
...         print key, "=" , value
...
>>> fun(1, 2, 3, 4, 5, m=6, n=7, p=8) # 注意传递的参数对应
a = 1
b = 2
args = (3, 4, 5)
kwargs:
p = 8
m = 6
n = 7
>>>
>>>
>>>
>>> c = (3, 4, 5)
>>> d = {"m":6, "n":7, "p":8}
>>> fun(1, 2, *c, **d)      # 注意元组与字典的传参方式
```

赋值和拆包

```
894 * 518 python@ubuntu:~/Desktop/python06/pthon基础05
1 def test(a,b,c=33,*args,**kwargs):
2     print(a)
3     print(b)
4     print(c)
5     print(args)
6     print(kwargs)
7
8
9 #test(11,22,33,44,55,66,task=99,done=89)
10
11 A = (44,55,66)
12 B = {"name":"laowang","age":18}
13
14 test(11,22,33,*A,**B)
```

```

a = 1
b = 2
args = (3, 4, 5)
kwargs:
p = 8
m = 6
n = 7
>>>
>>>
>>>
>>> fun(1, 2, c, d) # 注意不加星号与上面的区别
a = 1
b = 2
args = ((3, 4, 5), {'p': 8, 'm': 6, 'n': 7})
kwargs:
>>>
>>>

```

### 3. 引用传参

- 可变类型与不可变类型的变量分别作为函数参数时，会有什么不同吗？
- Python有没有类似C语言中的指针传参呢？

```

>>> def selfAdd(a):
...     """自增"""
...     a += a
...
>>> a_int = 1
>>> a_int
1
>>> selfAdd(a_int)
>>> a_int
1
>>> a_list = [1, 2]
>>> a_list
[1, 2]

```

```
>>> selfAdd(a_list)
>>> a_list
[1, 2, 1, 2]
```

Python中函数参数是引用传递（注意不是值传递）。对于不可变类型，因变量不能修改，所以运算不会影响到变量自身；而对于可变类型来说，函数体中的运算有可能会更改传入的参数变量。

## 想一想为什么

```
>>> def selfAdd(a):
...     """自增"""
...     a = a + a    # 我们更改了函数体的这句话
...
>>> a_int = 1
>>> a_int
1
>>> selfAdd(a_int)
>>> a_int
1
>>> a_list = [1, 2]
>>> a_list
[1, 2]
>>> selfAdd(a_list)
>>> a_list
[1, 2]      # 想一想为什么没有变呢?
```

# 递归函数

## <1>什么是递归函数

通过前面的学习知道一个函数可以调用其他函数。

如果一个函数在内部不调用其它的函数，而是自己本身的话，这个函数就是递归函数。

## <2>递归函数的作用

举个例子，我们来计算阶乘  $n! = 1 * 2 * 3 * \dots * n$

解决办法1：

The screenshot shows a Sublime Text editor with a file named 'test.py'. The code defines a function 'calNum' that calculates the factorial of a number 'num' using a while loop. The terminal window to the right shows the command 'python3 test.py' being run, and the output '6' is displayed.

```

/test.py - Sublime Text
test.py

1 #coding=utf-8
2
3 def calNum(num):
4     i = 1
5     result = 1
6
7     while i<=num:
8         result *= i
9         i+=1
10
11     return result
12
13 ret = calNum(3)
14 print(ret)
15
16

```

使用循环来完成

## 看阶乘的规律

```

1! = 1
2! = 2 × 1 = 2 × 1!
3! = 3 × 2 × 1 = 3 × 2!
4! = 4 × 3 × 2 × 1 = 4 × 3!
...
n! = n × (n-1)!

```

## 解决办法2:

```
/test.py - Sublime Text
test.py

1 #coding=utf-8
2
3 def calNum(num):
4     if num>=1:
5         result = num * calNum(num-1)
6     else:
7         result = 1
8     return result
9
10 # def calNum(num):
11 #     i = 1
12 #     result = 1
13 #
14 #     while i<=num:
15 #         result *= i
16 #         i+=1
17 #
18 #     return result
19
20 ret = calNum(3)
21 print(ret)
```

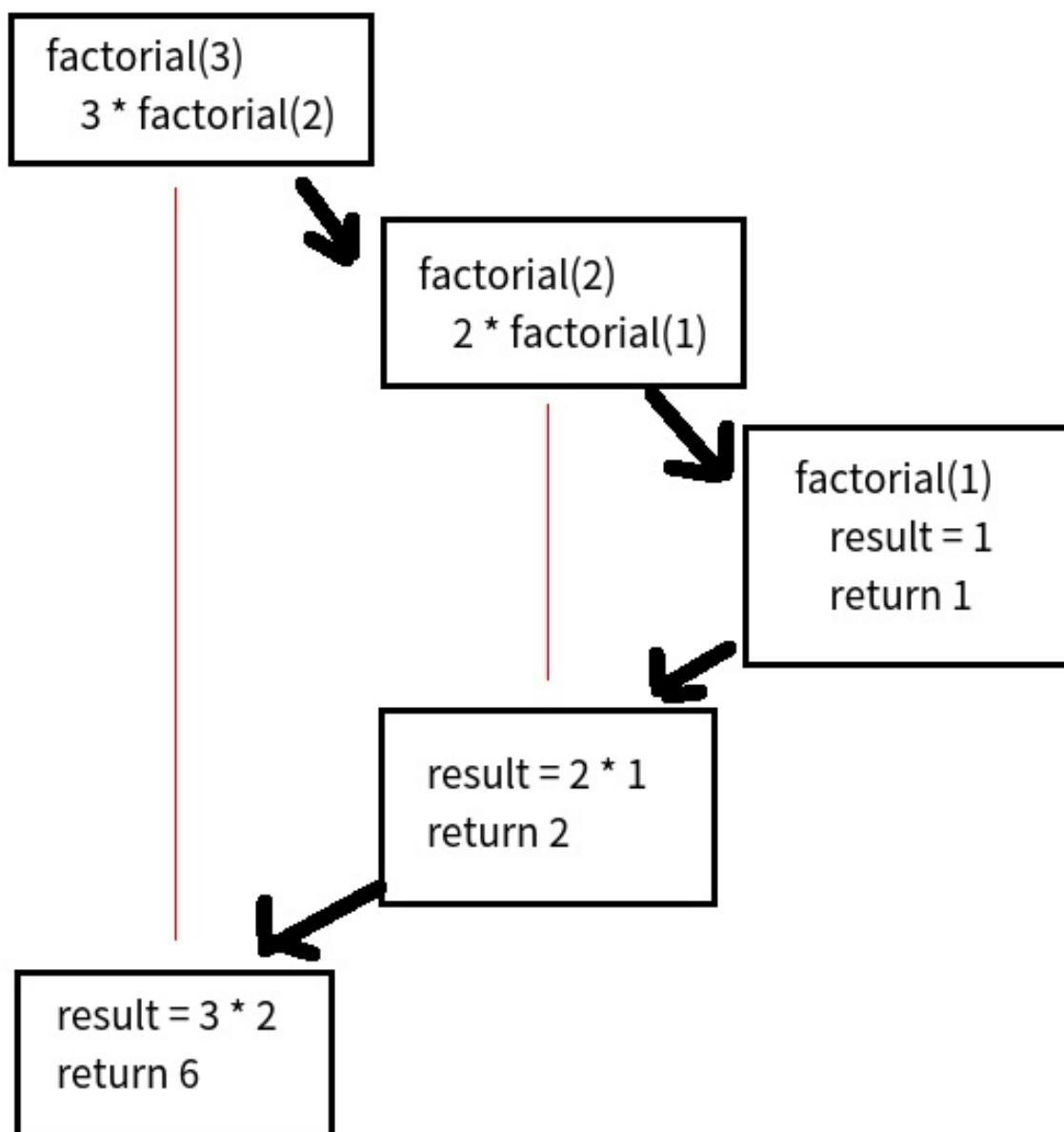
使用递归来完成

```
python@ubuntu: ~/Desktop
python@ubuntu:~/Desktop$ python3 test.py
6
python@ubuntu:~/Desktop$
```

## 原理

```
def factorial(num):  
    if num>1:  
        result = num * factorial(num-1)  
    else:  
        result = 1  
    return result
```

factorial(3)调用过程：





# 匿名函数

用lambda关键词能创建小型匿名函数。这种函数得名于省略了用def声明函数的标准步骤。

lambda函数的语法只包含一个语句，如下：

```
lambda [arg1 [,arg2,.....argn]]:expression
      参数           表达式
```

如下实例：

```
sum = lambda arg1, arg2: arg1 + arg2
#调用sum函数
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

以上实例输出结果：

```
Value of total : 30
Value of total : 40
```

Lambda函数能接收任何数量的参数但只能返回一个表达式的值

匿名函数不能直接调用print，因为lambda需要一个表达式

## 应用场合

### 函数作为参数传递

#### 1. 自己定义函数

```
>>> def fun(a, b, opt):
```

```

...
    print "a =", a
...
    print "b =", b
...
    print "result =", opt(a, b)
...
>>> fun(1, 2, lambda x,y:x+y)
a = 1
b = 2
result = 3

```

```

1 #coding=utf-8
2
3 def test(a,b,func):
4     result = func(a,b)
5     print(result)
6
7
8 func_new = input("请输入一个匿名函数:")
9 func_new = eval(func_new)
10
11 test(11,22,func_new)

```

## 2. 作为**内置函数**的参数

想一想，下面的数据如何指定按age或name排序？

```

stus = [
    {"name": "zhangsan", "age": 18},
    {"name": "lisi", "age": 19},
    {"name": "wangwu", "age": 17}
]

```

**按name排序：**

```

>>> stus.sort(key = lambda x:x['name'])
>>> stus
[{'age': 19, 'name': 'lisi'}, {'age': 17, 'name': 'wangwu'}, {'a

```

**按age排序：**

```

>>> stus.sort(key = lambda x:x['age'])
>>> stus
[{'age': 17, 'name': 'wangwu'}, {'age': 18, 'name': 'zhangsan'},

```

```
on 1276*530 ~/Desktop/python05期/python语法-06
python@ubuntu: ~/Desktop/python05期/python语法

1 #a = 100
2 a = [100]
3
4 def test(num):
5     num+=num
6     print(num)
7
8
9 test(a)
10
11 print(a)
```

调用函数的时候使用了成员变量需要问一问是不是可变函数,若不是那么在函数中及时重新创造了数据,若是可变那就是真的改变了

```
python@ub: ~
1 a = 4
2 b = 5
3
4 #交换方式1
5 c = 0
6 c = a
7 a = b
8 b = c
9
10 #方法2
11 a = a+b
12 b = a-b
13 a = a-b
14
15 #方法3
16 a,b = b,a
```

# 函数使用注意事项

## 1. 自定义函数

### <1>无参数、无返回值

```
def 函数名():
    语句
```

### <2>无参数、有返回值

```
def 函数名():
    语句
    return 需要返回的数值
```

注意：

- 一个函数到底有没有返回值，就看有没有return，因为只有return才可以返回数据
- 在开发中往往根据需求来设计函数需不需要返回值
- 函数中，可以有多个return语句，但是只要执行到一个return语句，那么就意味着这个函数的调用完成

### <3>有参数、无返回值

```
def 函数名(形参列表):
    语句
```

注意：

- 在调用函数时，如果需要把一些数据一起传递过去，被调用函数就需要用参数来接收

- 参数列表中变量的个数根据实际传递的数据的多少来确定

#### <4>有参数、有返回值

```
def 函数名(形参列表):  
    语句  
    return 需要返回的数值
```

#### <5>函数名不能重复

```
test.py  
1 #coding=utf-8  
2  
3 def test():  
4     print("你好啊")  
5  
6 def test(a,b):  
7     print("你好啊ab")  
8  
9 test()  
10
```

```
dongGe@bogon Desktop$ python test.py  
Traceback (most recent call last):  
  File "test.py", line 9, in <module>  
    test()  
TypeError: test() takes exactly 2 arguments (0 given)  
dongGe@bogon Desktop$
```

函数的名字不要相同

## 2. 调用函数

#### <1>调用的方式为：

```
函数名([实参列表])
```

#### <2>调用时，到底写不写 实参

- 如果调用的函数 在定义时有形参，那么在调用的时候就应该传递参数

#### <3>调用时，实参的个数和先后顺序应该和定义函数中要求的一致

#### <4>如果调用的函数有返回值，那么就可以用一个变量来进行保存这个值

### 3. 作用域

<1>在一个函数中定义的变量，只能在本函数中用(局部变量)

```
test.py
1 #coding=utf-8
2
3 def test():
4     a=100
5     print("----1----a=%d"%a)
6
7 def test2():
8     print("----2----a=%d"%a)
9
10
11 test()
12
13 test2()
```

```
[dongGe@bogon Desktop]$ python test.py
----1----a=100
Traceback (most recent call last):
  File "test.py", line 13, in <module>
    test2()
  File "test.py", line 8, in test2
    print("----2----a=%d"%a)
NameError: global name 'a' is not defined
[dongGe@bogon Desktop]$
```

<2>在函数外定义的变量，可以在所有的函数中使用(全局变量)

# 作业

## 必做题

### 1. 编程实现 9\*9乘法表

提示：使用循环嵌套

### 2.用函数实现求100-200里面所有的素数

提示：素数的特征是除了1和其本身能被整除，其它数都不能被整除的数

### 3.请用函数实现一个判断用户输入的年份是否是闰年的程序

提示：

1. 能被400整除的年份
  2. 能被4整除，但是不能被100整除的年份
- 以上2种方法满足一种即为闰年

## 选做题

### 1.用函数实现输入某年某月某日，判断这一天是这一年的第几天？闰年情况也考虑进去

20160818  
是今年第x天

## 2. 编写“学生管理系统”，要求如下：

必须使用自定义函数，完成对程序的模块化

学生信息至少包含：姓名、年龄、学号，除此以外可以适当添加

必须完成的功能：添加、删除、修改、查询、退出



# 文件操作介绍

## <1>什么是文件

示例如下：



## <2>文件的作用

大家应该听说过一句话：“好记性不如烂笔头”。

不仅人的大脑会遗忘事情，计算机也会如此，比如一个程序在运行过程中用了九牛二虎之力终于计算出了结果，试想一下如果不把这些数据存放起来，相比重启电脑之后，“哭都没地方哭了”

可见，在把数据存储起来有做么大的价值

使用文件的目的：

就是把一些存储存放起来，可以让程序下一次执行的时候直接使用，而不必重新制作一份，省时省力

# 文件的打开与关闭

想一想：

如果想用word编写一份简历，应该有哪些流程呢？

1. 打开word软件，新建一个word文件
2. 写入个人简历信息
3. 保存文件
4. 关闭word软件

同样，在操作文件的整体过程与使用word编写一份简历的过程是很相似的

1. 打开文件，或者新建立一个文件
2. 读/写数据
3. 关闭文件

## <1>打开文件

在python，使用open函数，可以打开一个已经存在的文件，或者创建一个新文件

open(文件名，访问模式)

示例如下：

```
f = open('test.txt', 'w')
```

说明：

访问模式	说明

	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。
>	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。
>>	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。
图片, mp3 等	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。
	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。
	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。
	r+	打开一个文件用于 <u>读写</u> 。文件指针将会放在文件的开头。
	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。
	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。
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。
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。
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。

## <2>关闭文件

`close()`

示例如下：

```
# 新建一个文件，文件名为:test.txt
f = open('test.txt', 'w')

# 关闭这个文件
f.close()
```

# 文件的读写

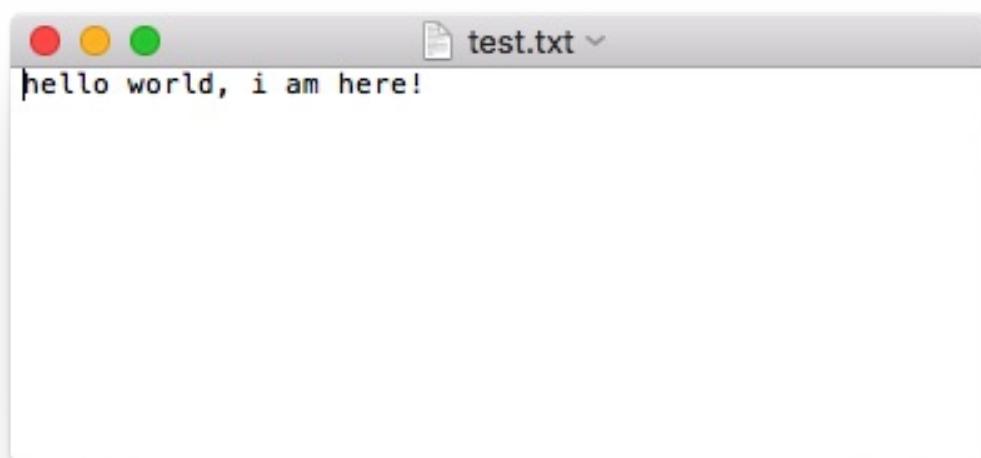
## <1>写数据(write)

使用write()可以完成向文件写入数据

demo:

```
f = open('test.txt', 'w')
f.write('hello world, i am here!')
f.close()
```

运行现象:



注意:

- 如果文件不存在那么创建, 如果存在那么就先清空, 然后写入数据

## <2>读数据(read)

使用read(num)可以从文件中读取数据，num表示要从文件中读取的数据的长度（单位是字节），如果没有传入num，那么就表示读取文件中所有的数据

demo:

```
f = open('test.txt', 'r')

content = f.read(5)

print(content)

print("-"*30)

content = f.read()

print(content)

f.close()
```

运行现象：

```
[dongGe@dongGe-Mac Desktop$ python filetest.py
hello
-----
world, i am here!
```

注意：

- 如果open是打开一个文件，那么可以不用~~w~~打开的模式，即只写  
open('test.txt')
- 如果使用读了多次，那么后面读取的数据是从上次读完后的位置开始的

## <3>读数据 (readlines)

就像read没有参数时一样，readlines可以按照行的方式把整个文件中的内容进行一次性读取，并且返回的是一个列表，其中每一行的数据为一个元素

```
#coding=utf-8

f = open('test.txt', 'r')

content = f.readlines()

print(type(content))

i=1
for temp in content:
    print("%d:%s"%(i, temp))
    i+=1

f.close()
```

运行现象：

```
[dongGe@dongGe-Mac Desktop$ python filetest.py
<type 'list'>
1:hello world, i am here!
2:hello world, i am here!
3:hello world, i am here!
4:hello world, i am here!
5:hello world, i am here!
dongGe@dongGe-Mac Desktop$ ]
```

## <4>读数据 (readline)

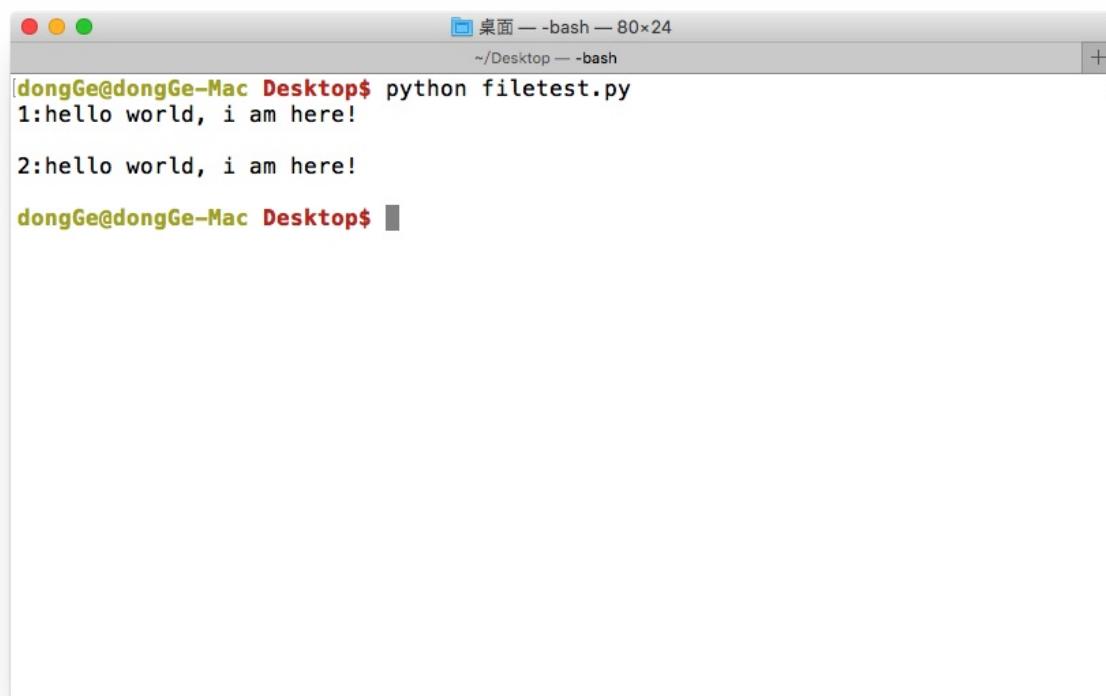
```
#coding=utf-8

f = open('test.txt', 'r')

content = f.readline()
print("1:%s"%content)

content = f.readline()
print("2:%s"%content)

f.close()
```



The screenshot shows a terminal window titled '桌面 — bash — 80x24' with the command `~/Desktop — bash`. The user has run the script `python filetest.py`, which outputs two lines of text: '1:hello world, i am here!' and '2:hello world, i am here!'. The terminal window has a standard OS X look with red, yellow, and green close buttons.

想一想：

如果一个文件很大，比如5G，试想应该怎样把文件的数据读取到内存然后进行处理呢？

# 应用1:制作文件的备份

## 任务描述

- 输入文件的名字，然后程序自动完成对文件进行备份



## 参考代码

```
#coding=utf-8

oldFileName = input("请输入要拷贝的文件名字:")

oldFile = open(oldFileName, 'r')

# 如果打开文件
if oldFile:

    # 提取文件的后缀
    fileFlagNum = oldFileName.rfind('.')
    if fileFlagNum > 0:
        fileFlag = oldFileName[fileFlagNum:]

    # 组织新的文件名字
    newFileName = oldFileName[:fileFlagNum] + '[复印件]' + fileFlag

    # 创建新文件
    newFile = open(newFileName, 'w')

    # 把旧文件中的数据，一行一行的进行复制到新文件中
    for lineContent in oldFile.readlines():
        newFile.write(lineContent)

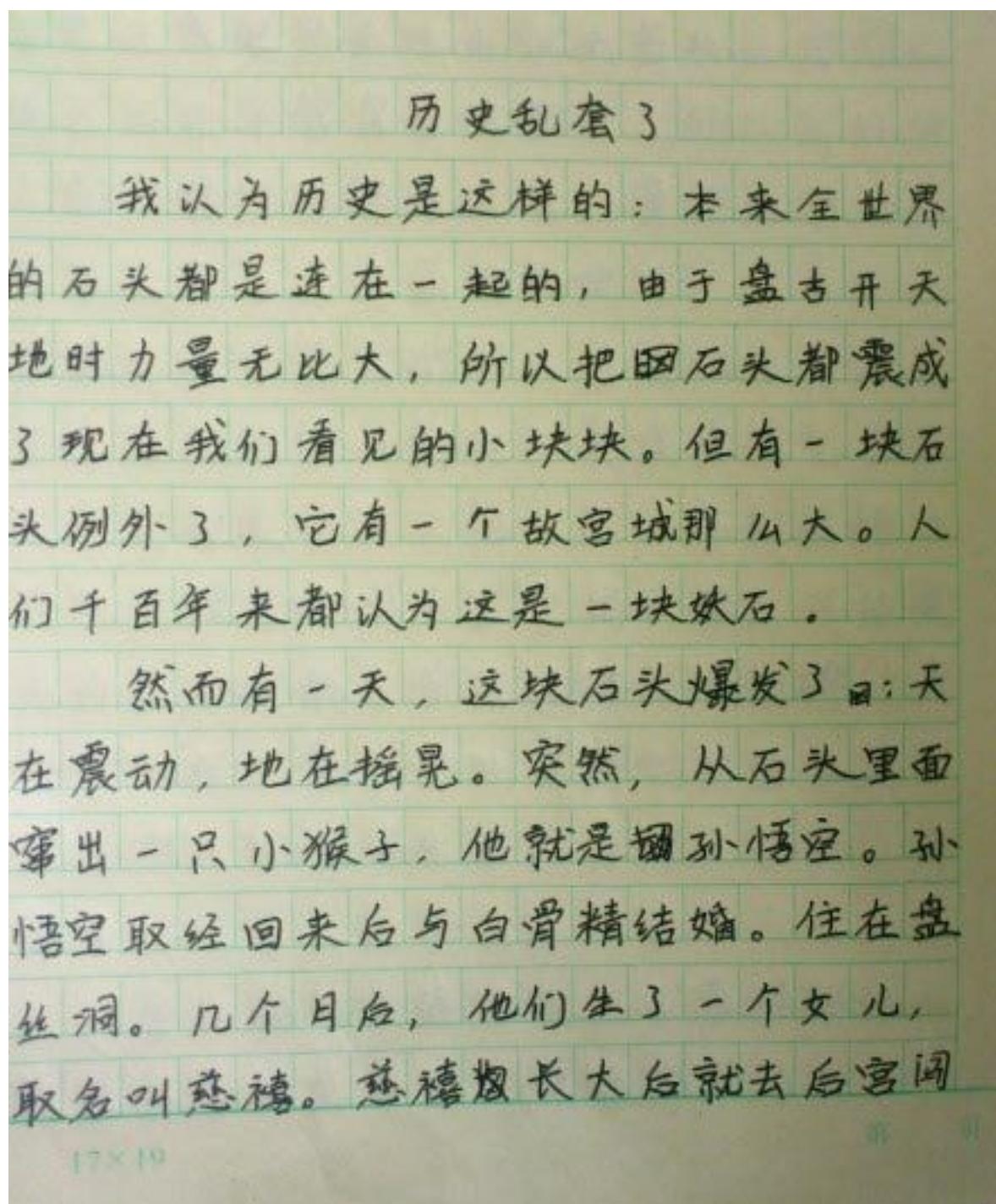
    # 关闭文件
    oldFile.close()
    newFile.close()
```

Linux不管后缀,但是windows不行啊

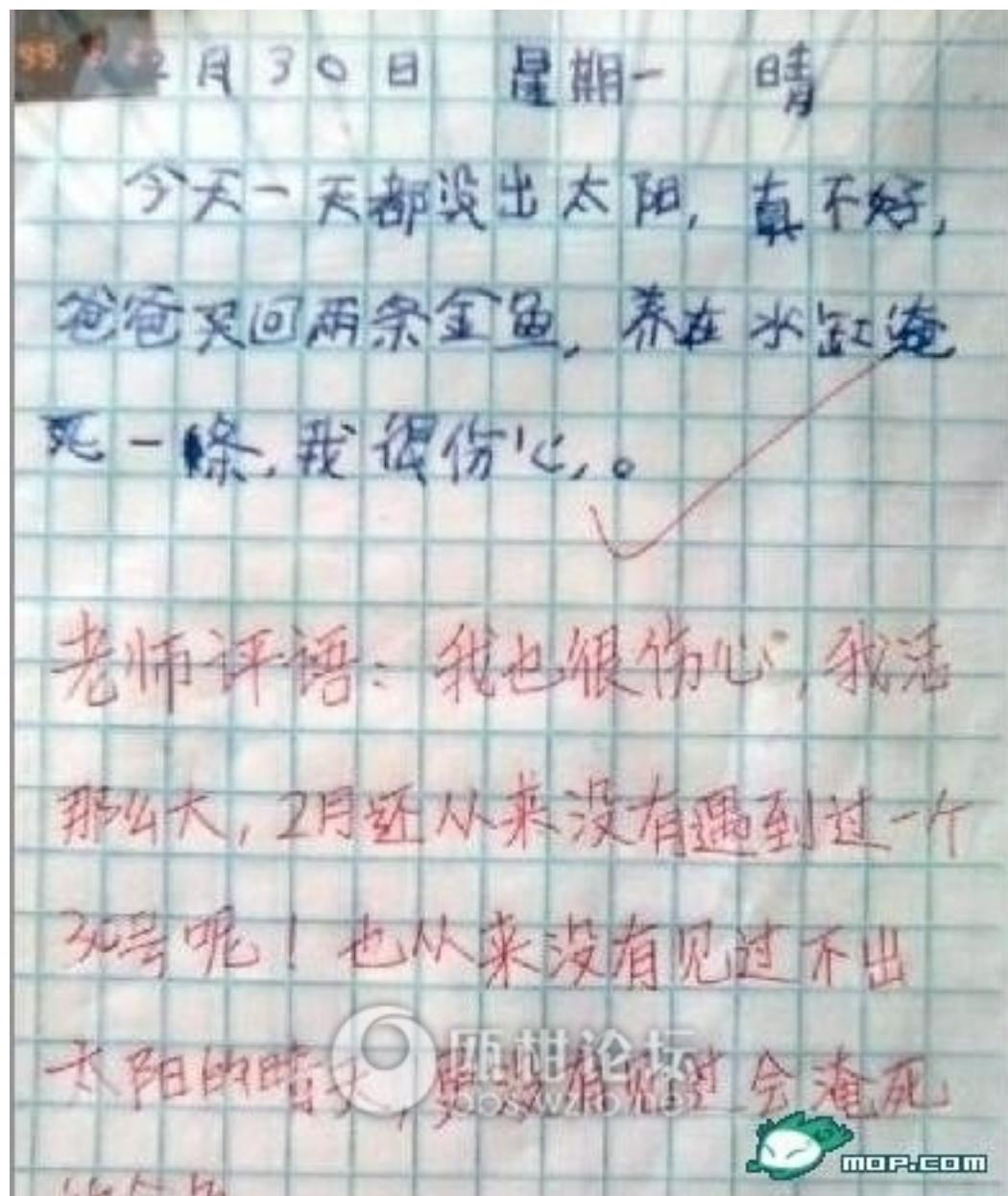
大文件使用readline()但是也有漏洞你知道一行有多大么,所以应该是指定大小使用read(1024)在使用while True循环当在读到的内容为空的时候进行break即可

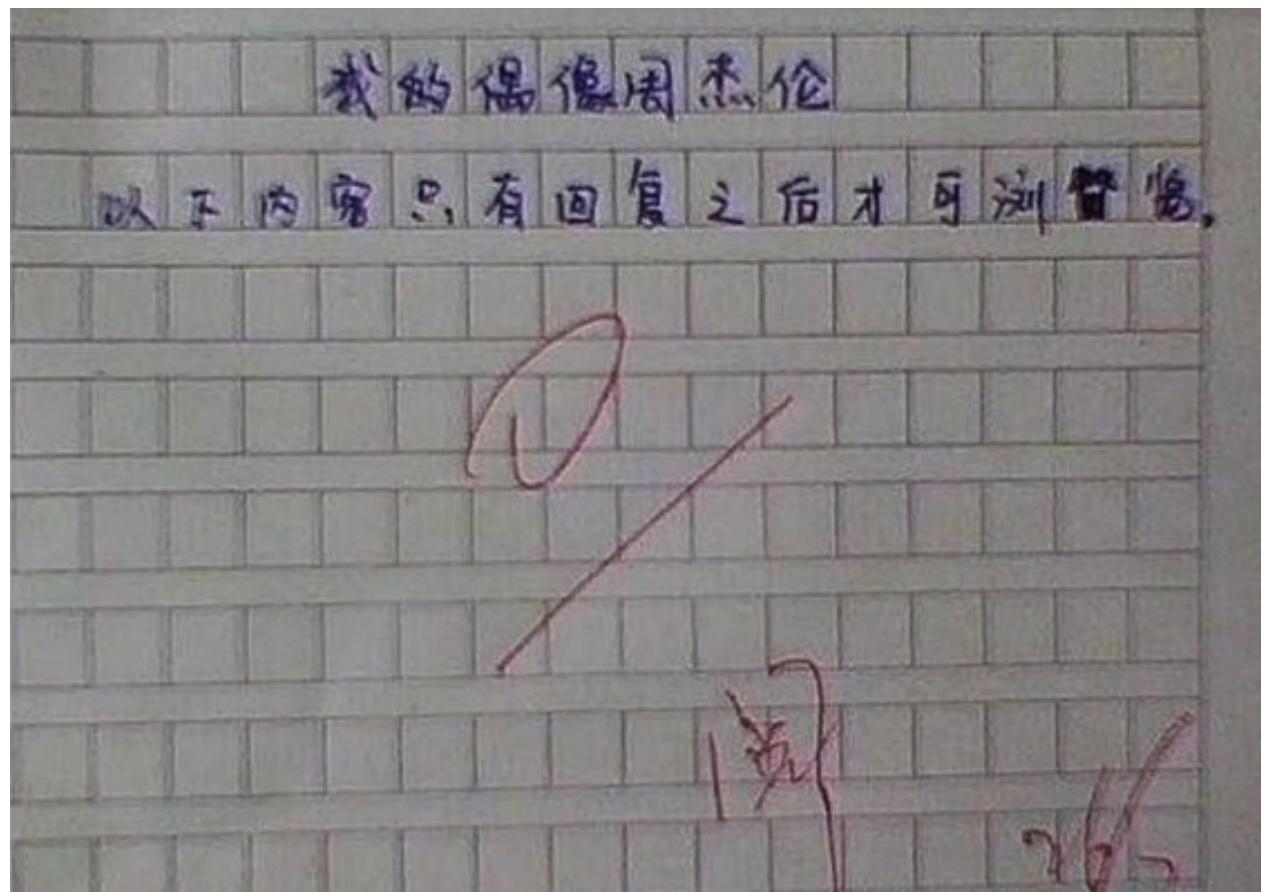
## 文件的随机读写

什么是定位？



17×10





## <1>获取当前读写的位置

在读写文件的过程中，如果想知道当前的位置，可以使用tell()来获取

```
# 打开一个已经存在的文件
f = open("test.txt", "r")
str = f.read(3)
print "读取的数据是 : ", str

# 查找当前位置
position = f.tell()
print "当前文件位置 : ", position

str = f.read(3)
print "读取的数据是 : ", str

# 查找当前位置
```

```

position = f.tell()
print "当前文件位置 : ", position

f.close()

```

## <2>定位到某个位置

如果在读写文件的过程中，需要从另外一个位置进行操作的话，可以使用 `seek()`

`seek(offset, from)`有2个参数

- `offset`:偏移量
- `from`:方向
  - 0:表示文件开头
  - 1:表示当前位置
  - 2:表示文件末尾

demo:把位置设置为：从文件开头，偏移5个字节

```

# 打开一个已经存在的文件
f = open("test.txt", "r")
str = f.read(30)
print "读取的数据是 : ", str

# 查找当前位置
position = f.tell()
print "当前文件位置 : ", position

# 重新设置位置
f.seek(5, 0)

# 查找当前位置
position = f.tell()
print "当前文件位置 : ", position

```

```
f.close()
```

demo:把位置设置为：离文件末尾， 3字节处

```
# 打开一个已经存在的文件
f = open("test.txt", "r")

# 查找当前位置
position = f.tell()
print "当前文件位置 : ", position

# 重新设置位置
f.seek(-3, 2)

# 读取到的数据为：文件最后3个字节数据
str = f.read()
print "读取的数据是 : ", str

f.close()
```

# 文件的重命名、删除

有些时候，需要对文件进行重命名、删除等一些操作，python的os模块中都有这么功能

## <1>文件重命名

os模块中的rename()可以完成对文件的重命名操作

rename(需要修改的文件名, 新的文件名)

```
import os  
  
os.rename("毕业论文.txt", "毕业论文-最终版.txt")
```

## <2>删除文件

os模块中的remove()可以完成对文件的删除操作

remove(待删除的文件名)

```
import os  
  
os.remove("毕业论文.txt")
```

# 文件夹的相关操作

实际开发中，有时需要用程序的方式对文件夹进行一定的操作，比如创建、删除等

就像对文件操作需要os模块一样，如果要操作文件夹，同样需要os模块

## <1>创建文件夹

```
import os  
  
os.mkdir("张三")
```

## <2>获取当前目录

```
import os  
  
os.getcwd()
```

## <3>改变默认目录

```
import os  
  
os.chdir("../")
```

## <4>获取目录列表

```
import os  
  
os.listdir("./")
```

## <5>删除文件夹

```
import os  
  
os.rmdir("张三")
```

# 应用：批量修改文件名

## <1>运行过程演示

- 运行程序之前

```
[dongGe@localhost renameDir$ ls -l
total 0
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-1.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-2.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-3.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-4.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-5.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-6.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-7.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-8.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 新三国-9.txt]
```

- 运行程序之后

```
[dongGe@localhost renameDir$ ls -l
total 0
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-1.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-2.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-3.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-4.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-5.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-6.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-7.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-8.txt
-rw-r--r-- 1 dongGe staff 0 6 13 17:33 [东哥出品]-新三国-9.txt]
```

## <2>参考代码

```
#coding=utf-8

# 批量在文件名前加前缀

import os

funFlag = 1 # 1表示添加标志 2表示删除标志
```

```
folderName = './renameDir/'\n\n# 获取指定路径的所有文件名字\ndirList = os.listdir(folderName)\n\n# 遍历输出所有文件名字\nfor name in dirList:\n    print name\n\n    if funFlag == 1:\n        newName = '[东哥出品]-' + name\n    elif funFlag == 2:\n        num = len('[东哥出品]-')\n        newName = name[num:]\n    print newName\n\nos.rename(folderName+name, folderName+newName)
```

```
1 import os\n2\n3 #1. 获取一个要重命名的文件夹的名字\n4 folder_name = input("请输入要重命名的文件夹:")\n5\n6 #2. 获取那个文件夹中所有的文件名字\n7 file_names = os.listdir(folder_name)\n8\n9 #第1中方法\n10 #os.chdir(folder_name)\n11\n12 #3. 对获取的名字进行重命名即可\n13 #for name in file_names:\n14 #    print(name)\n15 #    os.rename(name, "[京东出品]-"+name)\n16\n17\n18\n19 for name in file_names:\n20     #print(name)\n21     old_file_name = "./"+ folder_name + "/" +name\n22     new_file_name = "./"+folder_name+"/"+"[京东出品]-"+name\n23     os.rename(old_file_name, new_file_name)
```

## 综合应用:学生管理系统(文件版)

## 作业

1. 读取一个文件，显示除了以井号(#)开头的行以外的所有行
  
2. 制作一个“密码薄”，其可以存储一个网址（例如 [www.itcast.cn](http://www.itcast.cn)），和一个密码（例如 123456），请编写程序完成这个“密码薄”的增删改查功能，并且实现文件存储功能



# 面向对象编程介绍

## 想一想

请用程序描述如下事情：

- A同学报道登记信息
- B同学报道登记信息
- C同学报道登记信息
- A同学做自我介绍
- B同学做自我介绍
- C同学做自我介绍

```
stu_a = {  
    "name": "A",  
    "age": 21,  
    "gender": 1,  
    "hometown": "河北"  
}  
stu_b = {  
    "name": "B",  
    "age": 22,  
    "gender": 0,  
    "hometown": "山东"  
}  
stu_c = {  
    "name": "C",  
    "age": 20,  
    "gender": 1,  
    "hometown": "安徽"  
}  
def stu_intro(stu):  
    """自我介绍"""  
    for key, value in stu.items():
```

```
print("key=%s, value=%d"%(key,value))

stu_intro(stu_a)
stu_intro(stu_b)
stu_intro(stu_c)
```

考虑现实生活中，我们的思维方式是放在学生这个个人上，是学生做了自我介绍。而不是像我们刚刚写出的代码，先有了介绍的行为，再去看介绍了谁。

用我们的现实思维方式该怎么用程序表达呢？

```
stu_a = Student(个人信息)
stu_b = Student(个人信息)
stu_c = Student(个人信息)
stu_a.intro()
stu_a.intro()
stu_a.intro()
```

- 面向过程：根据业务逻辑从上到下写代码
- 面向对象：将数据与函数绑定到一起，进行封装，这样能够更快速的开发程序，减少了重复代码的重写过程

面向过程编程最易被初学者接受，其往往用一段长代码来实现指定功能，开发过程的思路是将数据与函数按照执行的逻辑顺序组织在一起，数据与函数分开考虑。

```
def 发送邮件(内容)
    #发送邮件提醒
    连接邮箱服务器
    发送邮件
    关闭连接

while True:
```

```
if cpu利用率 > 90%:  
    发送邮件('CPU报警')  
  
if 硬盘使用空间 > 90%:  
    发送邮件('硬盘报警')  
  
if 内存占用 > 80%:  
    发送邮件('内存报警')
```

今天我们来学习一种新的编程方式：面向对象编程（Object Oriented Programming, OOP，面向对象程序设计）

- 1) 解决菜鸟买电脑的故事

第一种方式：

- 1)在网上查找资料
- 2)根据自己预算和需求定电脑的型号 MacBook 15 顶配 1W8
- 3)去市场找到苹果店各种店无法甄别真假 随便找了一家
- 4)找到业务员,业务员推荐了另外一款 配置更高价格便宜,也是苹果系统的 1W
- 5)砍价30分钟 付款9999
- 6)成交

回去之后发现各种问题

第二种方式：

- 1)找一个靠谱的电脑高手
- 2)给钱交易

- 面向对象和面向过程都是解决问题的一种思路而已
  - 买电脑的第一种方式：

- 强调的是步骤、过程、每一步都是自己亲自去实现的
  - 这种解决问题的思路我们就叫做面向过程
  - 买电脑的第二种方式：
    - 强调的是电脑高手，电脑高手是处理这件事的主角，对我们而言，我们并不必亲自实现整个步骤只需要调用电脑高手就可以解决问题
    - 这种解决问题的思路就是面向对象
  - 用面向对象的思维解决问题的重点
    - 当遇到一个需求的时候不用自己去实现，如果自己一步步实现那就是面向过程
    - 应该找一个专门做这个事的人来做
    - 面向对象是基于面向过程的
- 2) 解决吃啤酒鸭的问题

第一种方式（面向过程）：

1)养鸭子

2)鸭子长成

3)杀

4)作料

5)烹饪

6)吃

7)卒

第二种方式（面向对象）：

1)找个卖啤酒鸭的人

2)给钱交易

3)吃

#### 4)胖6斤

需要了解的定义性文字:

面向对象(object-oriented ;简称: OO) 至今还没有统一的概念 我这里把它定义为: 按人们 认识客观世界的系统思维方式,采用基于对象(实体) 的概念建立模型,模拟客观世界分析、设计、实现软件的办法。

面向对象编程(Object Oriented Programming-OOP) 是一种解决软件复用的设计和编程方法。 这种方法把软件系统中相近相似的操作逻辑和操作 应用数据、状态,以类的型式描述出来,以对象实例的形式在软件系统中复用,以达到提高软件开发效率的作用。

# 类和对象

面向对象编程的2个非常重要的概念：**类和对象**

对象是面向对象编程的核心，在使用对象的过程中，为了将具有共同特征和行为的一组对象抽象定义，提出了另外一个新的概念——类

类就相当于制造飞机时的图纸，用它来进行创建的飞机就相当于对象

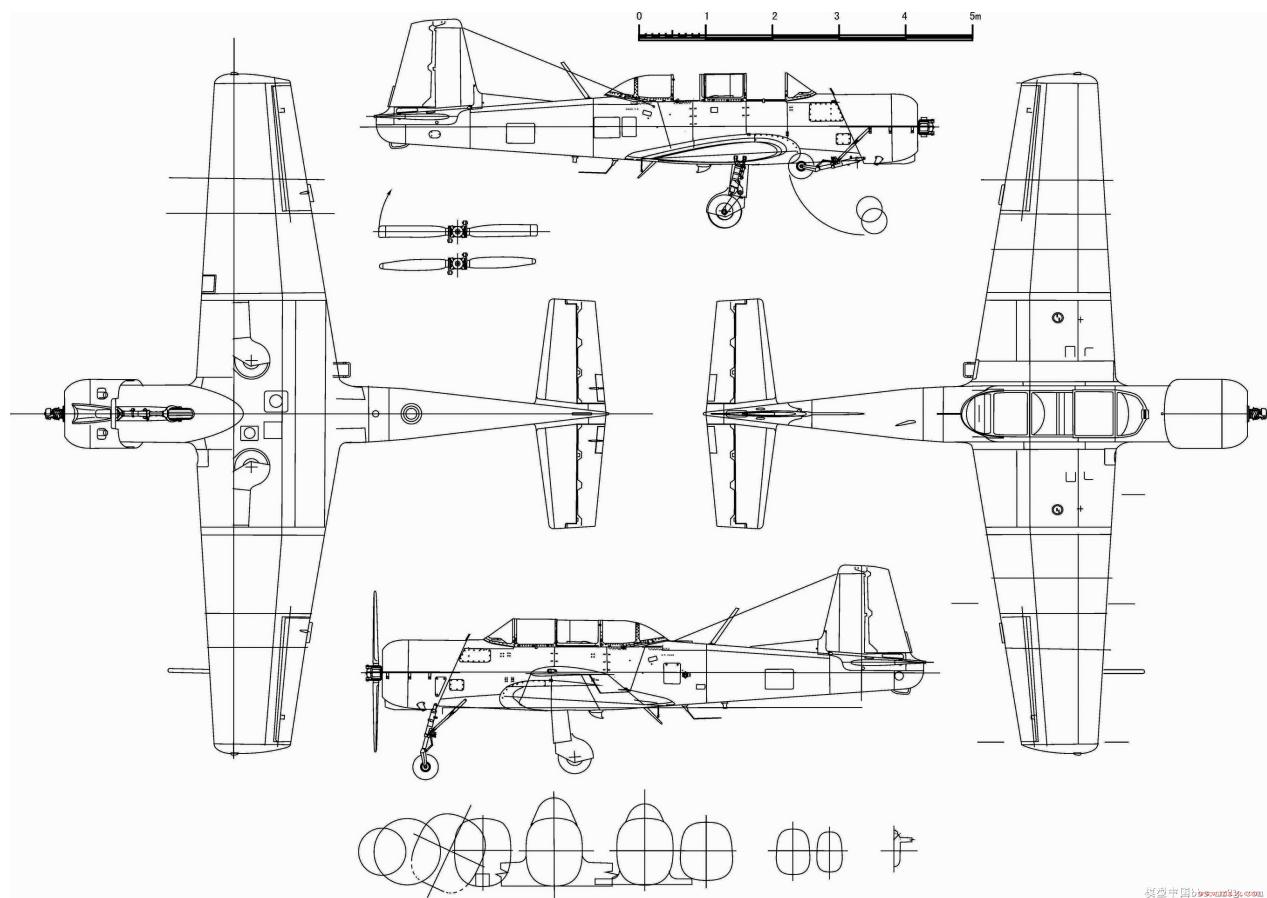
## 1. 类

人以类聚 物以群分。

具有相似内部状态和运动规律的实体的集合(或统称为抽象)。

具有相同属性和行为事物的统称

类是抽象的，在使用的时候通常会找到这个类的一个具体的存在，使用这个具体的存在。一个类可以找到多个对象



模型中国 <http://www.model3d.com>

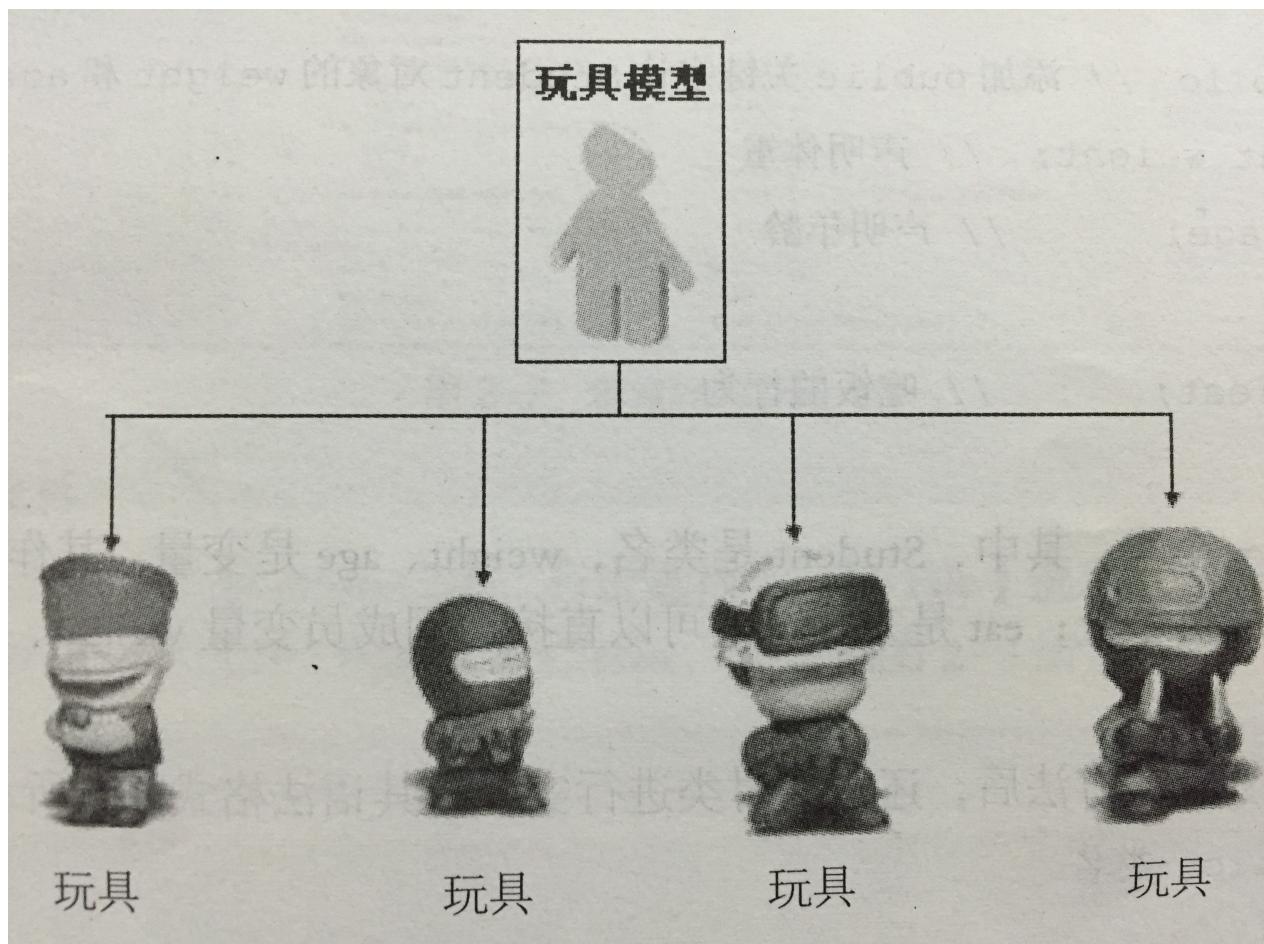
## 2. 对象

某一个具体事物的存在，在现实世界中可以是看得见摸得着的。

可以是直接使用的



### 3. 类和对象之间的关系



小总结：类就是创建对象的模板

## 4. 练习：区分类和对象

奔驰汽车 类  
奔驰smart 类  
张三的那辆奔驰smart 对象  
狗 类  
大黄狗 类  
李四家那只大黄狗 对象  
水果 类  
苹果 类  
红苹果 类 红富士苹果 类  
我嘴里吃了一半的苹果 对象

## 5. 类的构成

类(Class) 由3个部分构成

- **类的名称**:类名
- **类的属性**:一组数据
- **类的方法**:允许对进行操作的方法 (行为)

### <1> 举例：

1) 人类设计,只关心3样东西:

- 事物名称(类名):人(Person)
- 属性:身高(height)、年龄(age)
- 方法(行为/功能):跑(run)、打架(fight)

2) 狗类的设计

- 类名:狗(Dog)
- 属性:品种、毛色、性别、名字、腿儿的数量

- 方法(行为/功能):叫、跑、咬人、吃、摇尾巴



## 6. 类的抽象

如何把日常生活中的事物抽象成程序中的类?

拥有相同(或者类似)属性和行为的对象都可以抽象出一个类

方法:一般名词都是类(名词提炼法)

### <1> 坦克发射3颗炮弹轰掉了2架飞机

- 坦克-->可以抽象成类
- 炮弹-->可以抽象成类
- 飞机-->可以抽象成类

### <2> 小明在公车上牵着一条叼着热狗的狗

- 小明-->人类
- 公车-->交通工具类
- 热狗-->食物类
- 狗-->狗类

<3> 【想一想】如下图中，有哪些类呢？



说明：

- 人
- 枪
- 子弹
- 手榴弹
- 刀子
- 箱子

<4> 【想一想】如下图中，有哪些类呢？



说明:

- 向日葵
  - 类名: xrk
  - 属性:
  - 行为: 放阳光
- 豌豆
  - 类名: wd
  - 属性: 颜色、发型, 血量
  - 行为: 发炮, 摆头
- 坚果:
  - 类名:jg
  - 属性: 血量, 类型
  - 行为: 阻挡;
- 僵尸:
  - 类名:js
  - 属性: 颜色、血量、类型、速度
  - 行为: 走跑跳吃死

# 定义类

定义一个类，格式如下：

```
class 类名:  
    方法列表
```

demo： 定义一个Car类

```
# 定义类  
class Car:  
    # 方法  
    def getCarInfo(self):  
        print('车轮子个数：%d, 颜色%s'%(self.wheelNum, self.color))  
  
    def move(self):  
        print("车正在移动...")
```

## 说明：

- 定义类时有2种：新式类和经典类，上面的Car为经典类，如果是Car(object)则为新式类
- 类名的命名规则按照“大驼峰”

# 创建对象

通过上一节课程，定义了一个Car类；就好比有车一个张图纸，那么接下来就应该把图纸交给生成工人们去生成了

python中，可以根据已经定义的类去创建出一个个对象

创建对象的格式为：

```
对象名 = 类名()
```

创建对象demo：

```
# 定义类
class Car:
    # 移动
    def move(self):
        print('车在奔跑...')

    # 鸣笛
    def toot(self):
        print("车在鸣笛...嘟嘟...")

# 创建一个对象，并用变量BMW来保存它的引用
BMW = Car()
BMW.color = '黑色'
BMW.wheelNum = 4 #轮子数量
BMW.move()
BMW.toot()
print(BMW.color)
print(BMW.wheelNum)
```

```

1 class Car: ← 告诉python
2                                     我们在这里定
3     # 移动                         义了一个类
4     def move(self):
5         print('车在奔跑 ...')
6
7     # 鸣笛
8     def toot(self):
9         print("车在鸣笛 ...嘟嘟...")
10
11
12 # 创建一个对象，并用变量BMW来保存它的引用
13 BMW = Car()
14 BMW.color = '黑色' ← 给对象添加 属性
15 BMW.oil = 30 #油量为30升
16 BMW.move() ← 调用对象的方法
17 BMW.toot()
18 print(BMW.color)
19 print(BMW.oil)
20

```

## 总结：

- BMW = Car(), 这样就产生了一个Car的实例对象，此时也可以通过实例对象BMW来访问属性或者方法
- 第一次使用BMW.color = '黑色'表示给BMW这个对象添加属性，如果后面再次出现BMW.color = xxx表示对属性进行修改
- BMW是一个对象，它拥有属性（数据）和方法（函数）
- 当创建一个对象时，就是用一个模子，来制造一个实物



## \_\_init\_\_() 方法

想一想：

在上一小节的demo中，我们已经给BMW这个对象添加了2个属性，wheelNum（车的轮胎数量）以及color（车的颜色），试想如果再次创建一个对象的话，肯定也需要进行添加属性，显然这样做很费事，那么有没有办法能够在创建对象的时候，就顺便把车这个对象的属性给设置呢？

答：

\_\_init\_\_() 方法

### <1> 使用方式

```
def 类名:  
    # 初始化函数，用来完成一些默认的设定  
    def __init__():  
        pass
```

### <2> \_\_init\_\_() 方法的调用

```
# 定义汽车类  
class Car:  
  
    def __init__(self):  
        self.wheelNum = 4  
        self.color = '蓝色'  
  
    def move(self):  
        print('车在跑，目标：夏威夷')
```

```
# 创建对象
BMW = Car()

print('车的颜色为:%s' %BMW.color)
print('车轮胎数量为:%d' %BMW.wheelNum)
```



```
[dongGe@bogon Desktop$ python oop.py
车的颜色为:blue
车轮胎数量为:4
dongGe@bogon Desktop$ ]
```

## 总结1

当创建Car对象后，在没有调用 `__init__()` 方法的前提下，BMW就默认拥有了2个属性`wheelNum`和`color`，原因是 `__init__()` 方法是在创建对象后，就立刻被默认调用了

## 想一想

既然在创建完对象后 `__init__()` 方法已经被默认的执行了，那么能否让对象在调用 `__init__()` 方法的时候传递一些参数呢？如果可以，那怎样传递呢？

```
# 定义汽车类
class Car:

    def __init__(self, newWheelNum, newColor):
```

```
        self.wheelNum = newWheelNum  
        self.color = newColor  
  
    def move(self):  
        print('车在跑, 目标:夏威夷')  
  
# 创建对象  
BMW = Car(4, 'green')  
  
print('车的颜色为:%s' %BMW.color)  
print('车轮子数量为:%d' %BMW.wheelNum)
```



## 总结2

- `__init__()` 方法, 在创建一个对象时默认被调用, 不需要手动调用
- `__init__(self)` 中, 默认有1个参数名字为self, 如果在创建对象时传递了2个实参, 那么 `__init__(self)` 中除了self作为第一个形参外还需要2个形参, 例如 `__init__(self, x, y)`
- `__init__(self)` 中的self参数, 不需要开发者传递, python解释器会自动把当前的对象引用传递进去

## 应用:创建多个对象

- 根据上两节创建一个Car类
- 创建出多个汽车对象，比如BMW、AUDI等

# "魔法"方法

## 1. 打印id()

如果把BMW使用print进行输出的话，会看到如下的信息

```
[python@ubuntu:~/Desktop/test$ python3 oop.py
车在奔跑...
车在鸣笛...嘟嘟..
黑色
30
-----分割线-----
<__main__.Car object at 0x7ff4da7a7940>
python@ubuntu:~/Desktop/test$ ]
```

即看到的是创建出来的BMW对象在内存中的地址

## 2. 定义 \_\_str\_\_() 方法

```
class Car:

    def __init__(self, newWheelNum, newColor):
        self.wheelNum = newWheelNum
        self.color = newColor

    def __str__(self):
        msg = "嘿。。。我的颜色是" + self.color + "我有" + int(self
        return msg 类似于toString(), 所以一定有返回值

    def move(self):
        print('车在跑, 目标:夏威夷')

BMW = Car(4, "白色")
print(BMW)
```

```
[python@ubuntu:~/Desktop/test$ python3 oop.py  
嘿。。。我的颜色是白色我有4个轮胎...  
python@ubuntu:~/Desktop/test$ ]
```

## 总结

- 在python中方法名如果是 `__xxxx__()` 的，那么就有特殊的功能，因此叫做“魔法”方法
- 当使用print输出对象的时候，只要自己定义了 `__str__(self)` 方法，那么就会打印从在这个方法中return的数据

# self

## 1. 理解self

看如下示例：

```
# 定义一个类
class Animal:

    # 方法
    def __init__(self, name):
        self.name = name

    def printName(self):
        print('名字为：%s' % self.name)

# 定义一个函数
def myPrint(animal):
    animal.printName()

dog1 = Animal('西西')
myPrint(dog1)

dog2 = Animal('北北')
myPrint(dog2)
```

运行结果：

```
[python@ubuntu:~/Desktop/test]$ python3 oop2.py
名字为：西西
名字为：北北
python@ubuntu:~/Desktop/test$
```

## 总结

- 所谓的self，可以理解为自己
- 可以把self当做C++中类里面的this指针一样理解，就是对象自身的意思
- 某个对象调用其方法时，python解释器会把这个对象作为第一个参数传递给self，所以开发者只需要传递后面的参数即可

# 应用:烤地瓜

为了更好的理解面向对象编程，下面以“烤地瓜”为案例，进行分析

## 1. 分析“烤地瓜”的属性和方法

示例属性如下：

- cookedLevel：这是数字；0~3表示还是生的，超过3表示半生不熟，超过5表示已经烤好了，超过8表示已经烤成木炭了！我们的地瓜开始时时生的
- cookedString：这是字符串；描述地瓜的生熟程度
- condiments：这是地瓜的配料列表，比如番茄酱、芥末酱等

示例方法如下：

- cook()：把地瓜烤一段时间
- addCondiments()：给地瓜添加配料
- \_\_init\_\_()：设置默认的属性
- \_\_str\_\_()：让print的结果看起来更好一些

## 2. 定义类，并且定义 \_\_init\_\_() 方法

```
#定义`地瓜`类
class SweetPotato:
    '这是烤地瓜的类'

    #定义初始化方法
    def __init__(self):
        self.cookedLevel = 0
        self.cookedString = "生的"
```

```
self.condiments = []
```

### 3. 添加"烤地瓜"方法

```
#烤地瓜方法
def cook(self, time):
    self.cookedLevel += time
    if self.cookedLevel > 8:
        self.cookedString = "烤成灰了"
    elif self.cookedLevel > 5:
        self.cookedString = "烤好了"
    elif self.cookedLevel > 3:
        self.cookedString = "半生不熟"
    else:
        self.cookedString = "生的"
```

### 4. 基本的功能已经有了一部分， 赶紧测试一下

把上面2块代码合并为一个程序后，在代码的下面添加以下代码进行测试

```
mySweetPotato = SweetPotato()
print(mySweetPotato.cookedLevel)
print(mySweetPotato.cookedString)
print(mySweetPotato.condiments)
```

完整的代码为：

```
class SweetPotato:
    '这是烤地瓜的类'

    #定义初始化方法
    def __init__(self):
        self.cookedLevel = 0
        self.cookedString = "生的"
```

```
self.condiments = []

#烤地瓜方法
def cook(self, time):
    self.cookedLevel += time
    if self.cookedLevel > 8:
        self.cookedString = "烤成灰了"
    elif self.cookedLevel > 5:
        self.cookedString = "烤好了"
    elif self.cookedLevel > 3:
        self.cookedString = "半生不熟"
    else:
        self.cookedString = "生的"

# 用来进行测试
mySweetPotato = SweetPotato()
print(mySweetPotato.cookedLevel)
print(mySweetPotato.cookedString)
print(mySweetPotato.condiments)
```

```
python@ubuntu:~/Desktop/test$ python3 oop3.py
0
生的
[]
python@ubuntu:~/Desktop/test$
```

## 5. 测试cook方法是否好用

在上面的代码最后面添加如下代码:

```
print("-----接下来要进行烤地瓜了-----")
mySweetPotato.cook(4) #烤4分钟
print(mySweetPotato.cookedLevel)
print(mySweetPotato.cookedString)
```

```
[python@ubuntu:~/Desktop/test$ python3 oop3.py
0
生的
[]
-----接下来要进行烤地瓜了-----
4
半生不熟
python@ubuntu:~/Desktop/test$ ]
```

## 6. 定义 addCondiments() 方法 和 \_\_str\_\_() 方法

```
def __str__(self):
    msg = self.cookedString + " 地瓜"
    if len(self.condiments) > 0:
        msg = msg + "("
        for temp in self.condiments:
            msg = msg + temp + ", "
        msg = msg.strip(", ")

    msg = msg + ")"
    return msg

def addCondiments(self, condiments):
    self.condiments.append(condiments)
```

## 7. 再次测试

完整的代码如下：

```
class SweetPotato:
    "这是烤地瓜的类"
```

```
#定义初始化方法
def __init__(self):
    self.cookedLevel = 0
    self.cookedString = "生的"
    self.condiments = []

#定制print时的显示内容
def __str__(self):
    msg = self.cookedString + " 地瓜"
    if len(self.condiments) > 0:
        msg = msg + "("

        for temp in self.condiments:
            msg = msg + temp + ", "
        msg = msg.strip(", ")

    msg = msg + ")"
    return msg

#烤地瓜方法
def cook(self, time):
    self.cookedLevel += time
    if self.cookedLevel > 8:
        self.cookedString = "烤成灰了"
    elif self.cookedLevel > 5:
        self.cookedString = "烤好了"
    elif self.cookedLevel > 3:
        self.cookedString = "半生不熟"
    else:
        self.cookedString = "生的"

#添加配料
def addCondiments(self, condiments):
    self.condiments.append(condiments)

# 用来进行测试
mySweetPotato = SweetPotato()
print("-----有了一个地瓜, 还没有烤-----")
```

```
print(mySweetPotato.cookedLevel)
print(mySweetPotato.cookedString)
print(mySweetPotato.condiments)
print("-----接下来要进行烤地瓜了-----")
print("-----地瓜经烤了4分钟-----")
mySweetPotato.cook(4) #烤4分钟
print(mySweetPotato)
print("-----地瓜又经烤了3分钟-----")
mySweetPotato.cook(3) #又烤了3分钟
print(mySweetPotato)
print("-----接下来要添加配料-番茄酱-----")
mySweetPotato.addCondiments("番茄酱")
print(mySweetPotato)
print("-----地瓜又经烤了5分钟-----")
mySweetPotato.cook(5) #又烤了5分钟
print(mySweetPotato)
print("-----接下来要添加配料-芥末酱-----")
mySweetPotato.addCondiments("芥末酱")
print(mySweetPotato)
```

```
[python@ubuntu:~/Desktop/test$ python3 oop5.py
-----有了一个地瓜，还没有烤-----
0
生的
[]
-----接下来要进行烤地瓜了-----
-----地瓜经烤了4分钟-----
半生不熟 地瓜
-----地瓜又经烤了3分钟-----
烤好了 地瓜
-----接下来要添加配料-番茄酱-----
烤好了 地瓜(番茄酱)
-----地瓜又经烤了5分钟-----
烤成灰了 地瓜(番茄酱)
-----接下来要添加配料-芥末酱-----
烤成灰了 地瓜(番茄酱，芥末酱)
python@ubuntu:~/Desktop/test$ ]
```

## 隐藏数据

可能你已经意识到，查看过着修改对象的属性（数据），有2种方法

### 1. 直接通过对象名修改

```
SweetPotato.cookedLevel = 5
```

### 2. 通过方法间接修改

```
SweetPotato.cook(5)
```

## 分析

明明可以使用第1种方法直接修改，为什么还要定义方法来间接修改呢？

至少有2个原因：

- 如果直接修改属性，烤地瓜至少需要修改2部分，即修改cookedLevel和cookedString。而使用方法来修改时，只需要调用一次即可完成
- 如果直接访问属性，可能会出现一些数据设置错误的情况产生例如 cookedLevel = -3。这会使地瓜比以前还生，当然了这也没有任何意义，通过使用方法来进行修改，就可以在方法中进行数据合法性的检查

# 应用:存放家具

```
#定义一个home类
class Home:

    def __init__(self, area):
        self.area = area #房间剩余的可用面积
        #self.light = 'on' #灯默认是亮的
        self.containsItem = []

    def __str__(self):
        msg = "当前房间可用面积为：" + str(self.area)
        if len(self.containsItem) > 0:
            msg = msg + " 容纳的物品有："
            for temp in self.containsItem:
                msg = msg + temp.getName() + ", "
            msg = msg.strip(", ")
        return msg

    #容纳物品
    def accommodateItem(self, item):
        #如果可用面积大于物品的占用面积
        needArea = item.getUsedArea()
        if self.area > needArea:
            self.containsItem.append(item)
            self.area -= needArea
            print("ok:已经存放到房间中")
        else:
            print("err:房间可用面积为:%d,但是当前要存放的物品需要的面积")

#定义bed类
class Bed:

    def __init__(self, area, name = '床'):
```

```
    self.name = name
    self.area = area

def __str__(self):
    msg = '床的面积为：' + str(self.area)
    return msg

#获取床的占用面积
def getUsedArea(self):
    return self.area

def getName(self):
    return self.name

#创建一个新家对象
newHome = Home(100)#100平米
print(newHome)

#创建一个床对象
newBed = Bed(20)
print(newBed)

#把床安放到家里
newHome.accommodateItem(newBed)
print(newHome)

#创建一个床对象
newBed2 = Bed(30, '席梦思')
print(newBed2)

#把床安放到家里
newHome.accommodateItem(newBed2)
print(newHome)
```

```
[python@ubuntu:~/Desktop/test$ python3 oop6.py
当前房间可用面积为:100
床的面积为:20
ok:已经存放到房间中
当前房间可用面积为:80 容纳的物品有: 床
床的面积为:30
ok:已经存放到房间中
当前房间可用面积为:50 容纳的物品有: 床, 席梦思
python@ubuntu:~/Desktop/test$ ]
```

## 总结:

- 如果一个对象与另外一个对象有一定的关系，那么一个对象可用是另外一个对象的属性

## 思维升华:

- 添加“开、关”灯，让房间、床一起亮、灭



# 应用：老王开枪



## 1. 人类

- 属性
  - 姓名
  - 血量
  - 持有的枪
- 方法
  - 安子弹
  - 安弹夹
  - 拿枪（持有枪）
  - 开枪

## 2. 子弹类

- 属性
  - 杀伤力

- 方法
  - 伤害敌人(让敌人掉血)

### 3. 弹夹类

- 属性
  - 容量 (子弹存储的最大值)
  - 当前保存的子弹
- 方法
  - 保存子弹 (安装子弹的时候)
  - 弹出子弹 (开枪的时候)

### 4. 枪类

- 属性
  - 弹夹 (默认没有弹夹, 需要安装)
- 方法
  - 连接弹夹 (保存弹夹)
  - 射子弹

## 参考代码

```
#人类
class Ren:
    def __init__(self, name):
        self.name = name
        self.xue = 100
        self.qiang = None

    def __str__(self):
        return self.name + "剩余血量为：" + str(self.xue)

    def anzidan(self, danjia, zidan):
```

```

danjia.baocunzidan(zidan)

def andanjia(self, qiang, danjia):
    qiang.lianjiedanjia(danjia)

def naqiang(self, qiang):
    self.qiang = qiang

def kaiqiang(self, diren):
    self.qiang.she(diren)

def diaoxue(self, shashangli):
    self.xue -= shashangli

#弹夹类
class Danjia:
    def __init__(self, rongliang):
        self.rongliang = rongliang
        self.rongnaList = []

    def __str__(self):
        return "弹夹当前的子弹数量为：" + str(len(self.rongnaList))

    def baocunzidan(self, zidan):
        if len(self.rongnaList) < self.rongliang:
            self.rongnaList.append(zidan)

    def chuzidan(self):
        #判断当前弹夹中是否还有子弹
        if len(self.rongnaList) > 0:
            #获取最后压入到单间中的子弹
            zidan = self.rongnaList[-1]
            self.rongnaList.pop()
            return zidan
        else:
            return None

#子弹类

```

```
class Zidan:
    def __init__(self, shashangli):
        self.shashangli = shashangli

    def shanghai(self, diren):
        diren.diaoxue(self.shashangli)

#枪类
class Qiang:
    def __init__(self):
        self.danjia = None

    def __str__(self):
        if self.danjia:
            return "枪当前有弹夹"
        else:
            return "枪没有弹夹"

    def lianjiedanjia(self, danjia):
        if not self.danjia:
            self.danjia = danjia

    def she(self, diren):
        zidan = self.danjia.chuzidan()
        if zidan:
            zidan.shanghai(diren)
        else:
            print("没有子弹了，放了空枪....")

#创建一个人对象
laowang = Ren("老王")

#创建一个弹夹
danjia = Danjia(20)
print(danjia)
```

```
#循环的方式创建一颗子弹，然后让老王把这颗子弹压入到弹夹中
i=0
while i<5:
    zidan = Zidan(5)
    laowang.anzidan(danjia,zidan)
    i+=1
#测试一下，安装完子弹后，弹夹中的信息
print(danjia)

#创建一个枪对象
qiang = Qiang()
print(qiang)
#让老王，把弹夹连接到枪中
laowang.andanjia(qiang,danjia)
print(qiang)

#创建一个敌人
diren = Ren("敌人")
print(diren)

#让老王拿起枪
laowang.naqiang(qiang)

#老王开枪射敌人
laowang.kaiqiang(diren)
print(diren)
print(danjia)

laowang.kaiqiang(diren)
print(diren)
print(danjia)
```

# 保护对象的属性

如果有一个对象，当需要对其进行修改属性时，有2种方法

- 对象名.属性名 = 数据 ---->直接修改
- 对象名.方法名() ---->间接修改

为了更好的保存属性安全，即不能随意修改，一般的处理方式为

- 将属性定义为私有属性
- 添加一个可以调用的方法，供调用

```
class People(object):
    私有方法也是加__即可

    def __init__(self, name):
        self.__name = name

    def getName(self):
        return self.__name

    def setName(self, newName):
        if len(newName) >= 5:
            self.__name = newName
        else:
            print("error:名字长度需要大于或者等于5")

xiaoming = People("dongGe")
print(xiaoming.__name)
```

```
[python@ubuntu:~/Desktop/test$ python3 oop9.py
Traceback (most recent call last):
  File "oop9.py", line 17, in <module>
    print(xiaoming.__name)
AttributeError: 'people' object has no attribute '__name'
python@ubuntu:~/Desktop/test$ ]
```

```

class People(object):

    def __init__(self, name):
        self.__name = name

    def getName(self):
        return self.__name

    def setName(self, newName):
        if len(newName) >= 5:
            self.__name = newName
        else:
            print("error:名字长度需要大于或者等于5")

xiaoming = People("dongGe")

xiaoming.setName("wanger")
print(xiaoming.getName())

xiaoming.setName("lisi")
print(xiaoming.getName())

```

```
[python@ubuntu:~/Desktop/test$ python3 oop9.py
wanger
error:名字长度需要大于或者等于5
wanger
python@ubuntu:~/Desktop/test$ ]
```

## 总结

- Python中没有像C++中public和private这些关键字来区别公有属性和私有属性
- 它是以属性命名方式来区分，如果在属性名前面加了2个下划线'\_\_'，则表明该属性是私有属性，否则为公有属性（方法也是一样，方法名前面加了2个下划线的话表示该方法是私有的，否则为公有的）。



## \_\_del\_\_() 方法

创建对象后， python解释器默认调用 \_\_init\_\_() 方法；

当删除一个对象时， python解释器也会默认调用一个方法， 这个方法为 \_\_del\_\_() 方法

```
import time
class Animal(object):

    # 初始化方法
    # 创建完对象后会自动被调用
    def __init__(self, name):
        print('__init__方法被调用')
        self.__name = name

    # 析构方法
    # 当对象被删除时，会自动被调用
    def __del__(self):
        print("__del__方法被调用")
        print("%s对象马上被干掉了..." % self.__name)
```

```
# 创建对象
dog = Animal("哈皮狗")
```

```
# 删除对象
del dog
```

```
cat = Animal("波斯猫")
cat2 = cat
cat3 = cat
```

import sys

sys.getrefcount(a) 测试对象a指向的对象的连接数, 测量数量比实际数量多1, 若是对象完全被删除再去测试的话会产生crash

硬链接为零的时候调用

```
print("---马上 删除cat对象")
del cat
print("---马上 删除cat2对象")
del cat2
print("---马上 删除cat3对象")
del cat3

print("程序2秒钟后结束")
time.sleep(2)
```

结果：

```
[python@ubuntu:~/Desktop/test$ python3 oop10.py
__init__方法被调用
__del__方法被调用
哈皮狗对象马上被干掉了...
__init__方法被调用
---马上 删除cat对象
---马上 删除cat2对象
---马上 删除cat3对象
__del__方法被调用
波斯猫对象马上被干掉了...
程序2秒钟后结束
python@ubuntu:~/Desktop/test$ ]
```

### 总结

#### 硬链接数目

- 当有1个变量保存了对象的引用时，此对象的引用计数就会加1
- 当使用del删除变量指向的对象时，如果对象的引用计数不会1，比如3，那么此时只会让这个引用计数减1，即变为2，当再次调用del时，变为1，如果再调用1次del，此时会真的把对象进行删除

# 继承介绍以及单继承

## 1. 继承的概念

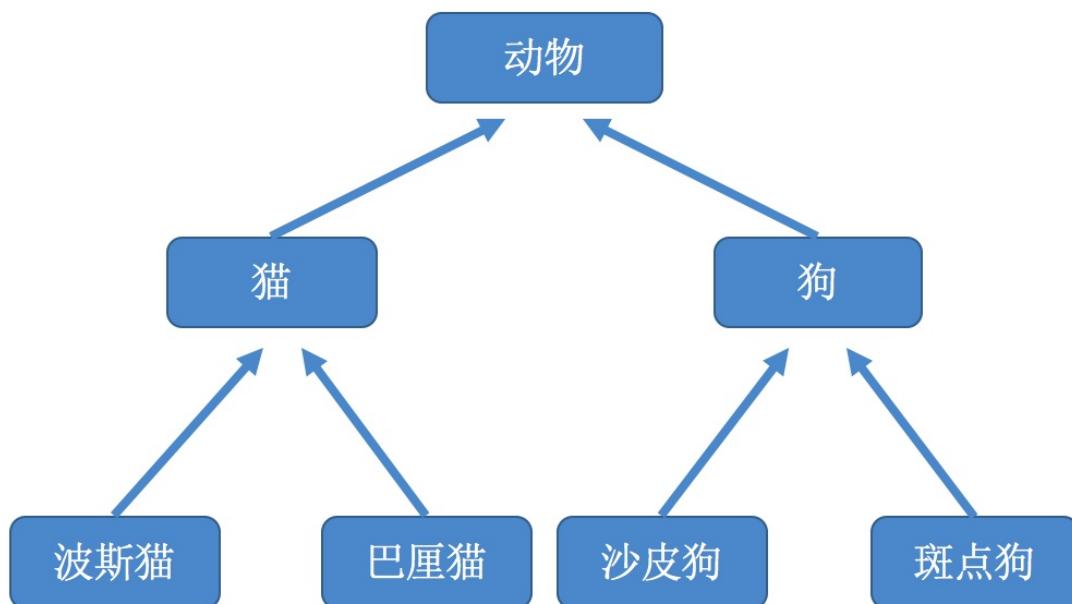
在现实生活中，继承一般指的是子女继承父辈的财产，如下图



搞不好,结果如下..



在程序中，继承描述的是事物之间的所属关系，例如猫和狗都属于动物，程序中便可以描述为猫和狗继承自动物；同理，波斯猫和巴厘猫都继承自猫，而沙皮狗和斑点狗都继承自狗，如下所示：



## 2. 继承示例

```
# 定义一个父类，如下：  
class Cat(object):  
  
    def __init__(self, name, color="白色"):  
        self.name = name  
        self.color = color  
  
    def run(self):  
        print("%s--在跑"%self.name)  
  
  
# 定义一个子类，继承Cat类如下：  
class Bosi(Cat):  
  
    def setNewName(self, newName):  
        self.name = newName  
  
    def eat(self):  
        print("%s--在吃"%self.name)  
  
  
bs = Bosi("印度猫")  
print('bs的名字为：%s'%bs.name)  
print('bs的颜色为：%s'%bs.color)  
bs.eat()  
bs.setNewName('波斯')  
bs.run()
```

运行结果：

```
[python@ubuntu:~/Desktop/test$ python3 oop11.py
bs的名字为:印度猫
bs的颜色为:白色
印度猫--在吃
波斯--在跑
python@ubuntu:~/Desktop/test$ ]
```

说明：

- 虽然子类没有定义 `__init__` 方法，但是父类有，所以在子类继承父类的时候这个方法就被继承了，所以只要创建Bosi的对象，就默认执行了那个继承过来的 `__init__` 方法

## 总结

- 子类在继承的时候，在定义类时，小括号()中为父类的名字
- 父类的属性、方法，会被继承给子类

## 3. 注意点

```
class Animal(object):

    def __init__(self, name='动物', color='白色'):
        self.__name = name
        self.color = color

    def __test(self):
        print(self.__name)
        print(self.color)

    def test(self):
        print(self.__name)
        print(self.color)
```

```
class Dog(Animal):
```

```
def dogTest1(self):
    #print(self.__name) #不能访问到父类的私有属性
    print(self.color)

def dogTest2(self):
    #self.__test() #不能访问父类中的私有方法
    self.test()

A = Animal()
#print(A.__name) #程序出现异常，不能访问私有属性
print(A.color)
#A.__test() #程序出现异常，不能访问私有方法
A.test()

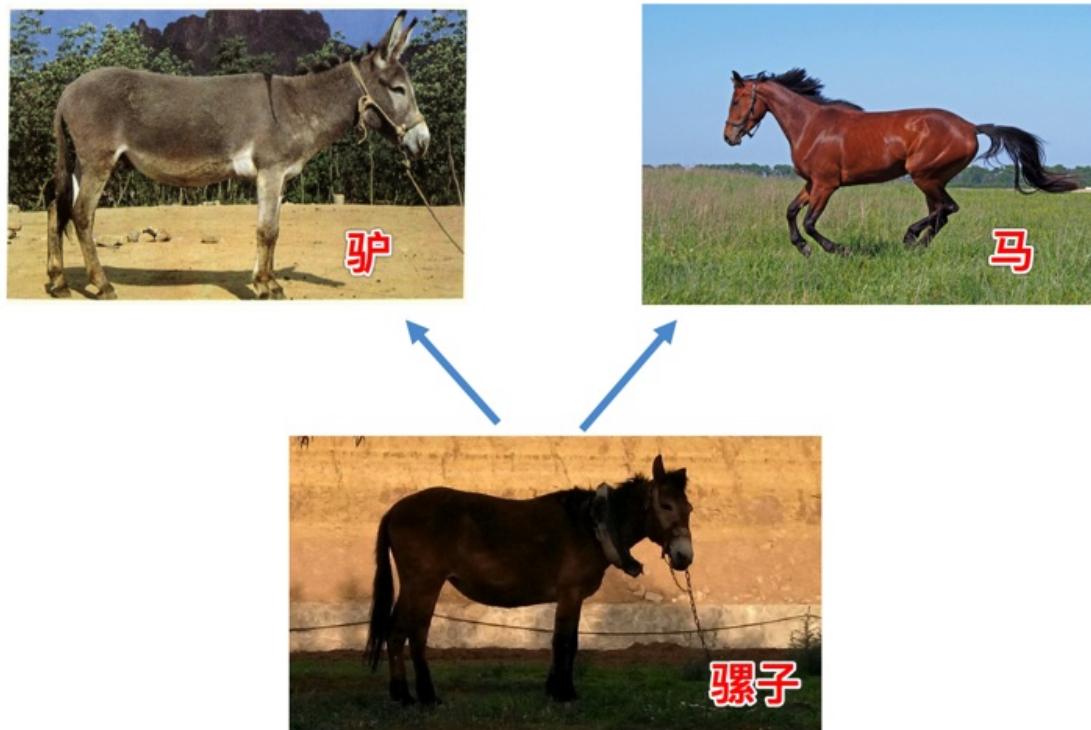
print("-----分割线-----")

D = Dog(name = "小花狗", color = "黄色")
D.dogTest1()
D.dogTest2()
```

- 私有的属性，不能通过对象直接访问，但是可以通过方法访问
- 私有的方法，不能通过对象直接访问
- 私有的属性、方法，不会被子类继承，也不能被访问
- 一般情况下，私有的属性、方法都是不对外公布的，往往用来做内部的事情，起到安全的作用

# 多继承

## 1. 多继承



从图中能够看出，所谓多继承，即子类有多个父类，并且具有它们的特征

Python中多继承的格式如下：

```
# 定义一个父类
class A:
    def printA(self):
        print('----A----')

# 定义一个父类
class B:
    def printB(self):
        print('----B----')
```

```
# 定义一个子类，继承自A、B
class C(A, B):
    def printC(self):
        print('----C----')

obj_C = C()
obj_C.printA()
obj_C.printB()
```

运行结果：

----A----  
----B----

方法寻找顺序：  
c, a, b, base  
顺序获取方式：  
print  
(C.\_\_mro\_\_)

### 说明

- python中是可以多继承的
- 父类中的方法、属性，子类会继承

## 注意点

- 想一想：

如果在上面的多继承例子中，如果父类A和父类B中，有一个同名的方法，那么通过子类去调用的时候，调用哪个？

```
#coding=utf-8
class base(object):
    def test(self):
        print('----base test----')
class A(base):
    def test(self):
        print('----A test----')

# 定义一个父类
class B(base):
```

```
def test(self):
    print('----B test----')

# 定义一个子类, 继承自A、B
class C(A, B):
    pass

obj_C = C()
obj_C.test()

print(C.__mro__) #可以查看C类的对象搜索方法时的先后顺序
```

# 重写父类方法与调用父类方法

## 1. 重写父类方法

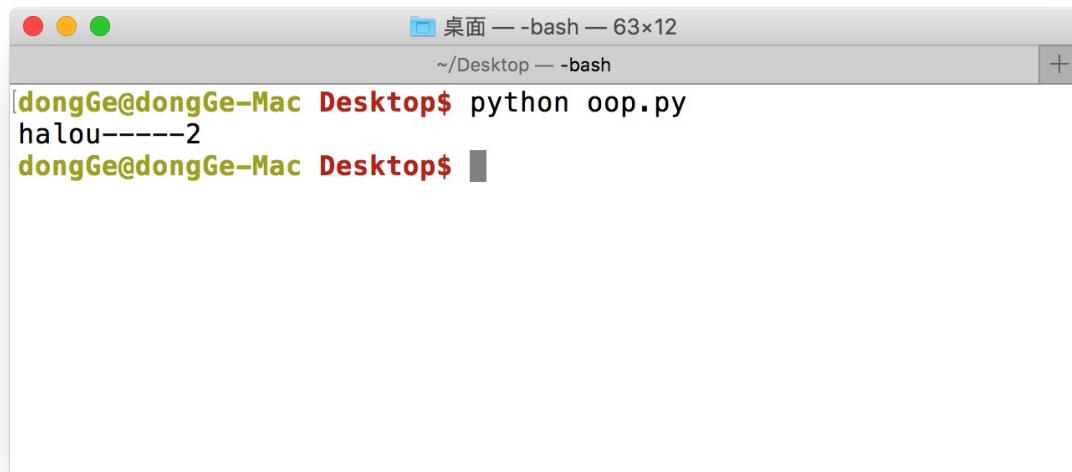
所谓重写，就是子类中，有一个和父类相同名字的方法，在子类中的方法会覆盖掉父类中同名的方法

```
#coding=utf-8
class Cat(object):
    def sayHello(self):
        print("halou----1")

class Bosi(Cat):
    def sayHello(self):
        print("halou----2")

bosi = Bosi()

bosi.sayHello()
```



## 2. 调用父类的方法

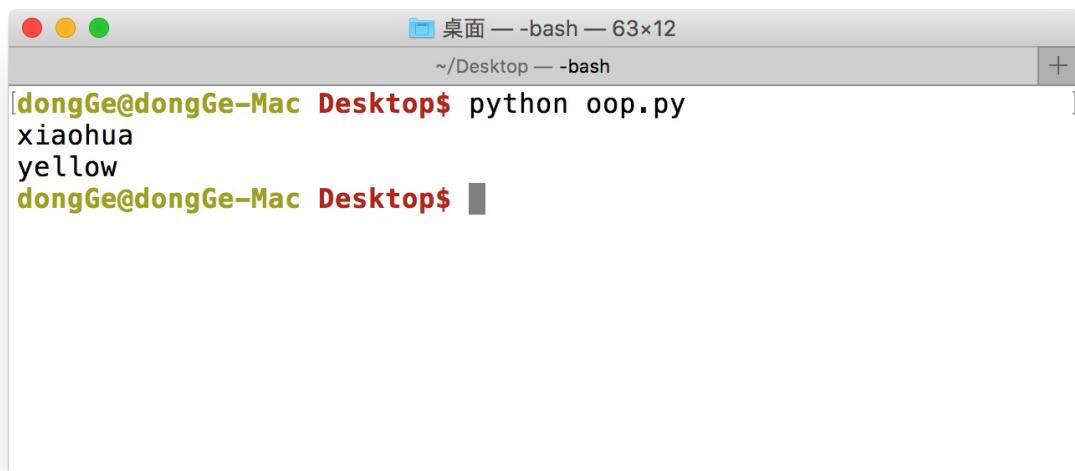
```
#coding=utf-8
class Cat(object):
    def __init__(self, name):
        self.name = name
        self.color = 'yellow'

class Bosi(Cat):
    def __init__(self, name):
        # 调用父类的__init__方法1(python2)
        #Cat.__init__(self, name)
        # 调用父类的__init__方法2
        #super(Bosi, self).__init__(name)
        # 调用父类的__init__方法3
        super().__init__(name)

    def getName(self):
        return self.name

bosi = Bosi('xiaohua')
```

```
print(bosi.name)  
print(bosi.color)
```



A screenshot of a macOS terminal window titled "桌面 — -bash — 63x12". The window shows the command "python oop.py" being run, followed by the output "xiaohua" and "yellow". The terminal has a standard OS X look with red, yellow, and green window control buttons.

```
[dongGe@dongGe-Mac Desktop$ python oop.py  
xiaohua  
yellow  
dongGe@dongGe-Mac Desktop$ ]
```

# 多态

多态的概念是应用于Java和C#这一类强类型语言中，而Python崇尚“鸭子类型”。

所谓多态：定义时的类型和运行时的类型不一样，此时就成为多态

- Python伪代码实现Java或C#的多态

```

class F1(object):
    def show(self):
        print 'F1.show'

class S1(F1):
    def show(self):
        print 'S1.show'

class S2(F1):
    def show(self):
        print 'S2.show'

# 由于在Java或C#中定义函数参数时，必须指定参数的类型
# 为了让Func函数既可以执行S1对象的show方法，又可以执行S2对象的show方法，F
# 而实际传入的参数是：S1对象和S2对象

def Func(F1 obj):
    """Func函数需要接收一个F1类型或者F1子类的类型"""

    print obj.show() 写的时候不知道到底调用的谁运行起来
                           就知道是谁就是多态了

s1_obj = S1()
Func(s1_obj) # 在Func函数中传入S1类的对象 s1_obj，执行 S1 的show方法，

s2_obj = S2()
Func(s2_obj) # 在Func函数中传入S2类的对象 s2_obj，执行 S2 的show方法,

```

- Python “鸭子类型”

```
class F1(object):
    def show(self):
        print 'F1.show'

class S1(F1):
    def show(self):
        print 'S1.show'

class S2(F1):
    def show(self):
        print 'S2.show'

def Func(obj):
    print obj.show()

s1_obj = S1()
Func(s1_obj)

s2_obj = S2()
Func(s2_obj)
```

# 类属性、实例属性

在了解了类基本的东西之后，下面看一下python中这几个概念的区别

先来谈一下 类属性 和 实例属性

在前面的例子中我们接触到的就是实例属性（对象属性），顾名思义，类属性就是类对象所拥有的属性，它被所有类对象的实例对象所共有，在内存中只存在一个副本，这个和C++中类的静态成员变量有点类似。对于公有的类属性，在类外可以通过类对象和实例对象访问

## 类属性

```
class People(object):
    name = 'Tom'      #公有的类属性
    __age = 12         #私有的类属性

p = People()

print(p.name)          #正确
print(People.name)     #正确
print(p.__age)         #错误，不能在类外通过实例对象访问私有的类属性
print(People.__age)    #错误，不能在类外通过类对象访问私有的类属性
```

## 实例属性(对象属性)

```
class People(object):
    address = '山东' #类属性
    def __init__(self):
        self.name = 'xiaowang' #实例属性
        self.age = 20 #实例属性
```

```
p = People()
p.age =12 #实例属性
print(p.address) #正确
print(p.name)    #正确
print(p.age)     #正确

print(People.address) #正确
print(People.name)   #错误
print(People.age)   #错误
```

## 通过实例(对象)去修改类属性

```
class People(object):
    country = 'china' #类属性

print(People.country)
p = People()
print(p.country)
p.country = 'japan'
print(p.country)      #实例属性会屏蔽掉同名的类属性
print(People.country)
del p.country        #删除实例属性
print(p.country)
```



The screenshot shows a terminal window titled '桌面 — -bash — 68x9' with the command '/Desktop — -bash'. The output of the script 'oop.py' is displayed:

```
[dongGe@bogon Desktop$ python oop.py
china
china
japan
china
china
dongGe@bogon Desktop$ ]
```

## 总结

- 如果需要在类外修改 类属性，必须通过 类对象 去引用然后进行修改。如果通过实例对象去引用，会产生一个同名的 实例属性，这种方式修改的是 实例属性，不会影响到 类属性，并且之后如果通过实例对象去引用该名称的属性，实例属性会强制屏蔽掉类属性，即引用的是 实例属性，除非删除了该 实例属性。

```
Dubuntu: ~/Desktop/python06期/python基础-08-面向对象2
1 class Tool(object):
2
3     #类属性
4     num = 0
5
6     #方法
7     def __init__(self, new_name):
8         #实例属性
9         self.name = new_name
10        #对类属性+=1
11        Tool.num += 1
12
13
14 tool1 = Tool("铁锹")
15 tool2 = Tool("工兵铲")
16 tool3 = Tool("水桶")
17
18 print(Tool.num)
~
```

# 静态方法和类方法

## 1. 类方法

是类对象所拥有的方法，需要用修饰器 `@classmethod` 来标识其为类方法，对于类方法，第一个参数必须是类对象，一般以 `cls` 作为第一个参数（当然可以用其他名称的变量作为其第一个参数，但是大部分人都习惯以'cls'作为第一个参数的名字，就最好用'cls'了），能够通过实例对象和类对象去访问。

```
class People(object):
    country = 'china'

    #类方法，用classmethod来进行修饰
    @classmethod
    def getCountry(cls):
        return cls.country

p = People()
print p.getCountry()      #可以用过实例对象引用
print People.getCountry() #可以通过类对象引用
```

类方法还有一个用途就是可以对类属性进行修改：

```
class People(object):
    country = 'china'

    #类方法，用classmethod来进行修饰
    @classmethod
    def getCountry(cls):
        return cls.country

    @classmethod
    def setCountry(cls,country):
        cls.country = country
```

```
p = People()  
print p.getCountry()      #可以用过实例对象引用  
print People.getCountry()    #可以通过类对象引用  
  
p.setCountry('japan')  
  
print p.getCountry()  
print People.getCountry()
```



```
[dongGe@bogon Desktop$ python oop.py  
china  
china  
japan  
japan  
dongGe@bogon Desktop$ ]
```

结果显示在用类方法对类属性修改之后，通过类对象和实例对象访问都发生了改变

## 2. 静态方法

需要通过修饰器 `@staticmethod` 来进行修饰，静态方法不需要多定义参数

```
class People(object):  
    country = 'china'  
  
    @staticmethod  
    #静态方法  
    def getCountry():  
        return People.country
```

```
print People.getCountry()
```

## 总结

从类方法和实例方法以及静态方法的定义形式就可以看出来，类方法的第一个参数是类对象cls，那么通过cls引用的必定是类对象的属性和方法；而实例方法的第一个参数是实例对象self，那么通过self引用的可能是类属性、也有可能是实例属性（这个需要具体分析），不过在存在相同名称的类属性和实例属性的情况下，实例属性优先级更高。静态方法中不需要额外定义参数，因此在静态方法中引用类属性的话，必须通过类对象来引用

```

1 class Game(object):
2
3     #类属性
4     num = 0
5
6     #实例方法
7     def __init__(self):
8         #实例属性
9         self.name = "laowang"
10
11    #类方法
12    @classmethod
13    def add_num(cls):
14        cls.num = 100
15
16    #静态方法
17    @staticmethod
18    def print_menu():
19        print("-----")
20        print("    穿越火线V11.1")
21        print("  1. 开始游戏")
22        print("  2. 结束游戏")
23        print("-----")
24
25 game = Game()
26 #Game.add_num()#可以通过类的名字调用类方法
27 game.add_num()#还可以通过这个类创建出来的对象 去调用这个类方法
28 print(Game.num)
29
30 #Game.print_menu()#通过类 去调用静态方法
31 game.print_menu()#通过实例对象 去调用静态方法

```



## 练习：设计类

### 1. 设计一个卖车的4S店，该怎样做呢？

```
# 定义车类
class Car(object):

    # 定义车的方法
    def move(self):
        print("---车在移动---")

    def stop(self):
        print("---停车---")

# 定义一个销售车的店类
class CarStore(object):

    def order(self):
        self.car = Car() #找一辆车
        self.car.move()
        self.car.stop()
```

#### 说明

上面的4s店，只能销售那一种类型的车

如果这个是个销售北京现代品牌的车，比如伊兰特、索纳塔等，该怎样做呢？

### 2. 设计一个卖北京现代车的4S店

```
# 定义伊兰特车类
class YilanteCar(object):

    # 定义车的方法
    def move(self):
        print("---车在移动---")

    def stop(self):
        print("---停车---")

# 定义索纳塔车类
class SuonataCar(object):

    # 定义车的方法
    def move(self):
        print("---车在移动---")

    def stop(self):
        print("---停车---")

# 定义一个销售北京现代车的店类
class CarStore(object):

    def order(self, typeName):
        #根据客户的不同要求，生成不同的类型的车
        if typeName == "伊兰特":
            car = YilanteCar()
        elif typeName == "索纳塔":
            car = SuonataCar()
        return car
```

这样做，不太好，因为当北京现代又生产一种新类型的车时，又得在CarStore类中修改，有没有好的解决办法呢？

# 工厂模式

## 1. 简单工厂模式

在上一节中，最后留下的个问题，该怎样解决呢？

### 1.1. 使用函数实现

```
# 定义伊兰特车类
class YilanteCar(object):

    # 定义车的方法
    def move(self):
        print("---车在移动---")

    def stop(self):
        print("---停车---")

# 定义索纳塔车类
class SuonataCar(object):

    # 定义车的方法
    def move(self):
        print("---车在移动---")

    def stop(self):
        print("---停车---")

# 用来生成具体的对象
def createCar(typeName):
    if typeName == "伊兰特":
```

```
    car = YilanteCar()
elif typeName == "索纳塔":
    car = SuonataCar()
return car

# 定义一个销售北京现代车的店类
class CarStore(object):

    def order(self, typeName):
        # 让工厂根据类型，生产一辆汽车
        car = createCar(typeName)
        return car
```

## 1.2. 使用类来实现

```
# 定义伊兰特车类
class YilanteCar(object):

    # 定义车的方法
    def move(self):
        print("---车在移动---")

    def stop(self):
        print("---停车---")

# 定义索纳塔车类
class SuonataCar(object):

    # 定义车的方法
    def move(self):
        print("---车在移动---")

    def stop(self):
        print("---停车---")
```

```

# 定义一个生产汽车的工厂，让其根据具体的订单生产车
class CarFactory(object):

    def createCar(self, typeName):
        if typeName == "伊兰特":
            car = YilanteCar()
        elif typeName == "索纳塔":
            car = SuonataCar()

        return car

# 定义一个销售北京现代车的店类
class CarStore(object):

    def __init__(self):
        #设置4s店的指定生产汽车的工厂
        self.carFactory = CarFactory()

    def order(self, typeName):
        # 让工厂根据类型，生产一辆汽车
        car = self.carFactory.createCar(typeName)
        return car

```

咋一看来，好像只是把生产环节重新创建了一个类，这确实比较像是一种编程习惯，此种解决方式被称作**简单工厂模式**

工厂函数、工厂类对具体的生成环节进行了封装，这样有利于代码的后续扩展，即把功能划分的更具体，4s店只负责销售，汽车厂只负责制造

## 2. 工厂方法模式

### 多种品牌的汽车4S店

当买车时，有很多种品牌可以选择，比如北京现代、别克、凯迪拉克、特斯拉等，那么此时该怎样进行设计呢？

```
# 定义一个基本的4S店类
class CarStore(object):

    #仅仅是定义了有这个方法，并没有实现，具体功能，这个需要在子类中实现
    def createCar(self, typeName):
        pass

    def order(self, typeName):
        # 让工厂根据类型，生产一辆汽车
        self.car = self.createCar(typeName)
        self.car.move()
        self.car.stop()

# 定义一个北京现代4S店类
class XiandaiCarStore(CarStore):

    def createCar(self, typeName):
        self.carFactory = CarFactory()
        return self.carFactory.createCar(typeName)

# 定义伊兰特车类
class YilanteCar(object):

    # 定义车的方法
    def move(self):
        print("---车在移动---")

    def stop(self):
        print("---停车---")

# 定义索纳塔车类
class SuonataCar(object):

    # 定义车的方法
    def move(self):
        print("---车在移动---")
```

```
def stop(self):
    print("---停车---")

# 定义一个生产汽车的工厂，让其根据具体得订单生产车
class CarFactory(object):

    def createCar(self, typeName):
        self.typeName = typeName
        if self.typeName == "伊兰特":
            self.car = YilanteCar()
        elif self.typeName == "索纳塔":
            self.car = SuonataCar()

    return self.car

suonata = XiandaiCarStore()
suonata.order("索纳塔")
```

## 最后来看看 工厂方法模式 的定义

定义了一个创建对象的 接口 (可以理解为函数)，但由子类决定要实例化的类是哪一个， **工厂方法模式**让类的实例化推迟到子类，抽象的 CarStore 提供了一个创建对象的方法 createCar，也叫作 **工厂方法**。

子类真正实现这个 createCar 方法创建出具体产品。创建者类不需要直到实际创建的产品是哪一个，选择了使用了哪个子类，自然也就决定了实际创建的产品是什么。

## \_\_new\_\_ 方法

### \_\_new\_\_和\_\_init\_\_ 的作用

```
class A(object):
    def __init__(self):
        print("这是 init 方法")

    def __new__(cls):
        print("这是 new 方法")
        return object.__new__(cls)

A()
```

## 总结

- \_\_new\_\_ 至少要有一个参数cls，代表要实例化的类，此参数在实例化时由Python解释器自动提供
- \_\_new\_\_ 必须要有返回值，返回实例化出来的实例，这点在自己实现 \_\_new\_\_ 时要特别注意，可以return父类 \_\_new\_\_ 出来的实例，或者直接是object的 \_\_new\_\_ 出来的实例
- \_\_init\_\_ 有一个参数self，就是这个 \_\_new\_\_ 返回的实例，\_\_init\_\_ 在 \_\_new\_\_ 的基础上可以完成一些其它初始化的动作，\_\_init\_\_ 不需要返回值
- 我们可以将类比作制造商，\_\_new\_\_ 方法就是前期的原材料购买环节，\_\_init\_\_ 方法就是在有原材料的基础上，加工，初始化商品环节

## 注意点

```
In [8]: class A(object):
...:     def __init__(self):
...:         print(self)
...:         print("这是 init 方法")
...:
...:     def __new__(cls):
...:         print(id(cls))
...:         print("这是 new 方法")
...:         ret = object.__new__(cls)
...:         print(ret)
...:         return ret
...:
```

```
[In 9]: print(id(A))
140592776843416
```

```
[In 10]: a = A()
140592776843416
这是 new 方法
<__main__.A object at 0x105b96ac8>
<__main__.A object at 0x105b96ac8>
这是 init 方法
```

```
In [11]: 
```

# 单例模式

## 1. 单例是什么

举个常见的单例模式例子，我们日常使用的电脑上都有一个回收站，在整个操作系统中，回收站只能有一个实例，整个系统都使用这个唯一的实例，而且回收站自行提供自己的实例。因此回收站是单例模式的应用。

确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，单例模式是一种对象创建型模式。

## 2. 创建单例-保证只有1个对象

```
# 实例化一个单例
class Singleton(object):
    __instance = None

    def __new__(cls, age, name):
        #如果类数字能够__instance没有或者没有赋值
        #那么就创建一个对象，并且赋值为这个对象的引用，保证下次调用这个方法
        #能够知道之前已经创建过对象了，这样就保证了只有1个对象
        if not cls.__instance:
            cls.__instance = object.__new__(cls)
        return cls.__instance

a = Singleton(18, "dongGe")
b = Singleton(8, "dongGe")

print(id(a))
print(id(b))

a.age = 19 #给a指向的对象添加一个属性
print(b.age) #获取b指向的对象的age属性
```

运行结果：

```
In [12]: class Singleton(object):
....:     __instance = None
....:
....:     def __new__(cls, age, name):
....:         if not cls.__instance:
....:             cls.__instance = object.__new__(cls)
....:         return cls.__instance
....:
....:
....: a = Singleton(18, "dongGe")
....: b = Singleton(8, "dongGe")
....:
....: print(id(a))
....: print(id(b))
....:
....: a.age = 19
....: print(b.age)
....:
....:
4391023224
4391023224
19
```

### 3. 创建单例时，只执行1次\_\_init\_\_方法

```
# 实例化一个单例
class Singleton(object):
    __instance = None
    __first_init = False

    def __new__(cls, age, name):
        if not cls.__instance:
            cls.__instance = object.__new__(cls)
```

```
return cls.__instance

def __init__(self, age, name):
    if not self.__first_init:
        self.age = age
        self.name = name
    Singleton.__first_init = True


a = Singleton(18, "dongGe")
b = Singleton(8, "dongGe")

print(id(a))
print(id(b))

print(a.age)
print(b.age)

a.age = 19
print(b.age)
```

运行结果：

```
python@ubuntu:~/Desktop/test/11-单例模式$ python3 01-单例模式.py
140108717238928
140108717238928
18
18
19
python@ubuntu:~/Desktop/test/11-单例模式$
```

# 异常

## <1>异常简介

看如下示例:

```
print '----test--1---'
open('123.txt','r')
print '----test--2---'
```

运行结果:

```
code@ubuntu:~/python-test$ python test.py
----test--1---
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    open('123.txt','r')
IOError: [Errno 2] No such file or directory: '123.txt'
```

说明:

打开一个不存在的文件123.txt，当找不到123.txt文件时，就会抛出给我们一个IOError类型的错误，No such file or directory: 123.txt（没有123.txt这样的文件或目录）

**异常:**

当Python检测到一个错误时，解释器就无法继续执行了，反而出现了一些错误的提示，这就是所谓的“异常”

## 案例剖析

### <1>捕获异常 try...except...

看如下示例：

```
try:
    print('-----test--1---')
    open('123.txt','r')
    print('-----test--2---')
except IOError:
    pass
```

运行结果：

MacBook-Pro 01-python基础班-资料 \$ python test.py  
-----test--1---

说明：

- 此程序看不到任何错误，因为用except捕获到了IOError异常，并添加了处理的方法
- pass表示实现了相应的实现，但什么也不做；如果把pass改为print语句，那么就会输出其他信息

小总结：

```
try:
    print '-----test--1---'
    open('123.txt','r')
    print '-----test--2---'
except IOError:
    pass
```

可能产生异常的代码, 放在try中

如果产生错误时, 处理的方法

- 把可能出现问题的代码，放在try中
- 把处理异常的代码，放在except中

## <2> except捕获多个异常

看如下示例：

```
try:  
    print num  
except IOError:  
    print('产生错误了')
```

运行结果如下：

```
[MacBook-Pro 01-python基础班-资料$ python test.py  
Traceback (most recent call last):  
  File "test.py", line 4, in <module>  
    print num  
NameError: name 'num' is not defined
```

想一想：

上例程序，已经使用except来捕获异常了，为什么还会看到错误的信息提示？

答：

except捕获的错误类型是IOError，而此时程序产生的异常为NameError，所以except没有生效

修改后的代码为：

```
try:  
    print num  
except NameError:  
    print('产生错误了')
```

运行结果如下：

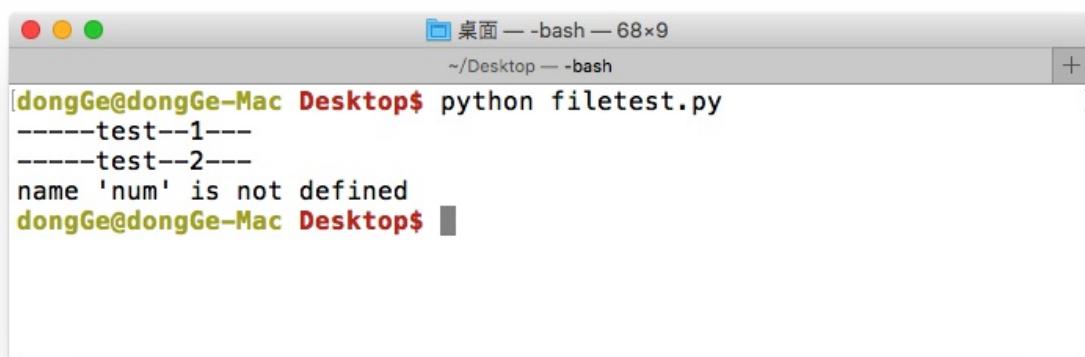
MacBook-Pro 01-python基础班-资料 \$ python test.py  
产生错误了；

实际开发中，捕获多个异常的方式，如下：

```
#coding=utf-8
try:
    print('-----test--1---')
    open('123.txt', 'r') # 如果123.txt文件不存在，那么会产生 IOError
    print('-----test--2---')
    print(num) # 如果num变量没有定义，那么会产生 NameError 异常

except (IOError, NameError):
    #如果想通过一次except捕获到多个异常可以用一个元组的方式

    # errorMsg里会保存捕获到的错误信息
    print(errorMsg)
```



注意：

- 当捕获多个异常时，可以把要捕获的异常的名字，放到except后，并使用元组的方式仅进行存储

## <3>获取异常的信息描述

```
In [8]: try:
....:     print(a)
....: except NameError as result
....:     print(result)
....:
name 'a' is not defined
```

存储异常的基本信息

要捕获的异常

```
In [13]: try:
....:     open("a.txt")
....: except (NameError,IOError) as result:
....:     print("哈哈, 捕获到了异常")
....:     print("异常的基本信息是:",result)
....:
哈哈, 捕获到了异常
异常的基本信息是: [Errno 2] No such file or directory: 'a.txt'
```

捕获多个异常，并且存储异常的基本信息

## <4>捕获所有异常

```
In [14]: try:
....:     open("a.txt")
....: except:
....:     print("产生了一个异常")
....:
产生了一个异常
```

没有存储异常的基本信息

```
In [15]: try:
....:     open("a.txt")
....: except Exception as result:
....:     print("捕获到了异常")
....:     print(result)
....:
捕获到了异常
[Errno 2] No such file or directory: 'a.txt'
```

捕获所有异常，并且存储异常的基本信息

## <5> else

咱们应该对 `else` 并不陌生，在`if`中，它的作用是当条件不满足时执行的实行；同样在`try...except...`中也是如此，即如果没有捕获到异常，那么就执行`else`中的事情

```

try:
    num = 100
    print num
except NameError as errorMsg:
    print('产生错误了:%s'%errorMsg)
else:
    print('没有捕获到异常, 真高兴')

```

运行结果如下:

```

In [17]: try:
....:     num = 100
....:     print(num)
....: except NameError as result:
....:     print("捕获到了一个异常, 信息是:%s"%result)
....: else:
....:     print("程序正常运行没有捕获到异常")
....:
100
程序正常运行没有捕获到异常

```

## <6> try...finally...

try...finally...语句用来表达这样的情况:

在程序中, 如果一个段代码必须要执行, 即无论异常是否产生都要执行, 那么此时就需要使用finally。比如文件关闭, 释放锁, 把数据库连接返还给连接池等

demo:

```

import time
try:
    f = open('test.txt')
    try:
        while True:
            content = f.readline()
            if len(content) == 0:
                break

```

```
    time.sleep(2)
    print(content)
except:
    #如果在读取文件的过程中，产生了异常，那么就会捕获到
    #比如 按下了 ctrl+c
    pass
finally:
    f.close()
    print('关闭文件')
except:
    print("没有这个文件")
```

### 说明：

test.txt文件中每一行数据打印，但是我有意在每打印一行之前用time.sleep方法暂停2秒钟。这样做的原因是让程序运行得慢一些。在程序运行的时候，按Ctrl+c中断（取消）程序。

我们可以观察到KeyboardInterrupt异常被触发，程序退出。但是在程序退出之前，finally从句仍然被执行，把文件关闭。

# 异常的传递

## 1. try嵌套中

```
import time
try:
    f = open('test.txt')
    try:
        while True:
            content = f.readline()
            if len(content) == 0:
                break
            time.sleep(2)
            print(content)
    finally:
        f.close()
        print('关闭文件')
except:
    print("没有这个文件")
```

运行结果：

```
In [26]: import time
....: try:
....:     f = open('test.txt')
....:     try:
....:         while True:
....:             content = f.readline()
....:             if len(content) == 0:
....:                 break
....:             time.sleep(2)
....:             print(content)
....:     finally:
```

```

....:         f.close()
....:         print('关闭文件')
....:     except:
....:         print("没有这个文件")
....:     finally:
....:         print("最后的finally")
....:

```

xxxxxx-->这是test.txt文件中读取到信息  
^C关闭文件  
没有这个文件  
最后的finally

## 2. 函数嵌套调用中

```

def test1():
    print("----test1-1----")
    print(num)
    print("----test1-2----")

def test2():
    print("----test2-1----")
    test1()
    print("----test2-2----")

def test3():
    try:
        print("----test3-1----")
        test1()
        print("----test3-2----")
    except Exception as result:
        print("捕获到了异常, 信息是:%s"%result)

    print("----test3-2----")

```

```
test3()
print("-----华丽的分割线-----")
test2()
```

运行结果：

```
python@ubuntu:~/Desktop/test$ python3 04-tye-except.py
----test3-1----
----test1-1----
捕获到了异常，信息是:name 'num' is not defined
----test3-3----
-----华丽的分割线-----
----test2-1----
----test1-1----
Traceback (most recent call last):
  File "04-tye-except.py", line 26, in <module>
    test2()
  File "04-tye-except.py", line 8, in test2
    test1()
  File "04-tye-except.py", line 3, in test1
    print(num)
NameError: name 'num' is not defined
```

总结：

- 如果try嵌套，那么如果里面的try没有捕获到这个异常，那么外面的try会接收到这个异常，然后进行处理，如果外边的try依然没有捕获到，那么再进行传递。。。
- 如果一个异常是在一个函数中产生的，例如函数A--->函数B--->函数C，而异常是在函数C中产生的，那么如果函数C中没有对这个异常进行处理，那么这个异常会传递到函数B中，如果函数B有异常处理那么就会按照函数B的处理方式进行执行；如果函数B也没有异常处理，那么这个异常会继续传递，以此类推。。。如果所有的函数都没有处理，那么此时就会进行异常的默认处理，即通常见到的那样
- 注意观察上图中，当调用test3函数时，在test1函数内部产生了异常，此异常被传递到test3函数中完成了异常处理，而当异常处理完后，并没有返回到函数test1中进行执行，而是在函数test3中继续执行

# 抛出自定义的异常

你可以用raise语句来引发一个异常。异常/错误对象必须有一个名字，且它们应是Error或Exception类的子类

下面是一个引发异常的例子：

```
class ShortInputException(Exception):
    '''自定义的异常类'''
    def __init__(self, length, atleast):
        #super().__init__()
        self.length = length
        self.atleast = atleast

def main():
    try:
        s = input('请输入 --> ')
        if len(s) < 3:
            # raise引发一个你定义的异常
            raise ShortInputException(len(s), 3)
    except ShortInputException as result:#x这个变量被绑定到了错误的
        print('ShortInputException: 输入的长度是 %d, 长度至少应是 %d' % result)
    else:
        print('没有异常发生。')

main()
```

运行结果如下：

```
python@ubuntu:~/Desktop/test$ subl 05-try-except.py
python@ubuntu:~/Desktop/test$ python3 05-try-except.py
请输入 --> hello
没有异常发生.
python@ubuntu:~/Desktop/test$ python3 05-try-except.py
请输入 --> abc
没有异常发生.
python@ubuntu:~/Desktop/test$ python3 05-try-except.py
请输入 --> ab
ShortInputException: 输入的长度是 2,长度至少应是 3
python@ubuntu:~/Desktop/test$ python3 05-try-except.py
请输入 --> a
ShortInputException: 输入的长度是 1,长度至少应是 3
python@ubuntu:~/Desktop/test$
python@ubuntu:~/Desktop/test$ python3 05-try-except.py
请输入 --> 只输入回车
ShortInputException: 输入的长度是 0,长度至少应是 3
python@ubuntu:~/Desktop/test$
```

## 注意

- 以上程序中，关于代码 `#super().__init__()` 的说明

这一行代码，可以调用也可以不调用，建议调用，因为 `__init__` 方法往往是用来对创建完的对象进行初始化工作，如果在子类中重写了父类的 `__init__` 方法，即意味着父类中的很多初始化工作没有做，这样就不保证程序的稳定了，所以在以后的开发中，如果重写了父类的 `__init__` 方法，最好是先调用父类的这个方法，然后再添加自己的功能

# 异常处理中抛出异常

```

class Test(object):
    def __init__(self, switch):
        self.switch = switch #开关
    def calc(self, a, b):
        try:
            return a/b
        except Exception as result:
            if self.switch:
                print("捕获开启，已经捕获到了异常，信息如下:")
                print(result)
            else:
                #重新抛出这个异常，此时就不会被这个异常处理给捕获到，从而
                raise
a = Test(True)
a.calc(11,0)

print("-----华丽的分割线-----")

a.switch = False
a.calc(11,0)

```

运行结果：

```

python@ubuntu:~/Desktop/test$ python3 06-try-except.py
捕获开启，已经捕获到了异常，信息如下:
division by zero
-----华丽的分割线-----
Traceback (most recent call last):
  File "06-try-except.py", line 22, in <module>
    a.calc(11,0)
  File "06-try-except.py", line 6, in calc
    return a/b
ZeroDivisionError: division by zero

```



# 模块

## <1>Python中的模块

有过C语言编程经验的朋友都知道在C语言中如果要引用 `sqrt` 函数，必须用语句 `#include <math.h>` 引入 `math.h` 这个头文件，否则是无法正常进行调用的。

那么在Python中，如果要引用一些其他的函数，该怎么处理呢？

在Python中有一个概念叫做模块（module），这个和C语言中的头文件以及Java中的包很类似，比如在Python中要调用 `sqrt` 函数，必须用 `import` 关键字引入 `math` 这个模块，下面就来了解一下Python中的模块。

说的通俗点：模块就好比是工具包，要想使用这个工具包中的工具（就好比函数），就需要导入这个模块

## <2>import

在Python中用关键字 `import` 来引入某个模块，比如要引用模块 `math`，就可以在文件最开始的地方用 `import math` 来引入。

形如：

```
import module1, module2...
```

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。

在调用 `math` 模块中的函数时，必须这样引用：

```
模块名.函数名
```

- 想一想：

## 为什么必须加上模块名调用呢?

- 答:

因为可能存在这样一种情况：在多个模块中含有相同名称的函数，此时如果只是通过函数名来调用，解释器无法知道到底要调用哪个函数。所以如果像上述这样引入模块的时候，调用函数必须加上模块名

```
import math

#这样会报错
print sqrt(2)

#这样才能正确输出结果
print math.sqrt(2)
```

有时候我们只需要用到模块中的某个函数，只需要引入该函数即可，此时可以用下面方法实现：

```
from 模块名 import 函数名1, 函数名2....
```

不仅可以引入函数，还可以引入一些全局变量、类等

- 注意:

- 通过这种方式引入的时候，调用函数时只能给出函数名，不能给出模块名，但是当两个模块中含有相同名称函数的时候，后面一次引入会覆盖前一次引入。也就是说假如模块A中有函数function( )，在模块B中也有函数function( )，如果引入A中的function在先、B中的function在后，那么当调用function函数的时候，是去执行模块B中的function函数。
- 如果想一次性引入math中所有的东西，还可以通过from math import \*来实现

## <3>from...import

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fib的fibonacci函数，使用如下语句：

```
from fib import fibonacci
```

### 注意

- 不会把整个fib模块导入到当前的命名空间中，它只会将fib里的fibonacci单个引入

## <4>from ... import \*

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 *
```

### 注意

- 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。

## <5> as

```
In [1]: import time as tt

In [2]: time.sleep(1)
-----
NameError                                 Traceback (most re
<ipython-input-2-07a34f5b1e42> in <module>()
----> 1 time.sleep(1)

NameError: name 'time' is not defined

In [3]: 

In [3]: 

In [3]: tt.sleep(1)

In [4]: 

In [4]: 

In [4]: from time import sleep as sp

In [5]: sleep(1)
-----
NameError                                 Traceback (most re
<ipython-input-5-82e5c2913b44> in <module>()
----> 1 sleep(1)

NameError: name 'sleep' is not defined

In [6]: 

In [6]: 

In [6]: sp(1)

In [7]:
```

## <6>定位模块

当你导入一个模块， Python解析器对模块位置的搜索顺序是：

1. 当前目录
2. 如果不在当前目录， Python则搜索在shell变量PYTHONPATH下的每个目录。
3. 如果都找不到， Python会察看默认路径。UNIX下， 默认路径一般为/usr/local/lib/python/
4. 模块搜索路径存储在system模块的sys.path变量中。变量里包含当前目录， PYTHONPATH和由安装过程决定的默认目录。

## 模块制作

### <1>定义自己的模块

在Python中，每个Python文件都可以作为一个模块，模块的名字就是文件的名字。

比如有这样一个文件test.py，在test.py中定义了函数add

test.py

```
def add(a, b):  
    return a+b
```

### <2>调用自己定义的模块

那么在其他文件中就可以先import test，然后通过test.add(a,b)来调用了，当然也可以通过from test import add来引入

main.py

```
import test  
  
result = test.add(11, 22)  
print(result)
```

### <3>测试模块

在实际开中，当一个开发人员编写完一个模块后，为了让模块能够在项目中达到想要的效果，这个开发人员会自行在py文件中添加一些测试信息，例如：

test.py

```
def add(a,b):
    return a+b

# 用来进行测试
ret = add(12,22)
print('int test.py file,,,12+22=%d'%ret)
```

如果此时，在其他py文件中引入了此文件的话，想想看，测试的那段代码是否也会执行呢！

main.py

```
import test

result = test.add(11,22)
print(result)
```

运行现象：



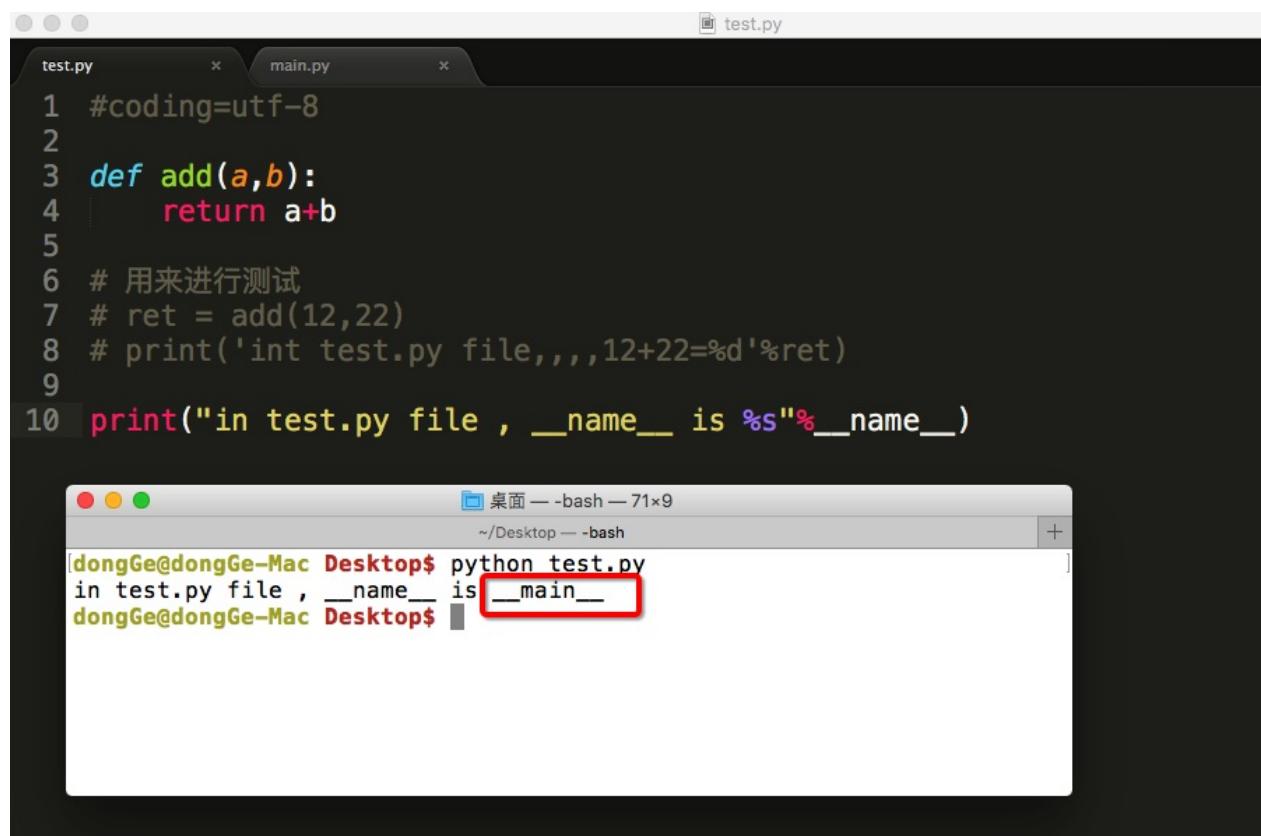
The screenshot shows a terminal window on a Mac OS X desktop. The title bar says "桌面 — bash — 71x9" and the path is "~/Desktop — bash". The command entered is "python main.py". The output shows the code from main.py being executed, including the print statement which outputs "33". The prompt "dongGe@dongGe-Mac Desktop\$" is visible at the end.

```
[dongGe@dongGe-Mac Desktop$ python main.py
int test.py file,,,12+22=34
33
dongGe@dongGe-Mac Desktop$ ]
```

至此，可发现test.py中的测试代码，应该是单独执行test.py文件时才应该执行的，不应该是其他的文件中引用而执行

为了解决这个问题，python在执行一个文件时有个变量 `__name__`

## 直接运行此文件



```
test.py      x  main.py      x  test.py
1 #coding=utf-8
2
3 def add(a,b):
4     return a+b
5
6 # 用来进行测试
7 # ret = add(12,22)
8 # print('int test.py file,,,12+22=%d'%ret)
9
10 print("in test.py file , __name__ is %s"%__name__)

[dongGe@dongGe-Mac Desktop$ python test.py
in test.py file , __name__ is __main__
dongGe@dongGe-Mac Desktop$ ]
```

## 在其他文件中import此文件

test.py

```
1 #coding=utf-8
2
3 def add(a,b):
4     return a+b
5
6 # 用来进行测试
7 # ret = add(12,22)
8 # print('int test.py file,,,12+22=%d'%ret)
9
10 print("in test.py file , __name__ is %s"%__name__)
```

Line 10, Column 51

main.py

```
1 #coding=utf-8
2 import test
3
4 # result = test.add(11,22)
5 # print(result)
6
```

main.py

```
dongGe@dongGe-Mac Desktop$ python main.py
in test.py file , __name__ is test
dongGe@dongGe-Mac Desktop$
```

## 总结：

- 可以根据`__name__`变量的结果能够判断出，是直接执行的python脚本还是被引入执行的，从而能够有选择性的执行测试代码

test.py

```
1 #coding=utf-8
2
3 def add(a,b):
4     return a+b
5
6 # 用来进行测试
7 if __name__ == 'main':
8     ret = add(12,22)
9     print('int test.py file,,,12+22=%d'%ret)
10
```

Line 7, Column 23

main.py

```
1 #coding=utf-8
2 import test
3
4 result = test.add(11,22)
5 print(result)
6
```

使用这种方法来  
有选择性的执行  
测试代码  
实际开发中常用

main.py

```
dongGe@dongGe-Mac Desktop$ python main.py
33
dongGe@dongGe-Mac Desktop$
```

## 模块中的 \_\_all\_\_

### 1. 没有 \_\_all\_\_

```

1 class Test(object):
2     def test(self):
3         print("----Test类中的test函数----")
4
5 def test1():
6     print("----test1函数----")
7
8 def test2():
9     print("----test2函数----")
~
```

test.py

```

1 from test import *
2
3 a = Test()
4 a.test()
5
6 test1()
7
8 test2()
~
```

main.py

```

python@ubuntu:~/Desktop/test$ python3 main.py
----Test类中的test函数----
----test1函数----
----test2函数----
python@ubuntu:~/Desktop/test$
```

### 2. 模块中有 \_\_all\_\_

```

1 __all__ = ["Test", "test1"]
2 class Test(object):
3     def test(self):
4         print("----Test类中的test函数----")
5
6 def test1():
7     print("----test1函数----")
8
9 def test2():
10    print("----test2函数----")
~
```

test.py

```
python@ubuntu:~/Desktop/test$ python3 main.py
----Test类中的test函数----
----test1函数----
Traceback (most recent call last):
  File "main.py", line 8, in <module>
    test2()
NameError: name 'test2' is not defined
python@ubuntu:~/Desktop/test$ █
```

## 总结

- 如果一个文件中有`__all__`变量，那么也就意味着这个变量中的元素，不  
会被`from xxx import *`时导入

# python中的包

## 1. 引入包

### 1.1 有2个模块功能有些联系

```
python@ubuntu:~/Desktop/test/08-包$ ls
recvmsg.py  sendmsg.py
python@ubuntu:~/Desktop/test/08-包$ █
```

### 1.2 所以将其放到同一个文件夹下

```
python@ubuntu:~/Desktop/test/08-包$ ls
msg
python@ubuntu:~/Desktop/test/08-包$ tree
.
└── msg
    ├── recvmsg.py
    └── sendmsg.py

1 directory, 2 files
python@ubuntu:~/Desktop/test/08-包$
```

### 1.3 使用import 文件.模块 的方式导入

```
python@ubuntu:~/Desktop/test/08-包$ ipython3
Python 3.5.2 (default, Jul 5 2016, 12:43:10)
Type "copyright", "credits" or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.

In [1]: import msg.sendmsg

In [2]: msg.sendmsg.sendmsg()
正在发送短信...
已经成功发送短信

In [3]: import msg.recvmsg

In [4]: msg.recvmsg.recvmsg()
接收到一条新信息
```

## 1.4 使用from 文件夹 import 模块 的方式导入

```
python@ubuntu:~/Desktop/test/08-包$ ipython3
Python 3.5.2 (default, Jul 5 2016, 12:43:10)
Type "copyright", "credits" or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.

In [1]: from msg import *
        没有导入文
        件夹msg下
        的模块
In [2]: sendmsg.sendmsg()
NameError                                     Traceback (most recent call last)
<ipython-input-2-40eca465700a> in <module>()
----> 1 sendmsg.sendmsg()

NameError: name 'sendmsg' is not defined

In [3]: recvmsg.recvmsg()
NameError                                     Traceback (most recent call last)
<ipython-input-3-2d1f9cdc8207> in <module>()
----> 1 recvmsg.recvmsg()

NameError: name 'recvmsg' is not defined

In [4]: ■
```

## 1.5 在msg文件夹下创建 \_\_init\_\_.py 文件

```
python@ubuntu:~/Desktop/test/08-包/msg$ ls
recvmsg.py  sendmsg.py
python@ubuntu:~/Desktop/test/08-包/msg$ touch __init__.py
python@ubuntu:~/Desktop/test/08-包/msg$ ls
__init__.py  recvmsg.py  sendmsg.py
python@ubuntu:~/Desktop/test/08-包/msg$
```

## 1.6 在 `__init__.py` 文件中写入

```
1 __all__ = ["sendmsg", "recvmsg"]
```

## 1.7 重新使用from 文件夹 import 模块 的方式导入

```
python@ubuntu:~/Desktop/test/08-包$ ipython3
Python 3.5.2 (default, Jul  5 2016, 12:43:10)
Type "copyright", "credits" or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help       -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.
```

```
In [1]: from msg import *          成功导入文
In [2]: sendmsg.sendmsg()           件夹msg下
正在发送短信...                   的所有模块
已经成功发送短信

In [3]: recvmsg.recvmsg()
接收到一条新信息

In [4]:
```

## 总结：

- 包将有联系的模块组织在一起，即放到同一个文件夹下，并且在这个文件夹创建一个名字为 `__init__.py` 文件，那么这个文件夹就称之为 包
- 有效避免模块名称冲突问题，让应用组织结构更加清晰

## 2. \_\_init\_\_.py 文件有什么用

\_\_init\_\_.py 控制着包的导入行为

### 2.1 \_\_init\_\_.py 为空

仅仅是把这个包导入，不会导入包中的模块

### 2.2 \_\_all\_\_

在 \_\_init\_\_.py 文件中，定义一个 \_\_all\_\_ 变量，它控制着 from 包名 import \* 时导入的模块

### 2.3 (了解)可以在 \_\_init\_\_.py 文件中编写内容

可以在这个文件中编写语句，当导入时，这些语句就会被执行

\_\_init\_\_.py文件

```
1 __all__ = ["sendmsg", "recvmsg"]
2 print("-----1-----")
3 def test():
4     print("-----2-----")
```

```
python@ubuntu:~/Desktop/test/08-包$ ipython3
Python 3.5.2 (default, Jul  5 2016, 12:43:10)
Type "copyright", "credits" or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?           -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help        -> Python's own help system.
object?    -> Details about 'object', use 'object??' for extra details.

In [1]: import msg
-----1-----

In [2]: test()

NameError                                 Traceback (most recent call last)
<ipython-input-2-ea594c21b25d> in <module>()
----> 1 test()

NameError: name 'test' is not defined

In [3]: msg.test()
-----2-----

In [4]:
```

### 3. 扩展：嵌套的包

假定我们的包的例子有如下的目录结构：

```
Phone/
    __init__.py
    common_util.py
Voicedta/
    __init__.py
    Pots.py
    Isdn.py
Fax/
    __init__.py
    G3.py
Mobile/
    __init__.py
    Analog.py
    igital.py
Pager/
    __init__.py
```

### Numeric.py

Phone 是最顶层的包，Voicedta 等是它的子包。我们可以这样导入子包：

```
import Phone.Mobile.Analog  
Phone.Mobile.Analog.dial()
```

你也可使用 from-import 实现不同需求的导入

第一种方法是只导入顶层的子包，然后使用属性/点操作符向下引用子包树：

```
from Phone import Mobile  
Mobile.Analog.dial('555-1212')
```

此外，我们可以还引用更多的子包：

```
from Phone.Mobile import Analog  
Analog.dial('555-1212')
```

事实上，你可以一直沿子包的树状结构导入：

```
from Phone.Mobile.Analog import dial  
dial('555-1212')
```

在我们上边的目录结构中，我们可以发现很多的 `__init__.py` 文件。这些是初始化模块，from-import 语句导入子包时需要用到它。如果没有用到，他们可以是空文件。

包同样支持 from-import all 语句：

```
from package.module import *
```

然而，这样的语句会导入哪些文件取决于操作系统的文件系统。所以我们  
在 `__init__.py` 中加入 `__all__` 变量。该变量包含执行这样的语句时应  
该导入的模块的名字。它由一个模块名字符串列表组成.。

# 模块发布

## 模块发布

1.mymodule目录结构体如下：

```
.  
├── setup.py  
├── suba  
│   ├── aa.py  
│   ├── bb.py  
│   └── __init__.py  
└── subb  
    ├── cc.py  
    ├── dd.py  
    └── __init__.py
```

2.编辑setup.py文件

py\_modules需指明所需包含的py文件

```
from distutils.core import setup  
  
setup(name="dongGe", version="1.0", description="dongGe's module")
```

3.构建模块

python setup.py build

构建后目录结构

```
.  
├── build  
|   └── lib.linux-i686-2.7
```

```
|   └── suba
|       ├── aa.py
|       ├── bb.py
|       └── __init__.py
└── subb
    ├── cc.py
    ├── dd.py
    └── __init__.py
└── setup.py
└── suba
    ├── aa.py
    ├── bb.py
    └── __init__.py
└── subb
    ├── cc.py
    ├── dd.py
    └── __init__.py
```

#### 4.生成发布压缩包

```
python setup.py sdist
```

打包后,生成最终发布压缩包dongGe-1.0.tar.gz , 目录结构

```
.
├── build
│   └── lib.linux-i686-2.7
│       ├── suba
│       │   ├── aa.py
│       │   ├── bb.py
│       │   └── __init__.py
│       └── subb
│           ├── cc.py
│           ├── dd.py
│           └── __init__.py
└── dist
    └── dongGe-1.0.tar.gz
└── MANIFEST
```

```
|── setup.py  
|── suba  
|   ├── aa.py  
|   ├── bb.py  
|   └── __init__.py  
└── subb  
    ├── cc.py  
    ├── dd.py  
    └── __init__.py
```

# 模块安装、使用

## 1. 安装的方式

1. 找到模块的压缩包
2. 解压
3. 进入文件夹
4. 执行命令 `python setup.py install`

注意：

- 如果在install的时候，执行目录安装，可以使用 `python setup.py install --prefix=安装路径`

## 2. 模块的引入

在程序中，使用`from import`即可完成对安装的模块使用

```
from 模块名 import 模块名或者*
```



## 给程序传参数

```
import sys  
  
print(sys.argv)
```

运行结果：

```
python@ubuntu:~/Desktop/test/12-加强练习$ ls  
01-传递参数.py  
python@ubuntu:~/Desktop/test/12-加强练习$ python3 01-传递参数.py  
['01-传递参数.py']  
python@ubuntu:~/Desktop/test/12-加强练习$ python3 01-传递参数.py haha 1 2 3 44  
['01-传递参数.py', 'haha', '1', '2', '3', '44']  
python@ubuntu:~/Desktop/test/12-加强练习$
```

# 列表推导式

所谓的列表推导式，就是指的轻量级循环创建列表

## 1. 基本的方式

```
In [2]: a = [x for x in range(4)]  
In [3]: a  
Out[3]: [0, 1, 2, 3]  
  
In [4]: a = [x for x in range(3,4)]  
In [5]: a  
Out[5]: [3]  
  
In [6]: a = [x for x in range(3,19)]  
In [7]: a  
Out[7]: [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]  
In [8]: a = [x for x in range(3,19,2)]  
In [9]: a  
Out[9]: [3, 5, 7, 9, 11, 13, 15, 17]
```

## 2. 在循环的过程中使用if

```
In [10]: a = [x for x in range(3,10) if x%2==0]  
In [11]: a  
Out[11]: [4, 6, 8]  
  
In [12]: a = [x for x in range(3,10) if x%2!=0]  
In [13]: a  
Out[13]: [3, 5, 7, 9]  
  
In [14]: a = [x for x in range(3,10)]  
In [15]: a  
Out[15]: [3, 4, 5, 6, 7, 8, 9]
```

## 3. 2个for循环

```
In [29]: a = [(x,y) for x in range(1,3) for y in range(3)]  
In [30]: a  
Out[30]: [(1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

## 4. 3个for循环

```
In [26]: a = [(x,y,z) for x in range(1,3) for y in range(3) for z in range(4,6)]  
In [27]: a  
Out[27]:  
[(1, 0, 4),  
(1, 0, 5),  
(1, 1, 4),  
(1, 1, 5),  
(1, 2, 4),  
(1, 2, 5),  
(2, 0, 4),  
(2, 0, 5),  
(2, 1, 4),  
(2, 1, 5),  
(2, 2, 4),  
(2, 2, 5)]
```

## 练习

- 生成一个[[1,2,3],[4,5,6]....]的列表最大值在100以内
- 请写出一段 Python 代码实现分组一个 list 里面的元素,比如 [1,2,3,...100] 变成 [[1,2,3],[4,5,6]....]

## set、list、tuple

### set是集合类型

```
In [45]: a = set()  
In [46]: type(a)  
Out[46]: set  
  
In [47]:  
In [47]: b = [1,2,3,12,3,1,3]  
In [48]: b  
Out[48]: [1, 2, 3, 12, 3, 1, 3]  
  
In [49]:  
In [49]: c = set(b)  
In [50]: type(c)  
Out[50]: set  
  
In [51]: c  
Out[51]: {1, 2, 3, 12}  
  
In [52]:  
In [52]: d = list(c)  
In [53]: d  
Out[53]: [1, 2, 3, 12]  
In [54]: █
```

set、list、tuple之间可以相互转换

```
In [52]: d = list(c)
In [53]: d
Out[53]: [1, 2, 3, 12]
In [54]:
In [54]: e = tuple(d)
In [55]: e
Out[55]: (1, 2, 3, 12)
In [56]:
In [56]: f = list(e)
In [57]: f
Out[57]: [1, 2, 3, 12]
In [58]:
In [58]: g = set(e)
In [59]: g
Out[59]: {1, 2, 3, 12}
```

使用set，可以快速的完成对list中的元素去重复的功能

(3) 请简单解释 Python 中 static method (静态方法) 和 class method (类方法), 并将以下代码填写完整:

```
class A(object):
    def foo(self, x):
        print "executing foo(%s,%s)"%(self, x)

    @classmethod
    def class_foo(cls, x):
        print "executing class_foo(%s,%s)"%(cls, x)

    @staticmethod
    def static_foo(x):
        print "executing static_foo(%s)"%x

a = A()
#调用 foo 函数, 参数传入 1
```

---

```
#调用 class_foo 函数, 参数传入 1
```

---

```
#调用 static_foo 函数, 参数传入 1
```

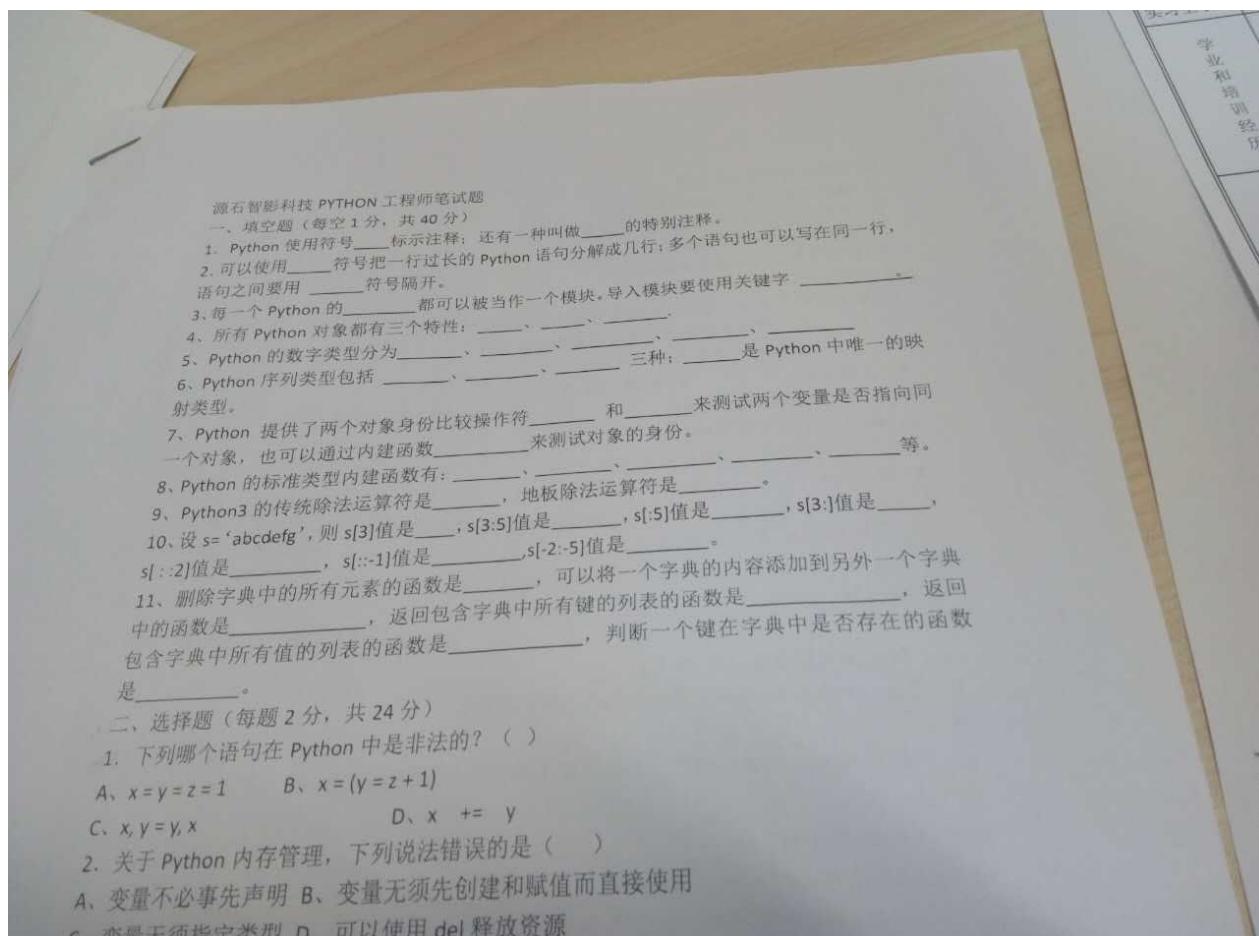
## 四、程序题

1、请书写一个函数，用于替换某个字符串的一个或某几个字串。

函数原型 strreplace(str , oldString, newString)

例如：

```
>>pstr = "Hello World!";
>>afterReplaceStr = strreplace(pstr, " World", " Tom");
那么 afterReplaceStr 的值为 "Hello Tom!"
```



1. 在 Python 中,类和对象有什么区别?对象如何访问类的方法? 创建一个对象时做了什么?
2. 请写出一段 Python 代码实现分组一个 list 里面的元素,比如 [1,2,3,...100] 变成 [[1,2,3],[4,5,6],...]
3. 请写出一段 Python 代码实现删除一个 list 里面的重复元素
4. 设计实现遍历目录与子目录,抓取.pyc 文件
5. 写出一个函数,给定参数 n,生成含有 n 个元素值为 1~n 的数 组,元素顺序随机,但值不重复
6. 在不用其他变量的情况下, 交换a、b变量的值
7. 如何在一个 function 里设置一个全局变量
8. 请问如下代码会输出什么?

```
def extendList(val, list=[]):  
    list.append(val)  
    return list  
list1 = extendList(10)  
list2 = extendList(123, ['a', 'b', 'c'])  
list3 = extendList('a')  
print "list1 = %s" % list1  
print "list2 = %s" % list2  
print "list3 = %s" % list3
```



# 打飞机代码：搭建界面



```
#coding=utf-8
import pygame

...
1. 搭建界面，主要完成窗口和背景图的显示
...

if __name__ == "__main__":
    #1. 创建一个窗口，用来显示内容
    screen = pygame.display.set_mode((480, 890), 0, 32)

    #2. 创建一个和窗口大小的图片，用来充当背景
    background = pygame.image.load("./feiji/background.png").convert()

    #3. 把背景图片放到窗口中显示
    while True:
        screen.blit(background, (0, 0))
        pygame.display.update()
```

# 打飞机代码：检测键盘

```
#coding=utf-8
import pygame
from pygame.locals import *

...
2. 用来检测事件，比如按键操作
...

if __name__ == "__main__":
    #1. 创建一个窗口，用来显示内容
    screen = pygame.display.set_mode((480, 890), 0, 32)

    #2. 创建一个和窗口大小的图片，用来充当背景
    background = pygame.image.load("./feiji/background.png").cor

    #3. 把背景图片放到窗口中显示
    while True:

        #设定需要显示的背景图
        screen.blit(background, (0, 0))

        #获取事件，比如按键等
        for event in pygame.event.get():

            #判断是否是点击了退出按钮
            if event.type == QUIT:
                print("exit")
                exit()
            #判断是否是按下了键
            elif event.type == KEYDOWN:
                #检测按键是否是a或者left
                if event.key == K_a or event.key == K_LEFT:
```

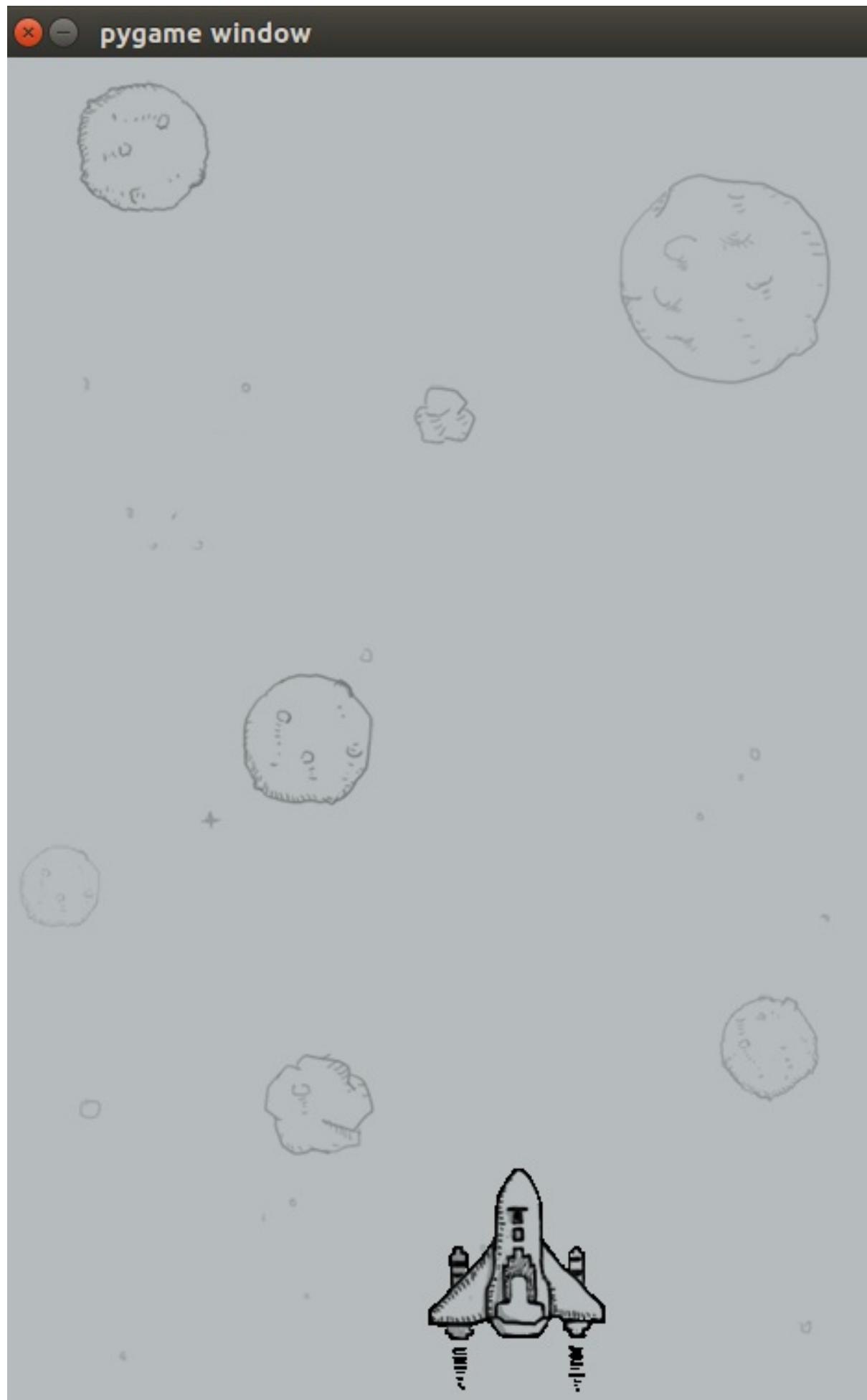
```
    print('left')

    #检测按键是否是d或者right
    elif event.key == K_d or event.key == K_RIGHT:
        print('right')

    #检测按键是否是空格键
    elif event.key == K_SPACE:
        print('space')

    #更新需要显示的内容
    pygame.display.update()
```

## 打飞机代码：显示、控制玩具飞机-面向过程



```
#coding=utf-8
import pygame
from pygame.locals import *

...
3. 使用面向过程的方式来显示一个飞机，并控制其左右移动
...

if __name__ == "__main__":
    #1. 创建一个窗口，用来显示内容
    screen = pygame.display.set_mode((480, 890), 0, 32)

    #2. 创建一个和窗口大小的图片，用来充当背景
    background = pygame.image.load("./feiji/background.png").convert()

    #测试，用来创建一个玩家飞机的图片
    hero = pygame.image.load("./feiji/hero.gif").convert()

    #用来保存飞机的x, y坐标
    x=0
    y=0

    #3. 把背景图片放到窗口中显示
    while True:
        screen.blit(background,(0,0))

        #设定需要显示的飞机图片
        screen.blit(hero,(x,y))

        #判断是否是点击了退出按钮
        for event in pygame.event.get():
            # print(event.type)
            if event.type == QUIT:
                print("exit")
                exit()
```

```
    elif event.type == KEYDOWN:  
        if event.key == K_a or event.key == K_LEFT:  
            print('left')  
            #控制飞机让其向左移动  
            x-=5  
        elif event.key == K_d or event.key == K_RIGHT:  
            print('right')  
            #控制飞机让其向右移动  
            x+=5  
        elif event.key == K_SPACE:  
            print('space')  
  
    pygame.display.update()
```

# 打飞机代码：显示、控制玩具飞机-面向对象



```
#coding=utf-8
import pygame
from pygame.locals import *
...
...
```

4. 使用面向对象的方式显示飞机，以及控制其左右移动

练一下：接下来要做的任务：

1. 实现飞机在你想要的位置显示
  2. 实现按键控制飞机移动
  3. 实现按下空格键的时候，显示一颗子弹
- ...

```
class HeroPlane(object):  
  
    def __init__(self, screen):  
  
        #设置飞机默认的位置  
        self.x = 230  
        self.y = 600  
  
        #设置要显示内容的窗口  
        self.screen = screen  
  
        #用来保存英雄飞机需要的图片名字  
        self.imageName = "./feiji/hero.gif"  
  
        #根据名字生成飞机图片  
        self.image = pygame.image.load(self.imageName).convert()  
  
        #用来保存英雄飞机发射出的所有子弹  
        self.bullet = []  
  
  
    def display(self):  
        self.screen.blit(self.image, (self.x, self.y))  
  
    def moveLeft(self):  
        self.x -= 10  
  
    def moveRight(self):  
        self.x += 10  
  
    def sheBullet(self):  
        pass
```

```
if __name__ == "__main__":
    #1. 创建一个窗口, 用来显示内容
    screen = pygame.display.set_mode((480, 890), 0, 32)

    #2. 创建一个和窗口大小的图片, 用来充当背景
    background = pygame.image.load("./feiji/background.png").convert()

    #3. 创建一个飞机对象
    heroPlane = HeroPlane(screen)

    #3. 把背景图片放到窗口中显示
    while True:
        screen.blit(background, (0, 0))

        heroPlane.display()

        #判断是否是点击了退出按钮
        for event in pygame.event.get():
            # print(event.type)
            if event.type == QUIT:
                print("exit")
                exit()
            elif event.type == KEYDOWN:
                if event.key == K_a or event.key == K_LEFT:
                    print('left')
                    heroPlane.moveLeft()
                    #控制飞机让其向左移动
                elif event.key == K_d or event.key == K_RIGHT:
                    print('right')
                    heroPlane.moveRight()
                elif event.key == K_SPACE:
                    print('space')

        pygame.display.update()
```



# 打飞机代码：玩家飞机发射子弹

```
#coding=utf-8
import pygame
from pygame.locals import *
...
实现玩家飞机发射子弹
```

接下来要做的任务：

1. 实现飞机在你想要的位置显示
2. 实现按键控制飞机移动
3. 实现按下空格键的时候，显示一颗子弹

```
...
class HeroPlane(object):

    def __init__(self, screen):
        #设置飞机默认的位置
        self.x = 230
        self.y = 600

        #设置要显示内容的窗口
        self.screen = screen

        self.imageName = "./feiji/hero.gif"
        self.image = pygame.image.load(self.imageName).convert()

        #用来存储英雄飞机发射的所有子弹
        self.bulletList = []

    def display(self):
        self.screen.blit(self.image, (self.x, self.y))
```

```
for bullet in self.bulletList:  
    bullet.display()  
    bullet.move()  
  
#修改所有子弹的位置  
# for bullet in self.bulletList:  
#     bullet.y -= 2  
  
def moveLeft(self):  
    self.x -= 10  
  
def moveRight(self):  
    self.x += 10  
  
def sheBullet(self):  
    newBullet = Bullet(self.x, self.y, self.screen)  
    self.bulletList.append(newBullet)  
  
class Bullet(object):  
    def __init__(self, x, y, screen):  
        self.x = x+40  
        self.y = y-20  
        self.screen = screen  
        self.image = pygame.image.load("./feiji/bullet-3.gif").convert()  
  
    def move(self):  
        self.y -= 2  
  
    def display(self):  
        self.screen.blit(self.image, (self.x, self.y))  
  
if __name__ == "__main__":  
    #1. 创建一个窗口, 用来显示内容  
    screen = pygame.display.set_mode((480, 890), 0, 32)  
  
    #2. 创建一个和窗口大小的图片, 用来充当背景
```

```
background = pygame.image.load("./feiji/background.png").convert()
#3. 创建一个飞机对象
heroPlane = HeroPlane(screen)

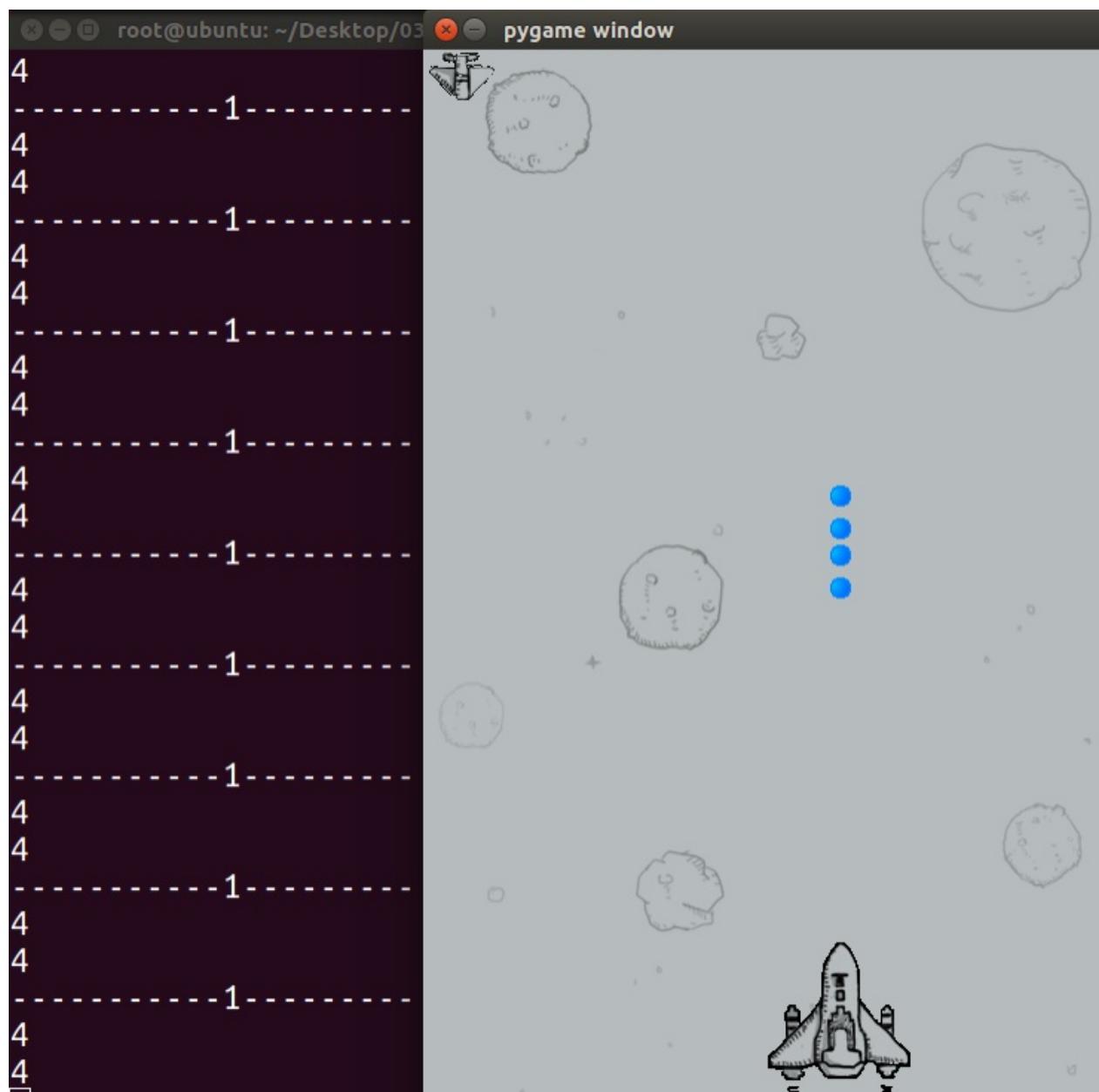
#3. 把背景图片放到窗口中显示
while True:
    screen.blit(background,(0,0))

    heroPlane.display()

#判断是否是点击了退出按钮
for event in pygame.event.get():
    # print(event.type)
    if event.type == QUIT:
        print("exit")
        exit()
    elif event.type == KEYDOWN:
        if event.key == K_a or event.key == K_LEFT:
            print('left')
            heroPlane.moveLeft()
            #控制飞机让其向左移动
        elif event.key == K_d or event.key == K_RIGHT:
            print('right')
            heroPlane.moveRight()
        elif event.key == K_SPACE:
            print("space")
            heroPlane.sh射Bullet()

    pygame.display.update()
```

## 打飞机代码：显示敌机



```
#coding=utf-8
import pygame
from pygame.locals import *

...
    显示敌人飞机
...
```

```
class HeroPlane(object):

    def __init__(self, screen):
        #设置飞机默认的位置
        self.x = 230
        self.y = 600

        #设置要显示内容的窗口
        self.screen = screen

        self.imageName = "./feiji/hero.gif"
        self.image = pygame.image.load(self.imageName).convert()

        #用来存储英雄飞机发射的所有子弹
        self.bulletList = []

    def display(self):
        self.screen.blit(self.image, (self.x, self.y))

        for bullet in self.bulletList:
            bullet.display()
            bullet.move()
        #修改所有子弹的位置
        # for bullet in self.bulletList:
        #     bullet.y -= 2

    def moveLeft(self):
        self.x -= 10

    def moveRight(self):
        self.x += 10

    def sheBullet(self):
        newBullet = Bullet(self.x, self.y, self.screen)
        self.bulletList.append(newBullet)

class Bullet(object):
```

```
def __init__(self,x,y,screen):
    self.x = x+40
    self.y = y-20
    self.screen = screen
    self.image = pygame.image.load("./feiji/bullet-3.gif").convert()

def move(self):
    self.y -= 2

def display(self):
    self.screen.blit(self.image,(self.x,self.y))

class EnemyPlane(object):

    def __init__(self,screen):

        #设置飞机默认的位置
        self.x = 0
        self.y = 0

        #设置要显示内容的窗口
        self.screen = screen

        self.imageName = "./feiji/enemy-1.gif"
        self.image = pygame.image.load(self.imageName).convert()

        #用来存储敌人飞机发射的所有子弹
        self.bulletList = []

    def display(self):
        self.screen.blit(self.image,(self.x,self.y))

if __name__ == "__main__":
    #1. 创建一个窗口, 用来显示内容
    screen = pygame.display.set_mode((480,890),0,32)
```

```
#2. 创建一个和窗口大小的图片，用来充当背景
background = pygame.image.load("./feiji/background.png").convert()

#3. 创建一个飞机对象
heroPlane = HeroPlane(screen)

#4. 创建一个敌人飞机
enemyPlane = EnemyPlane(screen)

#3. 把背景图片放到窗口中显示
while True:
    screen.blit(background,(0,0))

    heroPlane.display()
    enemyPlane.display()

    #判断是否是点击了退出按钮
    for event in pygame.event.get():
        # print(event.type)
        if event.type == QUIT:
            print("exit")
            exit()
        elif event.type == KEYDOWN:
            if event.key == K_a or event.key == K_LEFT:
                print('left')
                heroPlane.moveLeft()
                #控制飞机让其向左移动
            elif event.key == K_d or event.key == K_RIGHT:
                print('right')
                heroPlane.moveRight()
            elif event.key == K_SPACE:
                print("space")
                heroPlane.sheBullet()

    pygame.display.update()
```



# 打飞机代码：优化代码

```
#coding=utf-8
import pygame
from pygame.locals import *

...
    优化代码：优化发射出的子弹
...

class HeroPlane(object):

    def __init__(self, screen):
        #设置飞机默认的位置
        self.x = 230
        self.y = 600

        #设置要显示内容的窗口
        self.screen = screen

        self.imageName = "./feiji/hero.gif"
        self.image = pygame.image.load(self.imageName).convert()

        #用来存储英雄飞机发射的所有子弹
        self.bulletList = []

    def display(self):
        #更新飞机的位置
        self.screen.blit(self.image, (self.x, self.y))

        #判断一下子弹的位置是否越界，如果是，那么就要删除这颗子弹
        #

...
```

```
#这种方法会漏掉很多需要删除的数据
# for i in self.bulletList:
#     if i.y<0:
#         self.bulletList.remove(i)

#用来存放需要删除的对象信息
needDelItemList = []

#保存需要删除的对象
for i in self.bulletList:
    if i.judge():
        needDelItemList.append(i)
#删除self.bulletList中需要删除的对象
for i in needDelItemList:
    self.bulletList.remove(i)

#因为needDelItemList也保存了刚刚删除的对象的引用，所以可以删除整个列表中的引用就不存在了，也可以不调用下面的代码，因为needDelItemList当这个方法的调用结束时，这个局部变量也就不存在了
# del needDelItemList

#更新及这架飞机发射出的所有子弹的位置
for bullet in self.bulletList:
    bullet.display()
    bullet.move()

#修改所有子弹的位置
# for bullet in self.bulletList:
#     bullet.y -= 2

def moveLeft(self):
    self.x -= 10

def moveRight(self):
    self.x += 10

def sheBullet(self):
    newBullet = Bullet(self.x, self.y, self.screen)
```

```
        self.bulletList.append(newBullet)

class Bullet(object):
    def __init__(self,x,y,screen):
        self.x = x+40
        self.y = y-20
        self.screen = screen
        self.image = pygame.image.load("./feiji/bullet-3.gif").convert()

    def move(self):
        self.y -= 2

    def display(self):
        self.screen.blit(self.image,(self.x,self.y))

    def judge(self):
        if self.y<0:
            return True
        else:
            return False

class EnemyPlane(object):

    def __init__(self,screen):

        #设置飞机默认的位置
        self.x = 0
        self.y = 0

        #设置要显示内容的窗口
        self.screen = screen

        self.imageName = "./feiji/enemy-1.gif"
        self.image = pygame.image.load(self.imageName).convert()

        #用来存储敌人飞机发射的所有子弹
        self.bulletList = []
```

```
def display(self):
    self.screen.blit(self.image,(self.x,self.y))

if __name__ == "__main__":
    #1. 创建一个窗口, 用来显示内容
    screen = pygame.display.set_mode((480,890),0,32)

    #2. 创建一个和窗口大小的图片, 用来充当背景
    background = pygame.image.load("./feiji/background.png").convert()

    #3. 创建一个飞机对象
    heroPlane = HeroPlane(screen)

    #4. 创建一个敌人飞机
    enemyPlane = EnemyPlane(screen)

    #3. 把背景图片放到窗口中显示
    while True:
        screen.blit(background,(0,0))

        heroPlane.display()
        enemyPlane.display()

        #判断是否是点击了退出按钮
        for event in pygame.event.get():
            # print(event.type)
            if event.type == QUIT:
                print("exit")
                exit()
            elif event.type == KEYDOWN:
                if event.key == K_a or event.key == K_LEFT:
                    print('left')
                    heroPlane.moveLeft()
                    #控制飞机让其向左移动
                elif event.key == K_d or event.key == K_RIGHT:
                    print('right')
```

```
    heroPlane.moveRight()
elif event.key == K_SPACE:
    print("space")
    heroPlane.sh射Bullet()

pygame.display.update()
```

# 打飞机代码：让敌机移动

```
#coding=utf-8
import time
import random
import pygame
from pygame.locals import *

class HeroPlane(object):

    def __init__(self,screen):

        #设置飞机默认的位置
        self.x = 230
        self.y = 600

        #设置要显示内容的窗口
        self.screen = screen

        self.imageName = "./feiji/hero.gif"
        self.image = pygame.image.load(self.imageName).convert()

        #用来存储英雄飞机发射的所有子弹
        self.bulletList = []

    def display(self):
        #更新飞机的位置
        self.screen.blit(self.image,(self.x,self.y))

        #判断一下子弹的位置是否越界，如果是，那么就要删除这颗子弹
        #
        #这种方法会漏掉很多需要删除的数据
        # for i in self.bulletList:
        #     if i.y<0:
```

```
#           self.bulletList.remove(i)

#存放需要删除的对象信息
needDelItemList = []

for i in self.bulletList:
    if i.judge():
        needDelItemList.append(i)

for i in needDelItemList:
    self.bulletList.remove(i)

# del needDelItemList

#更新及这架飞机发射出的所有子弹的位置
for bullet in self.bulletList:
    bullet.display()
    bullet.move()

#修改所有子弹的位置
# for bullet in self.bulletList:
#     bullet.y -= 2

def moveLeft(self):
    self.x -= 10

def moveRight(self):
    self.x += 10

def sheBullet(self):
    newBullet = Bullet(self.x, self.y, self.screen)
    self.bulletList.append(newBullet)

class Bullet(object):
    def __init__(self, x, y, screen):
        self.x = x+40
        self.y = y-20
        self.screen = screen
```

```
    self.image = pygame.image.load("./feiji/bullet-3.gif").convert()

    def move(self):
        self.y -= 2

    def display(self):
        self.screen.blit(self.image,(self.x,self.y))

    def judge(self):
        if self.y<0:
            return True
        else:
            return False

class EnemyPlane(object):

    def __init__(self,screen):

        #设置飞机默认的位置
        self.x = 0
        self.y = 0

        #设置要显示内容的窗口
        self.screen = screen

        self.imageName = "./feiji/enemy-1.gif"
        self.image = pygame.image.load(self.imageName).convert()

        #用来存储敌人飞机发射的所有子弹
        self.bulletList = []

        self.direction = "right"

    def display(self):
        #更新飞机的位置
        self.screen.blit(self.image,(self.x,self.y))

        #判断一下子弹的位置是否越界，如果是，那么就要删除这颗子弹
```



```
#1. 创建一个窗口，用来显示内容
screen = pygame.display.set_mode((480, 890), 0, 32)

#2. 创建一个和窗口大小的图片，用来充当背景
background = pygame.image.load("./feiji/background.png").cor

#3. 创建一个飞机对象
heroPlane = HeroPlane(screen)

#4. 创建一个敌人飞机
enemyPlane = EnemyPlane(screen)

#3. 把背景图片放到窗口中显示
while True:
    screen.blit(background, (0, 0))

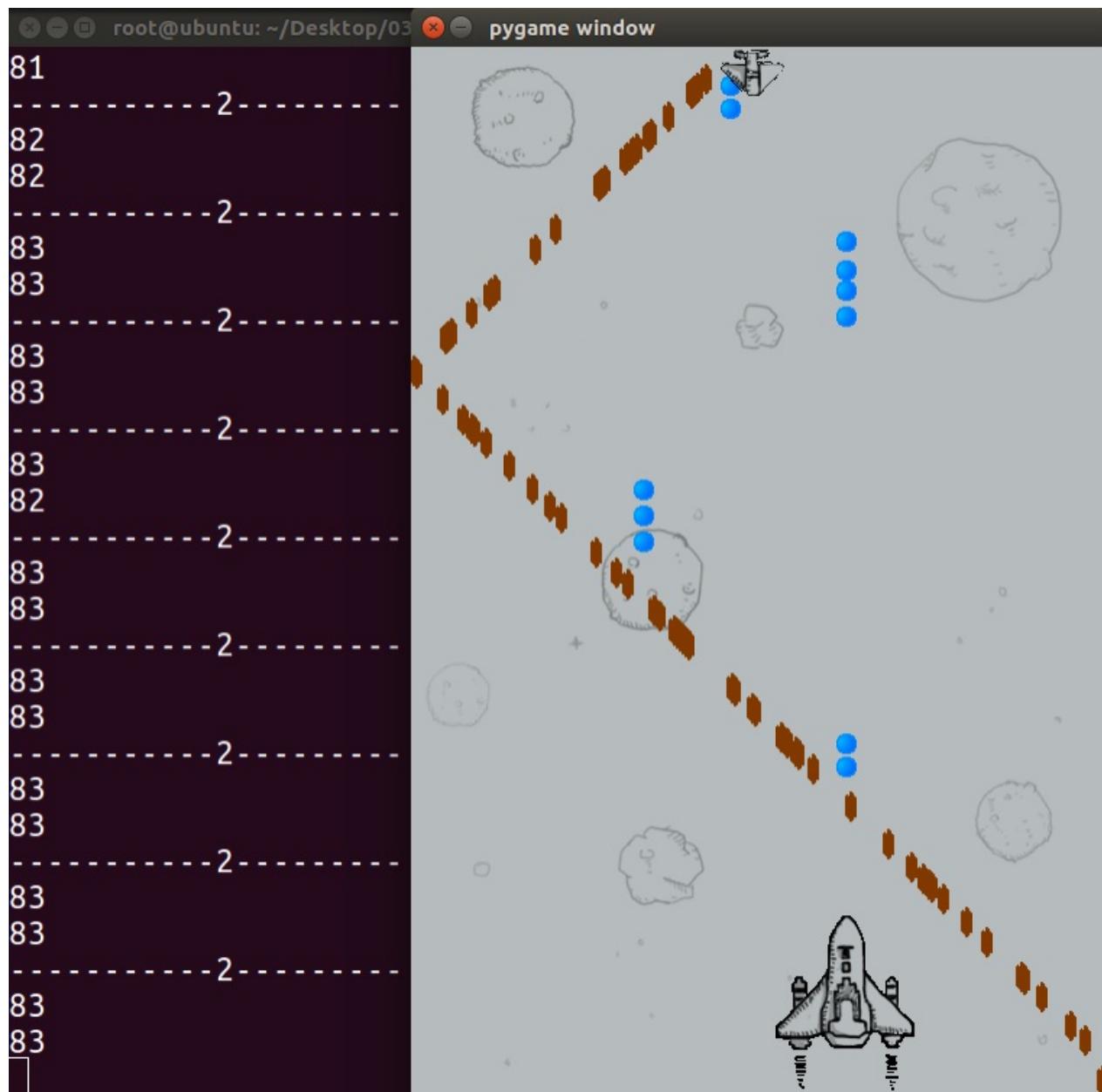
    heroPlane.display()

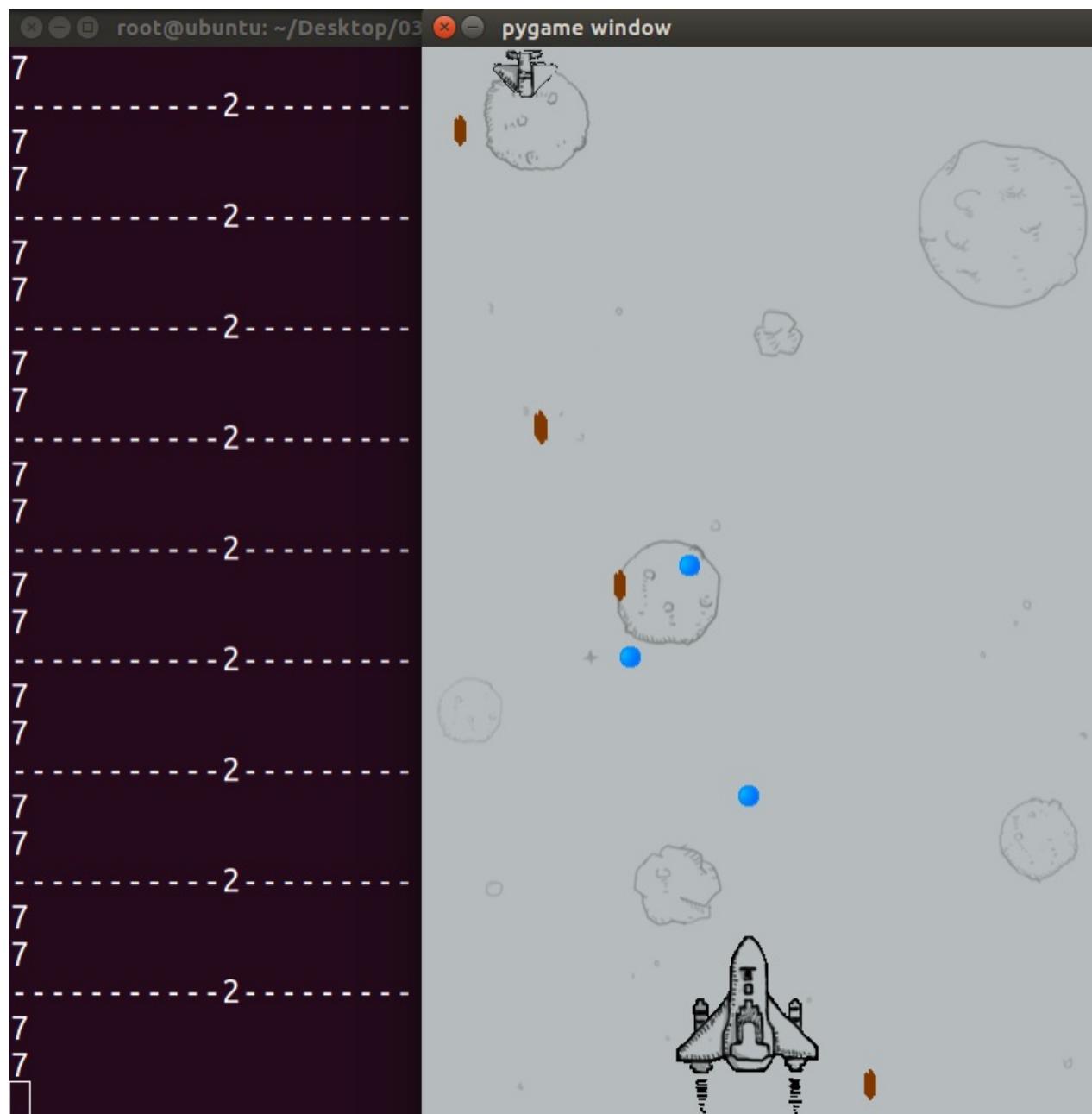
    enemyPlane.move()
    enemyPlane.display()

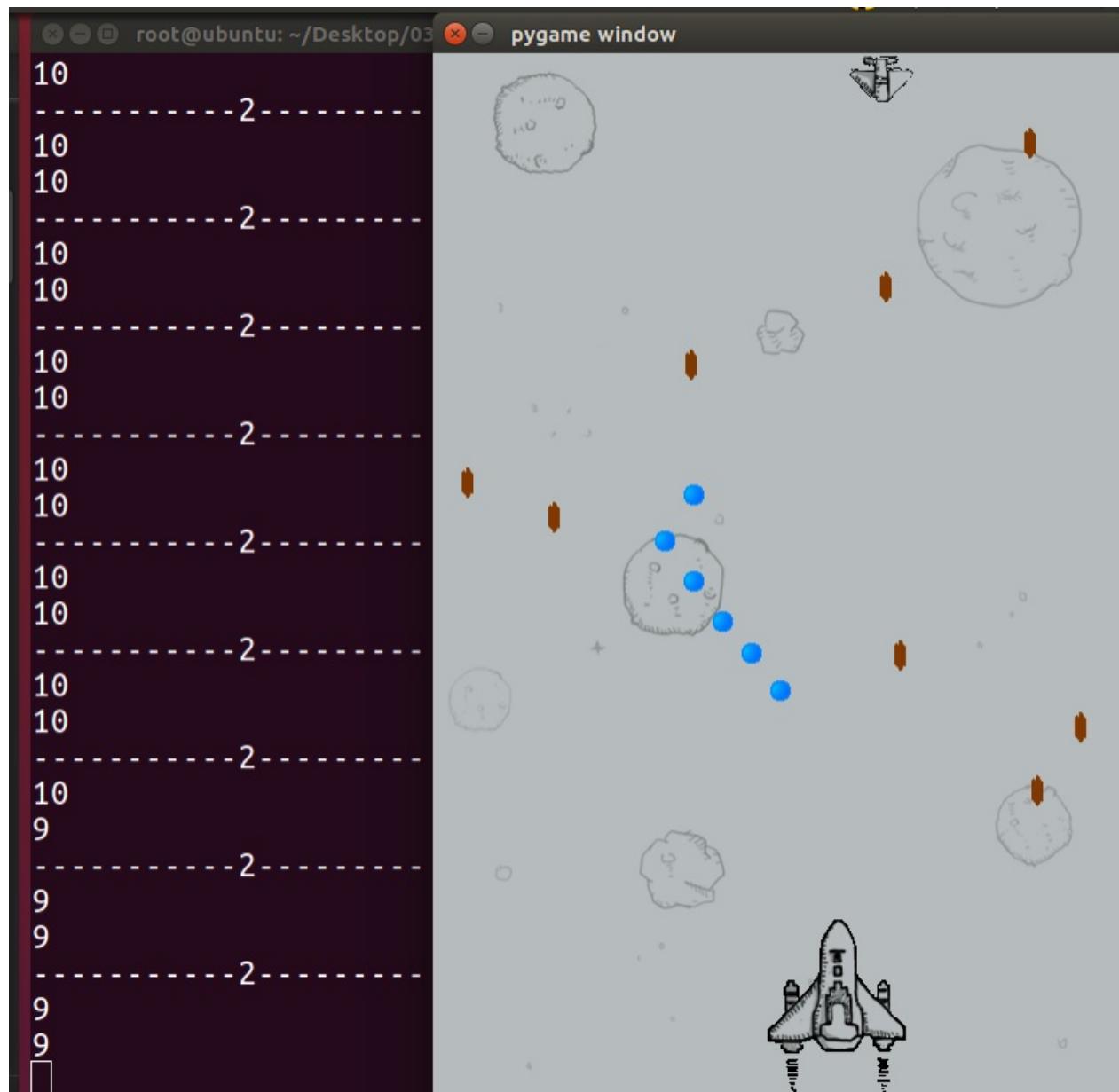
#判断是否是点击了退出按钮
for event in pygame.event.get():
    # print(event.type)
    if event.type == QUIT:
        print("exit")
        exit()
    elif event.type == KEYDOWN:
        if event.key == K_a or event.key == K_LEFT:
            print('left')
            heroPlane.moveLeft()
            #控制飞机让其向左移动
        elif event.key == K_d or event.key == K_RIGHT:
            print('right')
            heroPlane.moveRight()
        elif event.key == K_SPACE:
            print("space")
```

```
heroPlane.shootBullet()  
  
#通过延时的方式，来降低这个while循环的循环速度，从而降低了cpu占用  
time.sleep(0.01)  
  
pygame.display.update()
```

# 打飞机代码：敌机发射子弹







```
#coding=utf-8
import time
import random
import pygame
from pygame.locals import *

class HeroPlane(object):

    def __init__(self,screen):
```

```
#设置飞机默认的位置
self.x = 230
self.y = 600

#设置要显示内容的窗口
self.screen = screen

self.imageName = "./feiji/hero.gif"
self.image = pygame.image.load(self.imageName).convert()

#用来存储英雄飞机发射的所有子弹
self.bulletList = []

def display(self):
    #更新飞机的位置
    self.screen.blit(self.image,(self.x,self.y))

    #判断一下子弹的位置是否越界，如果是，那么就要删除这颗子弹
    #
    #这种方法会漏掉很多需要删除的数据
    # for i in self.bulletList:
    #     if i.y<0:
    #         self.bulletList.remove(i)

    #存放需要删除的对象信息
needDelItemList = []

for i in self.bulletList:
    if i.judge():
        needDelItemList.append(i)

for i in needDelItemList:
    self.bulletList.remove(i)

# del needDelItemList

#更新及这架飞机发射出的所有子弹的位置
for bullet in self.bulletList:
```

```
        bullet.display()
        bullet.move()

#修改所有子弹的位置
# for bullet in self.bulletList:
#     bullet.y -= 2

def moveLeft(self):
    self.x -= 10

def moveRight(self):
    self.x += 10

def sheBullet(self):
    newBullet = Bullet(self.x, self.y, self.screen)
    self.bulletList.append(newBullet)

class Bullet(object):
    def __init__(self,x,y,screen):
        self.x = x+40
        self.y = y-20
        self.screen = screen
        self.image = pygame.image.load("./feiji/bullet-3.gif").convert()

    def move(self):
        self.y -= 2

    def display(self):
        self.screen.blit(self.image,(self.x,self.y))

    def judge(self):
        if self.y<0:
            return True
        else:
            return False

class EnemyPlane(object):
```

```
def __init__(self,screen):  
  
    #设置飞机默认的位置  
    self.x = 0  
    self.y = 0  
  
    #设置要显示内容的窗口  
    self.screen = screen  
  
    self.imageName = "./feiji/enemy-1.gif"  
    self.image = pygame.image.load(self.imageName).convert()  
  
    #用来存储敌人飞机发射的所有子弹  
    self.bulletList = []  
  
    self.direction = "right"  
  
def display(self):  
    #更新飞机的位置  
    self.screen.blit(self.image,(self.x,self.y))  
  
    #判断一下子弹的位置是否越界，如果是，那么就要删除这颗子弹  
    #  
    #这种方法会漏掉很多需要删除的数据  
    # for i in self.bulletList:  
    #     if i.y<0:  
    #         self.bulletList.remove(i)  
  
    #存放需要删除的对象信息  
    needDelItemList = []  
  
    for i in self.bulletList:  
        if i.judge():  
            needDelItemList.append(i)  
    for i in needDelItemList:  
        self.bulletList.remove(i)  
  
    # del needDelItemList
```

```
#更新及这架飞机发射出的所有子弹的位置
for bullet in self.bulletList:
    bullet.display()
    bullet.move()

def move(self):
    #如果碰到了右边的边界，那么就往左走，如果碰到了左边的边界，那么就往右走
    if self.direction == "right":
        self.x += 2
    elif self.direction == "left":
        self.x -= 2

    if self.x>480-50:
        self.direction = "left"
    elif self.x<0:
        self.direction = "right"

def sheBullet(self):
    num = random.randint(1,100)
    if num == 88:
        newBullet = EnemyBullet(self.x,self.y,self.screen)
        self.bulletList.append(newBullet)

class EnemyBullet(object):
    def __init__(self,x,y,screen):
        self.x = x+30
        self.y = y+30
        self.screen = screen
        self.image = pygame.image.load("./feiji/bullet-1.gif").convert()

    def move(self):
        self.y += 2

    def display(self):
        self.screen.blit(self.image,(self.x,self.y))
```

```
def judge(self):
    if self.y>890:
        return True
    else:
        return False

if __name__ == "__main__":
    #1. 创建一个窗口，用来显示内容
    screen = pygame.display.set_mode((480,890),0,32)

    #2. 创建一个和窗口大小的图片，用来充当背景
    background = pygame.image.load("./feiji/background.png").convert()

    #3. 创建一个飞机对象
    heroPlane = HeroPlane(screen)

    #4. 创建一个敌人飞机
    enemyPlane = EnemyPlane(screen)

    #3. 把背景图片放到窗口中显示
    while True:
        screen.blit(background,(0,0))

        heroPlane.display()

        enemyPlane.move()
        enemyPlane.shootBullet()
        enemyPlane.display()

        #判断是否是点击了退出按钮
        for event in pygame.event.get():
            # print(event.type)
            if event.type == QUIT:
                print("exit")
                exit()
```

```
    elif event.type == KEYDOWN:  
        if event.key == K_a or event.key == K_LEFT:  
            print('left')  
            heroPlane.moveLeft()  
            #控制飞机让其向左移动  
        elif event.key == K_d or event.key == K_RIGHT:  
            print('right')  
            heroPlane.moveRight()  
        elif event.key == K_SPACE:  
            print("space")  
            heroPlane.sh射Bullet()  
  
#通过延时的方式，来降低这个while循环的循环速度，从而降低了cpu占用  
time.sleep(0.01)  
  
pygame.display.update()
```

# 打飞机代码：代码优化-抽象出基类

```
#coding=utf-8
import time
import random
import pygame
from pygame.locals import *

class Base(object):
    def __init__(self, screen, name):
        self.name = name
        #设置要显示内容的窗口
        self.screen = screen

class Plane(Base):
    def __init__(self, screen, name):
        super().__init__(screen, name)
        self.image = pygame.image.load(self.imageName).convert()
        #用来存储英雄飞机发射的所有子弹
        self.bulletList = []

    def display(self):
        #更新飞机的位置
        self.screen.blit(self.image, (self.x, self.y))
        #判断一下子弹的位置是否越界，如果是，那么就要删除这颗子弹
        #
        #这种方法会漏掉很多需要删除的数据
        # for i in self.bulletList:
        #     if i.y<0:
        #         self.bulletList.remove(i)
        #存放需要删除的对象信息
        needDelItemList = []
        for i in self.bulletList:
            if i.judge():
```

```
        needDelItemList.append(i)
    for i in needDelItemList:
        self.bulletList.remove(i)
    # del needDelItemList
    #更新及这架飞机发射出的所有子弹的位置
    for bullet in self.bulletList:
        bullet.display()
        bullet.move()

    #修改所有子弹的位置
    # for bullet in self.bulletList:
    #     bullet.y -= 2
def sheBullet(self):
    newBullet = PublicBullet(self.x, self.y, self.screen, self.
    self.bulletList.append(newBullet)

class HeroPlane(Plane):
    def __init__(self, screen, name):
        #设置飞机默认的位置
        self.x = 230
        self.y = 600
        self.imageName = "./feiji/hero.gif"
        super().__init__(screen, name)

    def moveLeft(self):
        self.x -= 10
    def moveRight(self):
        self.x += 10

class EnemyPlane(Plane):
    #重写父类的__init_-方法
    def __init__(self, screen, name):
        #设置飞机默认的位置
        self.x = 0
        self.y = 0
```

```
self.imageName = "./feiji/enemy-1.gif"

#调用父类的__init__方法
super().__init__(screen, name)

self.direction = "right"

def move(self):
    #如果碰到了右边的边界，那么就往左走，如果碰到了左边的边界，那么就往右走
    if self.direction == "right":
        self.x += 2
    elif self.direction == "left":
        self.x -= 2

    if self.x>480-50:
        self.direction = "left"
    elif self.x<0:
        self.direction = "right"

def sheBullet(self):
    num = random.randint(1,100)
    if num == 88:
        super().sheBullet()

class PublicBullet(Base):
    def __init__(self,x,y,screen,planeName):

        super().__init__(screen,planeName)

        if self.name == "hero":
            self.x = x+40
            self.y = y-20
            imageName = "./feiji/bullet-3.gif"

        elif self.name == "enemy":
            self.x = x+30
            self.y = y+30
            imageName = "./feiji/bullet-1.gif"
```

```
        self.image = pygame.image.load(imageName).convert()
def move(self):
    if self.name == "hero":
        self.y -= 2
    elif self.name == "enemy":
        self.y += 2

def display(self):
    self.screen.blit(self.image,(self.x,self.y))

def judge(self):
    if self.y>890 or self.y<0:
        return True
    else:
        return False

if __name__ == "__main__":
    #1. 创建一个窗口，用来显示内容
    screen = pygame.display.set_mode((480,890),0,32)

    #2. 创建一个和窗口大小的图片，用来充当背景
    background = pygame.image.load("./feiji/background.png").cor

    #3. 创建一个飞机对象
    heroPlane = HeroPlane(screen,"hero")

    #4. 创建一个敌人飞机
    enemyPlane = EnemyPlane(screen,"enemy")

    #3. 把背景图片放到窗口中显示
    while True:
        screen.blit(background,(0,0))

        heroPlane.display()

        enemyPlane.move()
```

```
enemyPlane.shoot()
enemyPlane.display()

#判断是否是点击了退出按钮
for event in pygame.event.get():
    # print(event.type)
    if event.type == QUIT:
        print("exit")
        exit()
    elif event.type == KEYDOWN:
        if event.key == K_a or event.key == K_LEFT:
            print('left')
            heroPlane.moveLeft()
            #控制飞机让其向左移动
        elif event.key == K_d or event.key == K_RIGHT:
            print('right')
            heroPlane.moveRight()
        elif event.key == K_SPACE:
            print("space")
            heroPlane.shoot()

time.sleep(0.01)

pygame.display.update()
```