[**Python之路,Day1 - Python基础1**](http://www.cnblogs.com/alex3714/articles/5465198.html)

<http://www.cnblogs.com/alex3714/articles/5465198.html>

**本节内容**

1. Python介绍
2. 发展史
3. Python 2 or 3?
4. 安装
5. Hello World程序
6. 变量
7. 用户输入
8. 模块初识
9. .pyc是个什么鬼？
10. 数据类型初识
11. 数据运算
12. 表达式if ...else语句
13. 表达式for 循环
14. break and continue
15. 表达式while 循环
16. 作业需求

**一、 Python介绍**

python的创始人为吉多·范罗苏姆（Guido van Rossum）。1989年的圣诞节期间，吉多·范罗苏姆为了在阿姆斯特丹打发时间，决心开发一个新的脚本解释程序，作为ABC语言的一种继承。

最新的TIOBE排行榜，Python赶超PHP占据第五， Python崇尚优美、清晰、简单，是一个优秀并广泛使用的语言。



由上图可见，Python整体呈上升趋势，反映出Python应用越来越广泛并且也逐渐得到业内的认可！！！

Python可以应用于众多领域，如：数据分析、组件集成、网络服务、图像处理、数值计算和科学计算等众多领域。目前业内几乎所有大中型互联网企业都在使用Python，如：Youtube、Dropbox、BT、Quora（中国知乎）、豆瓣、知乎、Google、Yahoo!、Facebook、NASA、百度、腾讯、汽车之家、美团等。

**目前Python主要应用领域：**

* **云计算**: 云计算最火的语言， 典型应用OpenStack
* **WEB开发**: 众多优秀的WEB框架，众多大型网站均为Python开发，Youtube, Dropbox, 豆瓣。。。， 典型WEB框架有Django
* **科学运算、人工智能**: 典型库NumPy, SciPy, Matplotlib, Enthought librarys,pandas
* **系统运维**: 运维人员必备语言
* **金融**：量化交易，金融分析，在金融工程领域，Python不但在用，且用的最多，而且重要性逐年提高。原因：作为动态语言的Python，语言结构清晰简单，库丰富，成熟稳定，科学计算和统计分析都很牛，生产效率远远高于c,c++,java,尤其擅长策略回测
* **图形GUI**: PyQT, WxPython,TkInter

**Python在一些公司的应用：**

* 谷歌：Google App Engine 、code.google.com 、Google earth 、谷歌爬虫、Google广告等项目都在大量使用Python开发
* CIA: 美国中情局网站就是用Python开发的
* NASA: 美国航天局(NASA)大量使用Python进行数据分析和运算
* YouTube:世界上最大的视频网站YouTube就是用Python开发的
* Dropbox:美国最大的在线云存储网站，全部用Python实现，每天网站处理10亿个文件的上传和下载
* Instagram:美国最大的图片分享社交网站，每天超过3千万张照片被分享，全部用python开发
* Facebook:大量的基础库均通过Python实现的
* Redhat: 世界上最流行的Linux发行版本中的yum包管理工具就是用python开发的
* 豆瓣: 公司几乎所有的业务均是通过Python开发的
* 知乎: 国内最大的问答社区，通过Python开发(国外Quora)
* 春雨医生：国内知名的在线医疗网站是用Python开发的
* 除上面之外，还有搜狐、金山、腾讯、盛大、网易、百度、阿里、淘宝 、土豆、新浪、果壳等公司都在使用Python完成各种各样的任务。

**Python 是一门什么样的语言？**

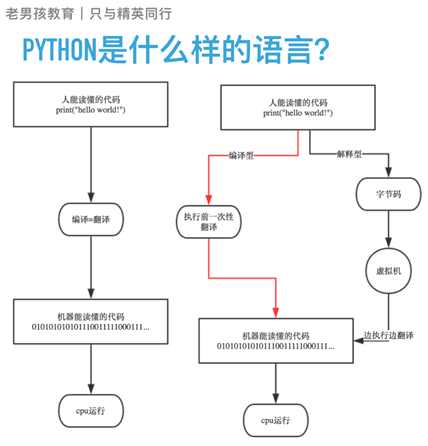
编程语言主要从以下几个角度为进行分类，编译型和解释型、静态语言和动态语言、强类型定义语言和弱类型定义语言，每个分类代表什么意思呢，我们一起来看一下。

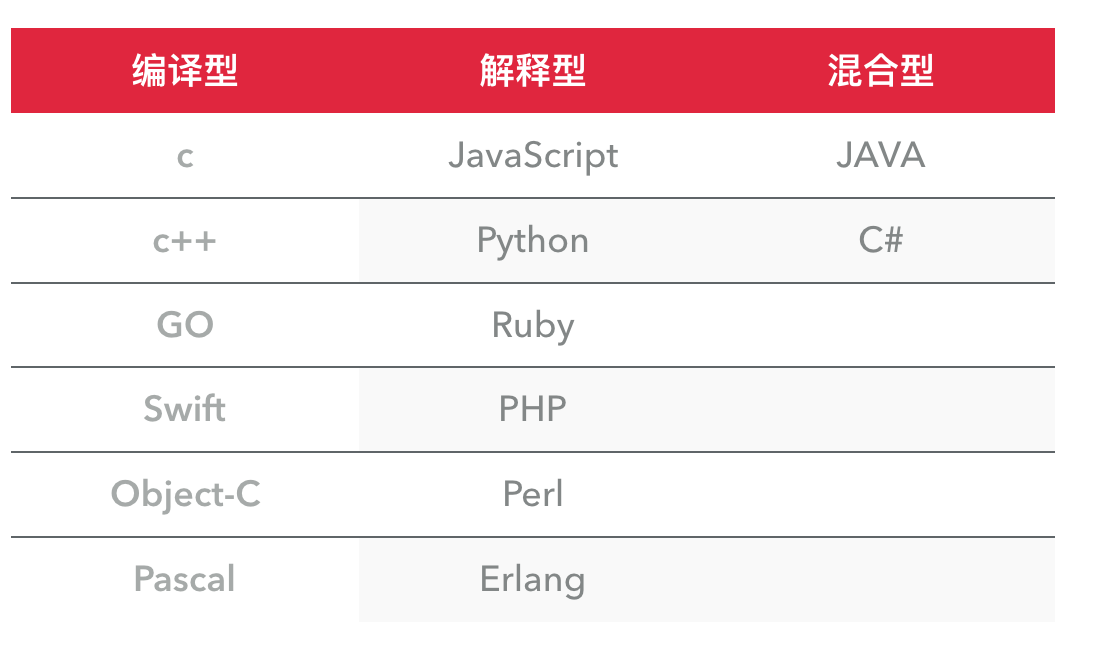
**编译和解释的区别是什么？**

**编译器**是把源程序的每一条语句都编译成机器语言,并保存成二进制文件,这样运行时计算机可以直接以机器语言来运行此程序,速度很快;

而**解释器**则是只在执行程序时,才一条一条的解释成机器语言给计算机来执行,所以运行速度是不如编译后的程序运行的快的.

这是因为计算机不能直接认识并执行我们写的语句,它只能认识机器语言(是二进制的形式)





**编译型vs解释型**

**编译型**  
优点：编译器一般会有预编译的过程对代码进行优化。因为编译只做一次，运行时不需要编译，所以编译型语言的程序执行效率高。可以脱离语言环境独立运行。  
缺点：编译之后如果需要修改就需要整个模块重新编译。编译的时候根据对应的运行环境生成机器码，不同的操作系统之间移植就会有问题，需要根据运行的操作系统环境编译不同的可执行文件。

**解释型**  
优点：有良好的平台兼容性，在任何环境中都可以运行，前提是安装了解释器（虚拟机）。灵活，修改代码的时候直接修改就可以，可以快速部署，不用停机维护。

缺点：每次运行的时候都要解释一遍，性能上不如编译型语言。

***一、低级语言与高级语言***  
  
*最初的计算机程序都是用0和1的序列表示的，程序员直接使用的是机器指令，无需翻译，从纸带打孔输入即可执行得到结果。后来为了方便记忆，就将用0、1序列表示的机器指令都用符号助记，这些与机器指令一一对应的助记符就成了汇编指令，从而诞生了汇编语言。无论是机器指令还是汇编指令都是面向机器的，统称为低级语言。因为是针对特定机器的机器指令的助记符，所以汇编语言是无法独立于机器(特定的CPU体系结构)的。但汇编语言也是要经过翻译成机器指令才能执行的，所以也有将运行在一种机器上的汇编语言翻译成运行在另一种机器上的机器指令的方法，那就是交叉汇编技术。*  
  
*高级语言是从人类的逻辑思维角度出发的计算机语言，抽象程度大大提高，需要经过编译成特定机器上的目标代码才能执行，一条高级语言的语句往往需要若干条机器指令来完成。高级语言独立于机器的特性是靠编译器为不同机器生成不同的目标代码(或机器指令)来实现的。那具体的说，要将高级语言编译到什么程度呢，这又跟编译的技术有关了，既可以编译成直接可执行的目标代码，也可以编译成一种中间表示，然后拿到不同的机器和系统上去执行，这种情况通常又需要支撑环境，比如解释器或虚拟机的支持，Java程序编译成bytecode，再由不同平台上的虚拟机执行就是很好的例子。所以，说高级语言不依赖于机器，是指在不同的机器或平台上高级语言的程序本身不变，而通过编译器编译得到的目标代码去适应不同的机器。从这个意义上来说，通过交叉汇编，一些汇编程序也可以获得不同机器之间的可移植性，但这种途径获得的移植性远远不如高级语言来的方便和实用性大。*

***二、编译与解释***  
  
*编译是将源程序翻译成可执行的目标代码，翻译与执行是分开的；而解释是对源程序的翻译与执行一次性完成，不生成可存储的目标代码。这只是表象，二者背后的最大区别是：对解释执行而言，程序运行时的控制权在解释器而不在用户程序；对编译执行而言，运行时的控制权在用户程序。*  
  
*解释具有良好的动态特性和可移植性，比如在解释执行时可以动态改变变量的类型、对程序进行修改以及在程序中插入良好的调试诊断信息等，而将解释器移植到不同的系统上，则程序不用改动就可以在移植了解释器的系统上运行。同时解释器也有很大的缺点，比如执行效率低，占用空间大，因为不仅要给用户程序分配空间，解释器本身也占用了宝贵的系统资源。*

*编译器是把源程序的每一条语句都编译成机器语言,并保存成二进制文件,这样运行时计算机可以直接以机器语言来运行此程序,速度很快;*  
*而解释器则是只在执行程序时,才一条一条的解释成机器语言给计算机来执行,所以运行速度是不如编译后的程序运行的快的.*

***编译型和解释型***  
*我们先看看编译型，其实它和汇编语言是一样的：也是有一个负责翻译的程序来对我们的源代码进行转换，生成相对应的可执行代码。这个过程说得专业一点，就称为编译（Compile），而负责编译的程序自然就称为编译器（Compiler）。如果我们写的程序代码都包含在一个源文件中，那么通常编译之后就会直接生成一个可执行文件，我们就可以直接运行了。但对于一个比较复杂的项目，为了方便管理，我们通常把代码分散在各个源文件中，作为不同的模块来组织。这时编译各个文件时就会生成目标文件（Object   file）而不是前面说的可执行文件。一般一个源文件的编译都会对应一个目标文件。这些目标文件里的内容基本上已经是可执行代码了，但由于只是整个项目的一部分，所以我们还不能直接运行。待所有的源文件的编译都大功告成，我们就可以最后把这些半成品的目标文件“打包”成一个可执行文件了，这个工作由另一个程序负责完成，由于此过程好像是把包含可执行代码的目标文件连接装配起来，所以又称为链接（Link），而负责链接的程序就叫……就叫链接程序（Linker）。链接程序除了链接目标文件外，可能还有各种资源，像图标文件啊、声音文件啊什么的，还要负责去除目标文件之间的冗余重复代码，等等，所以……也是挺累的。链接完成之后，一般就可以得到我们想要的可执行文件了。*  
  
*上面我们大概地介绍了编译型语言的特点，现在再看看解释型。噢，从字面上看，“编译”和“解释”的确都有“翻译”的意思，它们的区别则在于翻译的时机安排不大一样。打个比方：假如你打算阅读一本外文书，而你不知道这门外语，那么你可以找一名翻译，给他足够的时间让他从头到尾把整本书翻译好，然后把书的母语版交给你阅读；或者，你也立刻让这名翻译辅助你阅读，让他一句一句给你翻译，如果你想往回看某个章节，他也得重新给你翻译。*  
  
*两种方式，前者就相当于我们刚才所说的编译型：一次把所有的代码转换成机器语言，然后写成可执行文件；而后者就相当于我们要说的解释型：在程序运行的前一刻，还只有源程序而没有可执行程序；而程序每执行到源程序的某一条指令，则会有一个称之为解释程序的外壳程序将源代码转换成二进制代码以供执行，总言之，就是不断地解释、执行、解释、执行……所以，解释型程序是离不开解释程序的。像早期的BASIC就是一门经典的解释型语言，要执行BASIC程序，就得进入BASIC环境，然后才能加载程序源文件、运行。解释型程序中，由于程序总是以源代码的形式出现，因此只要有相应的解释器，移植几乎不成问题。编译型程序虽然源代码也可以移植，但前提是必须针对不同的系统分别进行编译，对于复杂的工程来说，的确是一件不小的时间消耗，况且很可能一些细节的地方还是要修改源代码。而且，解释型程序省却了编译的步骤，修改调试也非常方便，编辑完毕之后即可立即运行，不必像编译型程序一样每次进行小小改动都要耐心等待漫长的Compiling…Linking…这样的编译链接过程。不过凡事有利有弊，由于解释型程序是将编译的过程放到执行过程中，这就决定了解释型程序注定要比编译型慢上一大截，像几百倍的速度差距也是不足为奇的。*  
  
*编译型与解释型，两者各有利弊。前者由于程序执行速度快，同等条件下对系统要求较低，因此像开发操作系统、大型应用程序、数据库系统等时都采用它，像C/C++、Pascal/Object   Pascal（Delphi）、VB等基本都可视为编译语言，而一些网页脚本、服务器脚本及辅助开发接口这样的对速度要求不高、对不同系统平台间的兼容性有一定要求的程序则通常使用解释性语言，如Java、JavaScript、VBScript、Perl、Python等等。*  
  
*但既然编译型与解释型各有优缺点又相互对立，所以一批新兴的语言都有把两者折衷起来的趋势，例如Java语言虽然比较接近解释型语言的特征，但在执行之前已经预先进行一次预编译，生成的代码是介于机器码和Java源代码之间的中介代码，运行的时候则由JVM（Java的虚拟机平台，可视为解释器）解释执行。它既保留了源代码的高抽象、可移植的特点，又已经完成了对源代码的大部分预编译工作，所以执行起来比“纯解释型”程序要快许多。而像VB6（或者以前版本）、C#这样的语言，虽然表面上看生成的是.exe可执行程序文件，但VB6编译之后实际生成的也是一种中介码，只不过编译器在前面安插了一段自动调用某个外部解释器的代码（该解释程序独立于用户编写的程序，存放于系统的某个DLL文件中，所有以VB6编译生成的可执行程序都要用到它），以解释执行实际的程序体。C#（以及其它.net的语言编译器）则是生成.net目标代码，实际执行时则由.net解释系统（就像JVM一样，也是一个虚拟机平台）进行执行。当然.net目标代码已经相当“低级”，比较接近机器语言了，所以仍将其视为编译语言，而且其可移植程度也没有Java号称的这么强大，Java号称是“一次编译，到处执行”，而.net则是“一次编码，到处编译”。呵呵，当然这些都是题外话了。总之，随着设计技术与硬件的不断发展，编译型与解释型两种方式的界限正在不断变得模糊。*

***动态语言和静态语言***  
*通常我们所说的动态语言、静态语言是指动态类型语言和静态类型语言。*

*（1）动态类型语言：动态类型语言是指在运行期间才去做数据类型检查的语言，也就是说，在用动态类型的语言编程时，永远也不用给任何变量指定数据类型，该语言会在你第一次赋值给变量时，在内部将数据类型记录下来。Python和Ruby就是一种典型的动态类型语言，其他的各种脚本语言如VBScript也多少属于动态类型语言。*

*（2）静态类型语言：静态类型语言与动态类型语言刚好相反，它的数据类型是在编译其间检查的，也就是说在写程序时要声明所有变量的数据类型，C/C++是静态类型语言的典型代表，其他的静态类型语言还有C#、JAVA等。*

***强类型定义语言和弱类型定义语言***

*（1）强类型定义语言：强制数据类型定义的语言。也就是说，一旦一个变量被指定了某个数据类型，如果不经过强制转换，那么它就永远是这个数据类型了。举个例子：如果你定义了一个整型变量a,那么程序根本不可能将a当作字符串类型处理。强类型定义语言是类型安全的语言。*

*（2）弱类型定义语言：数据类型可以被忽略的语言。它与强类型定义语言相反, 一个变量可以赋不同数据类型的值。*

*强类型定义语言在速度上可能略逊色于弱类型定义语言，但是强类型定义语言带来的严谨性能够有效的避免许多错误。另外，“这门语言是不是动态语言”与“这门语言是否类型安全”之间是完全没有联系的！*  
*例如：Python是动态语言，是强类型定义语言（类型安全的语言）; VBScript是动态语言，是弱类型定义语言（类型不安全的语言）; JAVA是静态语言，是强类型定义语言（类型安全的语言）。*

*通过上面这些介绍，我们可以得出，****python是一门动态解释性的强类型定义语言。****那这些基因使成就了Python的哪些优缺点呢？我们继续往下看。*

**Python的优缺点**

先看优点

1. Python的定位是“优雅”、“明确”、“简单”，所以Python程序看上去总是简单易懂，初学者学Python，不但入门容易，而且将来深入下去，可以编写那些非常非常复杂的程序。
2. 开发效率非常高，Python有非常强大的第三方库，基本上你想通过计算机实现任何功能，Python官方库里都有相应的模块进行支持，直接下载调用后，在基础库的基础上再进行开发，大大降低开发周期，避免重复造轮子。
3. 高级语言，当你用Python语言编写程序的时候，你无需考虑诸如何管理你的程序使用的内存一类的底层细节
4. 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程序无需修改就几乎可以在市场上所有的系统平台上运行
5. 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。
6. 可嵌入性，你可以把Python嵌入你的C/C++程序，从而向你的程序用户提供脚本功能。

再看缺点：

1. 速度慢，Python 的运行速度相比C语言确实慢很多，跟JAVA相比也要慢一些，因此这也是很多所谓的大牛不屑于使用Python的主要原因，但其实这里所指的运行速度慢在大多数情况下用户是无法直接感知到的，必须借助测试工具才能体现出来，比如你用C运行一个程序花了0.01s,用Python是0.1s,这样C语言直接比Python快了10倍,算是非常夸张了，但是你是无法直接通过肉眼感知的，因为一个正常人所能感知的时间最小单位是0.15-0.4s左右。其实在大多数情况下Python已经完全可以满足你对程序速度的要求，除非你要写对速度要求极高的搜索引擎等，这种情况下，当然还是建议你用C去实现的。
2. 代码不能加密，因为PYTHON是解释性语言，它的源码都是以名文形式存放的，不过我不认为这算是一个缺点，如果你的项目要求源代码必须是加密的，那你一开始就不应该用Python来去实现。
3. 线程不能利用多CPU问题，这是Python被人诟病最多的一个缺点，GIL即全局解释器锁（Global Interpreter Lock），是[计算机程序设计语言](http://zh.wikipedia.org/wiki/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A8%8B%E5%BA%8F%E8%AE%BE%E8%AE%A1%E8%AF%AD%E8%A8%80)[解释器](http://zh.wikipedia.org/wiki/%E8%A7%A3%E9%87%8A%E5%99%A8)用于[同步](http://zh.wikipedia.org/wiki/%E5%90%8C%E6%AD%A5)[线程](http://zh.wikipedia.org/wiki/%E7%BA%BF%E7%A8%8B)的工具，使得任何时刻仅有一个线程在执行，Python的线程是操作系统的原生线程。在Linux上为pthread，在Windows上为Win thread，完全由操作系统调度线程的执行。一个python解释器进程内有一条主线程，以及多条用户程序的执行线程。即使在多核CPU平台上，由于GIL的存在，所以禁止多线程的并行执行。关于这个问题的折衷解决方法，我们在以后线程和进程章节里再进行详细探讨。

当然，Python还有一些其它的小缺点，在这就不一一列举了，我想说的是，任何一门语言都不是完美的，都有擅长和不擅长做的事情，建议各位不要拿一个语言的劣势去跟另一个语言的优势来去比较，语言只是一个工具，是实现程序设计师思想的工具，就像我们之前中学学几何时，有的时候需要要圆规，有的时候需要用三角尺一样，拿相应的工具去做它最擅长的事才是正确的选择。之前很多人问我Shell和Python到底哪个好？我回答说Shell是个脚本语言，但Python不只是个脚本语言，能做的事情更多。

**Python解释器**

当我们编写Python代码时，我们得到的是一个包含Python代码的以.py为扩展名的文本文件。要运行代码，就需要Python解释器去执行.py文件。

由于整个Python语言从规范到解释器都是开源的，所以理论上，只要水平够高，任何人都可以编写Python解释器来执行Python代码（当然难度很大）。事实上，确实存在多种Python解释器。

**CPython**

当我们从[Python官方网站](https://www.python.org/)下载并安装好Python 2.7后，我们就直接获得了一个官方版本的解释器：CPython。这个解释器是用C语言开发的，所以叫CPython。在命令行下运行python就是启动CPython解释器。

CPython是使用最广的Python解释器。教程的所有代码也都在CPython下执行。

**IPython**

IPython是基于CPython之上的一个交互式解释器，也就是说，IPython只是在交互方式上有所增强，但是执行Python代码的功能和CPython是完全一样的。好比很多国产浏览器虽然外观不同，但内核其实都是调用了IE。

CPython用>>>作为提示符，而IPython用In [序号]:作为提示符。

**PyPy**

PyPy是另一个Python解释器，它的目标是执行速度。PyPy采用[JIT技术](http://en.wikipedia.org/wiki/Just-in-time_compilation)，对Python代码进行动态编译（注意不是解释），所以可以显著提高Python代码的执行速度。

绝大部分Python代码都可以在PyPy下运行，但是PyPy和CPython有一些是不同的，这就导致相同的Python代码在两种解释器下执行可能会有不同的结果。如果你的代码要放到PyPy下执行，就需要了解[PyPy和CPython的不同点](http://pypy.readthedocs.org/en/latest/cpython_differences.html)。

**Jython**

Jython是运行在Java平台上的Python解释器，可以直接把Python代码编译成Java字节码执行。

**IronPython**

IronPython和Jython类似，只不过IronPython是运行在微软.Net平台上的Python解释器，可以直接把Python代码编译成.Net的字节码。

**小结**

Python的解释器很多，但使用最广泛的还是CPython。如果要和Java或.Net平台交互，最好的办法不是用Jython或IronPython，而是通过网络调用来交互，确保各程序之间的独立性。

**二、Python发展史**

* 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。
* 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。
* Granddaddy of Python web frameworks, Zope 1 was released in 1999
* Python 1.0 - January 1994 增加了 [lambda](https://en.wikipedia.org/wiki/Lambda_calculus), [map](https://en.wikipedia.org/wiki/Map_(higher-order_function)), [filter](https://en.wikipedia.org/wiki/Filter_(higher-order_function)) and [reduce](https://en.wikipedia.org/wiki/Fold_(higher-order_function)).
* Python 2.0 - October 16, 2000，加入了内存回收机制，构成了现在Python语言框架的基础
* Python 2.4 - November 30, 2004, 同年目前最流行的WEB框架Django 诞生
* Python 2.5 - September 19, 2006
* Python 2.6 - October 1, 2008
* Python 2.7 - July 3, 2010
* In November 2014, it was announced that Python 2.7 would be supported until 2020, and reaffirmed that there would be no 2.8 release as users were expected to move to Python 3.4+ as soon as possible
* Python 3.0 - December 3, 2008
* Python 3.1 - June 27, 2009
* Python 3.2 - February 20, 2011
* Python 3.3 - September 29, 2012
* Python 3.4 - March 16, 2014
* Python 3.5 - September 13, 2015

**三、Python 2 or 3?**

**In summary : Python 2.x is legacy, Python 3.x is the present and future of the language**

Python 3.0 was released in 2008. The final 2.x version 2.7 release came out in mid-2010, with a statement of

extended support for this end-of-life release. The 2.x branch will see no new major releases after that. 3.x is

under active development and has already seen over five years of stable releases, including version 3.3 in 2012,

3.4 in 2014, and 3.5 in 2015. This means that all recent standard library improvements, for example, are only

available by default in Python 3.x.

Guido van Rossum (the original creator of the Python language) decided to clean up Python 2.x properly, with less regard for backwards compatibility than is the case for new releases in the 2.x range. The most drastic improvement is the better Unicode support (with all text strings being Unicode by default) as well as saner bytes/Unicode separation.

Besides, several aspects of the core language (such as print and exec being statements, integers using floor division) have been adjusted to be easier for newcomers to learn and to be more consistent with the rest of the language, and old cruft has been removed (for example, all classes are now new-style, "range()" returns a memory efficient iterable, not a list as in 2.x).

**py2与3的详细区别**

**PRINT IS A FUNCTION**

The statement has been replaced with a print() function, with keyword arguments to replace most of the special syntax of the old statement (PEP 3105). Examples:

Old: print "The answer is", 2\*2 New: print("The answer is", 2\*2)

Old: print x, # Trailing comma suppresses newline New: print(x, end=" ") # Appends a space instead of a newline

Old: print # Prints a newline

New: print() # You must call the function!

Old: print >>sys.stderr, "fatal error" New: print("fatal error", file=sys.stderr)

Old: print (x, y) # prints repr((x, y))

New: print((x, y)) # Not the same as print(x, y)!

You can also customize the separator between items, e.g.:

|  |  |
| --- | --- |
| 1 | print("There are <", 2\*\*32, "> possibilities!", sep="") |

**ALL IS UNICODE NOW**

**从此不再为讨厌的字符编码而烦恼**

**还可以这样玩： (A,\*REST,B)=RANGE(5)**

|  |  |
| --- | --- |
| 1  2  3  4 | <strong>>>> a,\*rest,b = range(5)  >>> a,rest,b  (0, [1, 2, 3], 4)  </strong> |

**某些库改名了**

|  |  |
| --- | --- |
| Old Name | New Name |
| \_winreg | winreg |
| ConfigParser | configparser |
| copy\_reg | copyreg |
| Queue | queue |
| SocketServer | socketserver |
| markupbase | \_markupbase |
| repr | reprlib |
| test.test\_support | test.support |

**还有谁不支持PYTHON3?**

One popular module that don't yet support Python 3 is Twisted (for networking and other applications). Most

actively maintained libraries have people working on 3.x support. For some libraries, it's more of a priority than

others: Twisted, for example, is mostly focused on production servers, where supporting older versions of

Python is important, let alone supporting a new version that includes major changes to the language. (Twisted is

a prime example of a major package where porting to 3.x is far from trivial

**四、Python安装**

**windows**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 1、下载安装包      https://www.python.org/downloads/  2、安装      默认安装路径：C:\python27  3、配置环境变量      【右键计算机】--》【属性】--》【高级系统设置】--》【高级】--》【环境变量】--》【在第二个内容框中找到 变量名为Path 的一行，双击】 --> 【Python安装目录追加到变值值中，用 ； 分割】      如：原来的值;C:\python27，切记前面有分号 |

**linux、Mac**

|  |  |
| --- | --- |
| 1  2  3 | 无需安装，原装Python环境    ps：如果自带2.6，请更新至2.7 |

**五、Hello World程序**

在linux 下创建一个文件叫hello.py,并输入

|  |  |
| --- | --- |
| 1 | print("Hello World!") |

然后执行命令:python hello.py ,输出

|  |  |
| --- | --- |
| 1  2  3 | localhost:~ jieli$ vim hello.py  localhost:~ jieli$ python hello.py  Hello World! |

**指定解释器**

上一步中执行 python hello.py 时，明确的指出 hello.py 脚本由 python 解释器来执行。

如果想要类似于执行shell脚本一样执行python脚本，例： ./hello.py ，那么就需要在 hello.py 文件的头部指定解释器，如下：

|  |  |
| --- | --- |
| 1  2  3 | #!/usr/bin/env python    print "hello,world" |

如此一来，执行： ./hello.py 即可。

ps：执行前需给予 hello.py 执行权限，chmod 755 hello.py

**在交互器中执行**

除了把程序写在文件里，还可以直接调用python自带的交互器运行代码，

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | localhost:~ jieli$ python  Python 2.7.10 (default, Oct 23 2015, 18:05:06)  [GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin  Type "help", "copyright", "credits" or "license" for more information.  >>> print("Hello World!")  Hello World! |

对比下其它语言的hello world

 C++

1 #include <iostream>

2 int main(void)

3 {

4 std::cout<<"Hello world";

5 }

 C

1 #include <stdio.h>

2 int main(void)

3 {

4 printf("\nhello world!");

5 return 0;

6 }

 JAVA

1 public class HelloWorld{

2 // 程序的入口

3 public static void main(String args[]){

4 // 向控制台输出信息

5 System.out.println("Hello World!");

6 }

7 }

 PHP

1 <?php

2 echo "hello world!";

3 ?>

 RUBY

puts "Hello world."

 Go

1 package main

2

3 import "fmt"

4

5 func main(){

6

7 fmt.Printf("Hello World!\n God Bless You!");

8

9 }

**六、变量\字符编码**

**Variables** are used to store information to be referenced and manipulated in a computer program. They also provide a way of labeling data with a descriptive name, so our programs can be understood more clearly by the reader and ourselves. It is helpful to think of variables as containers that hold information. Their sole purpose is to label and store data in memory. This data can then be used throughout your program.

**声明变量**

|  |  |
| --- | --- |
| 1  2  3 | #\_\*\_coding:utf-8\_\*\_    name = "Alex Li" |

上述代码声明了一个变量，变量名为： name，变量name的值为："Alex Li"

**变量定义的规则：**

* + 变量名只能是 字母、数字或下划线的任意组合
  + 变量名的第一个字符不能是数字
  + 以下关键字不能声明为变量名  
    ['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try', 'while', 'with', 'yield']

**变量的赋值**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | name = "Alex Li"    name2 = name  print(name,name2)    name = "Jack"    print("What is the value of name2 now?") |

**七、字符编码**

python解释器在加载 .py 文件中的代码时，会对内容进行编码（默认ascill）

ASCII（American Standard Code for Information Interchange，美国标准信息交换代码）是基于拉丁字母的一套电脑编码系统，主要用于显示现代英语和其他西欧语言，其最多只能用 8 位来表示（一个字节），即：2\*\*8 = 256-1，所以，ASCII码最多只能表示 255 个符号。



**关于中文**

为了处理汉字，程序员设计了用于简体中文的GB2312和用于繁体中文的big5。

GB2312(1980年)一共收录了7445个字符，包括6763个汉字和682个其它符号。汉字区的内码范围高字节从B0-F7，低字节从A1-FE，占用的码位是72\*94=6768。其中有5个空位是D7FA-D7FE。

GB2312 支持的汉字太少。1995年的汉字扩展规范GBK1.0收录了21886个符号，它分为汉字区和图形符号区。汉字区包括21003个字符。2000年的 GB18030是取代GBK1.0的正式国家标准。该标准收录了27484个汉字，同时还收录了藏文、蒙文、维吾尔文等主要的少数民族文字。现在的PC平台必须支持GB18030，对嵌入式产品暂不作要求。所以手机、MP3一般只支持GB2312。

从ASCII、GB2312、GBK 到GB18030，这些编码方法是向下兼容的，即同一个字符在这些方案中总是有相同的编码，后面的标准支持更多的字符。在这些编码中，英文和中文可以统一地处理。区分中文编码的方法是高字节的最高位不为0。按照程序员的称呼，GB2312、GBK到GB18030都属于双字节字符集 (DBCS)。

有的中文Windows的缺省内码还是GBK，可以通过GB18030升级包升级到GB18030。不过GB18030相对GBK增加的字符，普通人是很难用到的，通常我们还是用GBK指代中文Windows内码。

**显然ASCII码无法将世界上的各种文字和符号全部表示，所以，就需要新出一种可以代表所有字符和符号的编码，即：Unicode**

Unicode（统一码、万国码、单一码）是一种在计算机上使用的字符编码。Unicode 是为了解决传统的字符编码方案的局限而产生的，它为每种语言中的每个字符设定了统一并且唯一的二进制编码，规定虽有的字符和符号最少由 16 位来表示（2个字节），即：2 \*\*16 = 65536，  
注：此处说的的是最少2个字节，可能更多

UTF-8，是对Unicode编码的压缩和优化，他不再使用最少使用2个字节，而是将所有的字符和符号进行分类：ascii码中的内容用1个字节保存、欧洲的字符用2个字节保存，东亚的字符用3个字节保存...

所以，python解释器在加载 .py 文件中的代码时，会对内容进行编码（默认ascill），如果是如下代码的话：

报错：ascii码无法表示中文

|  |  |
| --- | --- |
| 1  2  3 | #!/usr/bin/env python    print "你好，世界" |

改正：应该显示的告诉python解释器，用什么编码来执行源代码，即：

|  |  |
| --- | --- |
| 1  2  3  4 | #!/usr/bin/env python  # -\*- coding: utf-8 -\*-    print "你好，世界" |

**注释**

　　当行注视：# 被注释内容

　　多行注释：""" 被注释内容 """

**八、用户输入**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #!/usr/bin/env python  #\_\*\_coding:utf-8\_\*\_      #name = raw\_input("What is your name?") #only on python 2.x  name = input("What is your name?")  print("Hello " + name ) |

输入密码时，如果想要不可见，需要利用getpass 模块中的 getpass方法，即：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #!/usr/bin/env python  # -\*- coding: utf-8 -\*-    import getpass    # 将用户输入的内容赋值给 name 变量  pwd = getpass.getpass("请输入密码：")    # 打印输入的内容  print(pwd) |

**九、模块初识**

Python的强大之处在于他有非常丰富和强大的标准库和第三方库，几乎你想实现的任何功能都有相应的Python库支持，以后的课程中会深入讲解常用到的各种库，现在，我们先来象征性的学2个简单的。

**sys**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | #!/usr/bin/env python  # -\*- coding: utf-8 -\*-    import sys    print(sys.argv)      #输出  $ python test.py helo world  ['test.py', 'helo', 'world']  #把执行脚本时传递的参数获取到了 |

**os**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #!/usr/bin/env python  # -\*- coding: utf-8 -\*-    import os    os.system("df -h") #调用系统命令 |

**完全结合一下**

|  |  |
| --- | --- |
| 1  2  3 | import os,sys    os.system(''.join(sys.argv[1:])) #把用户的输入的参数当作一条命令交给os.system来执行 |

**自己写个模块**

python tab补全模块

 for mac

1 import sys

2 import readline

3 import rlcompleter

4

5 if sys.platform == 'darwin' and sys.version\_info[0] == 2:

6 readline.parse\_and\_bind("bind ^I rl\_complete")

7 else:

8 readline.parse\_and\_bind("tab: complete") # linux and python3 on mac

 for Linux

1 #!/usr/bin/env python

2 # python startup file

3 import sys

4 import readline

5 import rlcompleter

6 import atexit

7 import os

8 # tab completion

9 readline.parse\_and\_bind('tab: complete')

10 # history file

11 histfile = os.path.join(os.environ['HOME'], '.pythonhistory')

12 try:

13 readline.read\_history\_file(histfile)

14 except IOError:

15 pass

16 atexit.register(readline.write\_history\_file, histfile)

17 del os, histfile, readline, rlcompleter

写完保存后就可以使用了

|  |  |
| --- | --- |
| 1  2  3  4  5 | localhost:~ jieli$ python  Python 2.7.10 (default, Oct 23 2015, 18:05:06)  [GCC 4.2.1 Compatible Apple LLVM 7.0.0 (clang-700.0.59.5)] on darwin  Type "help", "copyright", "credits" or "license" for more information.  >>> import tab |

你会发现，上面自己写的tab.py模块只能在当前目录下导入，如果想在系统的任何一个地方都使用怎么办呢？ 此时你就要把这个tab.py放到python全局环境变量目录里啦，基本一般都放在一个叫 Python/2.7/site-packages 目录下，这个目录在不同的OS里放的位置不一样，用 print(sys.path) 可以查看python环境变量列表。

**十、.pyc是个什么鬼？**

**1. Python是一门解释型语言？**

我初学Python时，听到的关于Python的第一句话就是，Python是一门解释性语言，我就这样一直相信下去，直到发现了\*.pyc文件的存在。如果是解释型语言，那么生成的\*.pyc文件是什么呢？c应该是compiled的缩写才对啊！

为了防止其他学习Python的人也被这句话误解，那么我们就在文中来澄清下这个问题，并且把一些基础概念给理清。

**2. 解释型语言和编译型语言**

计算机是不能够识别高级语言的，所以当我们运行一个高级语言程序的时候，就需要一个“翻译机”来从事把高级语言转变成计算机能读懂的机器语言的过程。这个过程分成两类，第一种是编译，第二种是解释。

编译型语言在程序执行之前，先会通过编译器对程序执行一个编译的过程，把程序转变成机器语言。运行时就不需要翻译，而直接执行就可以了。最典型的例子就是C语言。

解释型语言就没有这个编译的过程，而是在程序运行的时候，通过解释器对程序逐行作出解释，然后直接运行，最典型的例子是Ruby。

通过以上的例子，我们可以来总结一下解释型语言和编译型语言的优缺点，因为编译型语言在程序运行之前就已经对程序做出了“翻译”，所以在运行时就少掉了“翻译”的过程，所以效率比较高。但是我们也不能一概而论，一些解释型语言也可以通过解释器的优化来在对程序做出翻译时对整个程序做出优化，从而在效率上超过编译型语言。

此外，随着Java等基于虚拟机的语言的兴起，我们又不能把语言纯粹地分成解释型和编译型这两种。

用Java来举例，Java首先是通过编译器编译成字节码文件，然后在运行时通过解释器给解释成机器文件。所以我们说Java是一种先编译后解释的语言。

**3. Python到底是什么**

其实Python和Java/C#一样，也是一门基于虚拟机的语言，我们先来从表面上简单地了解一下Python程序的运行过程吧。

当我们在命令行中输入python hello.py时，其实是激活了Python的“解释器”，告诉“解释器”：你要开始工作了。可是在“解释”之前，其实执行的第一项工作和Java一样，是编译。

熟悉Java的同学可以想一下我们在命令行中如何执行一个Java的程序：

javac hello.java

java hello

只是我们在用Eclipse之类的IDE时，将这两部给融合成了一部而已。其实Python也一样，当我们执行python hello.py时，他也一样执行了这么一个过程，所以我们应该这样来描述Python，Python是一门先编译后解释的语言。

**4. 简述Python的运行过程**

在说这个问题之前，我们先来说两个概念，PyCodeObject和pyc文件。

我们在硬盘上看到的pyc自然不必多说，而其实PyCodeObject则是Python编译器真正编译成的结果。我们先简单知道就可以了，继续向下看。

当python程序运行时，编译的结果则是保存在位于内存中的PyCodeObject中，当Python程序运行结束时，Python解释器则将PyCodeObject写回到pyc文件中。

当python程序第二次运行时，首先程序会在硬盘中寻找pyc文件，如果找到，则直接载入，否则就重复上面的过程。

所以我们应该这样来定位PyCodeObject和pyc文件，我们说pyc文件其实是PyCodeObject的一种持久化保存方式。

**十一、数据类型初识**

1、数字

2 是一个整数的例子。  
长整数 不过是大一些的整数。  
3.23和52.3E-4是浮点数的例子。E标记表示10的幂。在这里，52.3E-4表示52.3 \* 10-4。  
(-5+4j)和(2.3-4.6j)是复数的例子，其中-5,4为实数，j为虚数，数学中表示复数是什么？。

int（整型）

　　在32位机器上，整数的位数为32位，取值范围为-2\*\*31～2\*\*31-1，即-2147483648～2147483647  
　　在64位系统上，整数的位数为64位，取值范围为-2\*\*63～2\*\*63-1，即-9223372036854775808～9223372036854775807

long（长整型）　　跟C语言不同，Python的长整数没有指定位宽，即：Python没有限制长整数数值的大小，但实际上由于机器内存有限，我们使用的长整数数值不可能无限大。  
　　注意，自从Python2.2起，如果整数发生溢出，Python会自动将整数数据转换为长整数，所以如今在长整数数据后面不加字母L也不会导致严重后果了。  
float（浮点型）

*先扫盲 http://www.cnblogs.com/alex3714/articles/5895848.html*　　浮点数用来处理实数，即带有小数的数字。类似于C语言中的double类型，占8个字节（64位），其中52位表示底，11位表示指数，剩下的一位表示符号。  
complex（复数）  
　　复数由实数部分和虚数部分组成，一般形式为x＋yj，其中的x是复数的实数部分，y是复数的虚数部分，这里的x和y都是实数。

注：Python中存在小数字池：-5 ～ 257

2、布尔值

　　真或假

　　1 或 0

3、字符串

"hello world"

万恶的字符串拼接：

　　python中的字符串在C语言中体现为是一个字符数组，每次创建字符串时候需要在内存中开辟一块连续的空，并且一旦需要修改字符串的话，就需要再次开辟空间，万恶的+号每出现一次就会在内存中重新开辟一块空间。

**字符串格式化输出**

|  |  |
| --- | --- |
| 1  2  3  4 | name = "alex"  print "i am %s " % name    #输出: i am alex |

PS: 字符串是 %s;整数 %d;浮点数%f

字符串常用功能：

* 移除空白
* 分割
* 长度
* 索引
* 切片

4、列表

创建列表：

|  |  |
| --- | --- |
| 1  2  3 | name\_list = ['alex', 'seven', 'eric']  或  name\_list ＝ list(['alex', 'seven', 'eric']) |

基本操作：

* 索引
* 切片
* 追加
* 删除
* 长度
* 切片
* 循环
* 包含

5、元组(不可变列表)

创建元组：

|  |  |
| --- | --- |
| 1  2  3 | ages = (11, 22, 33, 44, 55)  或  ages = tuple((11, 22, 33, 44, 55)) |

6、字典（无序）

创建字典：

|  |  |
| --- | --- |
| 1  2  3 | person = {"name": "mr.wu", 'age': 18}  或  person = dict({"name": "mr.wu", 'age': 18}) |

常用操作：

* 索引
* 新增
* 删除
* 键、值、键值对
* 循环
* 长度

**十二、数据运算**

算数运算：



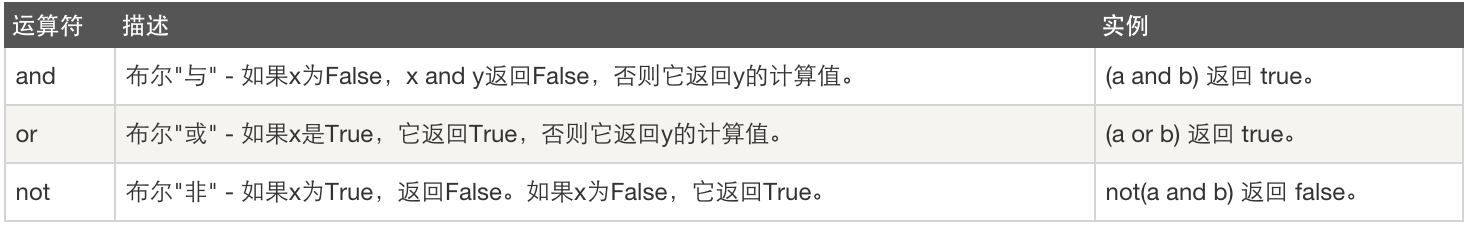
比较运算：



赋值运算：



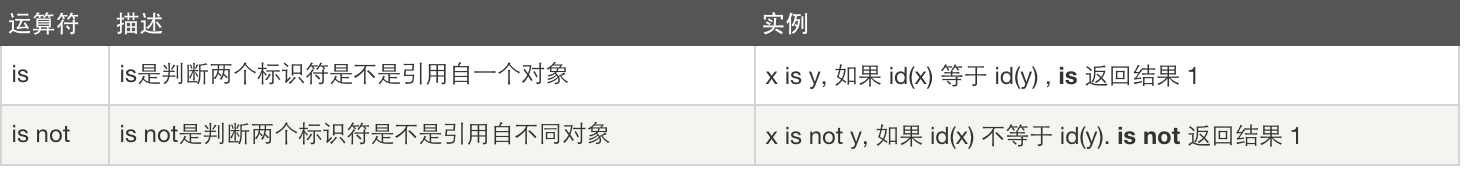
逻辑运算：



成员运算：



身份运算：



位运算：



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | #!/usr/bin/python    a = 60            # 60 = 0011 1100  b = 13            # 13 = 0000 1101  c = 0    c = a & b;        # 12 = 0000 1100  print "Line 1 - Value of c is ", c    c = a | b;        # 61 = 0011 1101  print "Line 2 - Value of c is ", c    c = a ^ b;        # 49 = 0011 0001 #相同为0，不同为1  print "Line 3 - Value of c is ", c    c = ~a;           # -61 = 1100 0011  print "Line 4 - Value of c is ", c    c = a << 2;       # 240 = 1111 0000  print "Line 5 - Value of c is ", c    c = a >> 2;       # 15 = 0000 1111  print "Line 6 - Value of c is ", c |

*\*按位取反运算规则(按位取反再加1)   详解http://blog.csdn.net/wenxinwukui234/article/details/42119265*

运算符优先级：



更多内容：http://www.runoob.com/python/python-operators.html

**十三、表达式if ... else**

场景一、用户登陆验证

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | # 提示输入用户名和密码    # 验证用户名和密码  #     如果错误，则输出用户名或密码错误  #     如果成功，则输出 欢迎，XXX!      #!/usr/bin/env python  # -\*- coding: encoding -\*-    import getpass      name = raw\_input('请输入用户名：')  pwd = getpass.getpass('请输入密码：')    if name == "alex" and pwd == "cmd":      print("欢迎，alex！")  else:      print("用户名和密码错误") |

场景二、猜年龄游戏

在程序里设定好你的年龄，然后启动程序让用户猜测，用户输入后，根据他的输入提示用户输入的是否正确，如果错误，提示是猜大了还是小了

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #!/usr/bin/env python  # -\*- coding: utf-8 -\*-      my\_age = 28    user\_input = int(input("input your guess num:"))    if user\_input == my\_age:      print("Congratulations, you got it !")  elif user\_input < my\_age:      print("Oops,think bigger!")  else:      print("think smaller!") |

外层变量，可以被内层代码使用

内层变量，不应被外层代码使用

**十四、表达式for loop**

**最简单的循环10次**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | #\_\*\_coding:utf-8\_\*\_  \_\_author\_\_ = 'Alex Li'      for i in range(10):      print("loop:", i ) |

输出：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | loop: 0  loop: 1  loop: 2  loop: 3  loop: 4  loop: 5  loop: 6  loop: 7  loop: 8  loop: 9 |

**需求一：还是上面的程序，但是遇到小于5的循环次数就不走了，直接跳入下一次循环**

|  |  |
| --- | --- |
| 1  2  3  4 | for i in range(10):      if i<5:          continue #不往下走了,直接进入下一次loop      print("loop:", i ) |

**需求二：还是上面的程序，但是遇到大于5的循环次数就不走了，直接退出**

|  |  |
| --- | --- |
| 1  2  3  4 | for i in range(10):      if i>5:          break #不往下走了,直接跳出整个loop      print("loop:", i ) |

**十五、while loop**

**有一种循环叫死循环，一经触发，就运行个天荒地老、海枯石烂。**

海枯石烂代码

|  |  |
| --- | --- |
| 1  2  3  4  5 | count = 0  while True:      print("你是风儿我是沙,缠缠绵绵到天涯...",count)      count +=1 |

其实除了时间，没有什么是永恒的，死loop还是少写为好

上面的代码循环100次就退出吧

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | count = 0  while True:      print("你是风儿我是沙,缠缠绵绵到天涯...",count)      count +=1      if count == 100:          print("去你妈的风和沙,你们这些脱了裤子是人,穿上裤子是鬼的臭男人..")          break |

**回到上面for 循环的例子，如何实现让用户不断的猜年龄，但只给最多3次机会，再猜不对就退出程序。**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #!/usr/bin/env python  # -\*- coding: utf-8 -\*-      my\_age = 28    count = 0  while count < 3:      user\_input = int(input("input your guess num:"))        if user\_input == my\_age:          print("Congratulations, you got it !")          break      elif user\_input < my\_age:          print("Oops,think bigger!")      else:          print("think smaller!")      count += 1 #每次loop 计数器+1  else:      print("猜这么多次都不对,你个笨蛋.") |

**十六、作业**

作业一：博客

作业二：编写登陆接口

* 输入用户名密码
* 认证成功后显示欢迎信息
* 输错三次后锁定

作业三：多级菜单

* 三级菜单
* 可依次选择进入各子菜单
* 所需新知识点：列表、字典

**入门知识拾遗**

**一、bytes类型**

 http://www.cnblogs.com/txw1958/archive/2012/07/19/2598885.html

**二、三元运算**

|  |  |
| --- | --- |
| 1 | result = 值1 if 条件 else 值2 |

如果条件为真：result = 值1  
如果条件为假：result = 值2

**三、进制**

* 二进制，01
* 八进制，01234567
* 十进制，0123456789
* 十六进制，0123456789ABCDEF  二进制到16进制转换http://jingyan.baidu.com/album/47a29f24292608c0142399cb.html?picindex=1

**计算机内存地址和为什么用16进制？**

**为什么用16进制**

1、计算机硬件是0101二进制的，16进制刚好是2的倍数，更容易表达一个命令或者数据。十六进制更简短，因为换算的时候一位16进制数可以顶4位2进制数，也就是一个字节（8位进制可以用两个16进制表示）

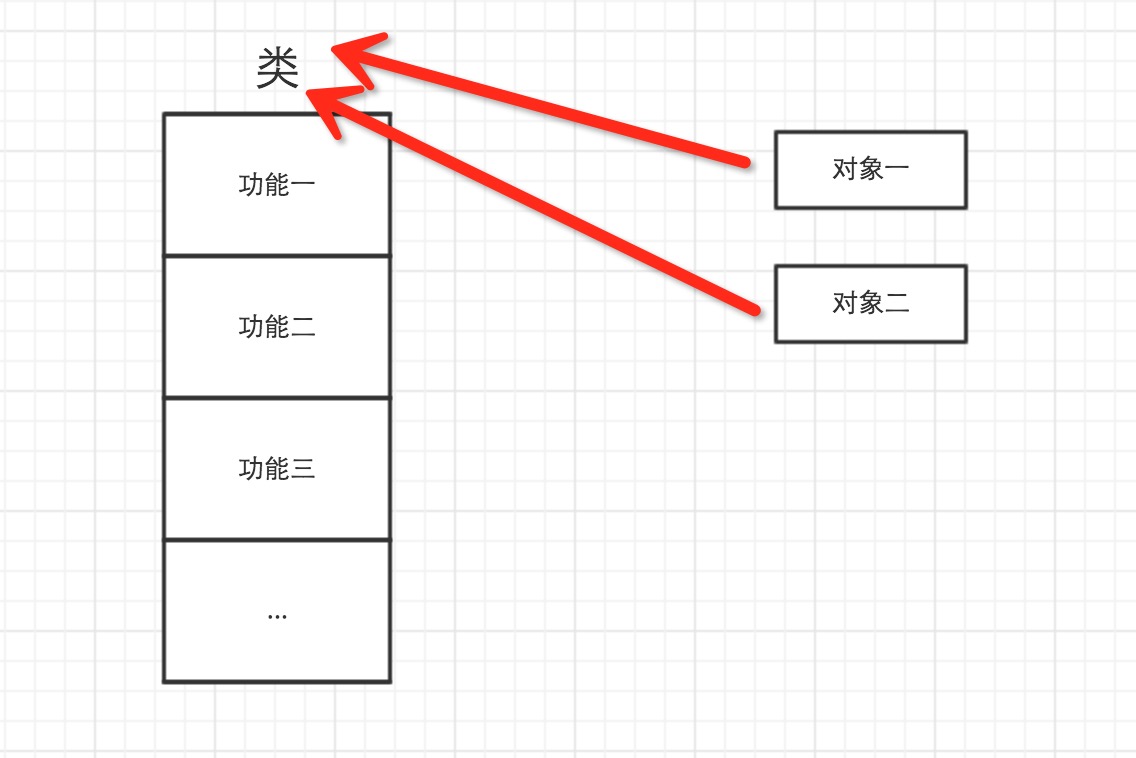
2、最早规定ASCII字符集采用的就是8bit(后期扩展了,但是基础单位还是8bit)，8bit用2个16进制直接就能表达出来，不管阅读还是存储都比其他进制要方便  
3、计算机中CPU运算也是遵照ASCII字符集，以16、32、64的这样的方式在发展，因此数据交换的时候16进制也显得更好  
4、为了统一规范，CPU、内存、硬盘我们看到都是采用的16进制计算

**16进制用在哪里**  
1、网络编程，数据交换的时候需要对字节进行解析都是一个byte一个byte的处理，1个byte可以用0xFF两个16进制来表达。通过网络抓包，可以看到数据是通过16进制传输的。  
2、数据存储，存储到硬件中是0101的方式，存储到系统中的表达方式都是byte方式

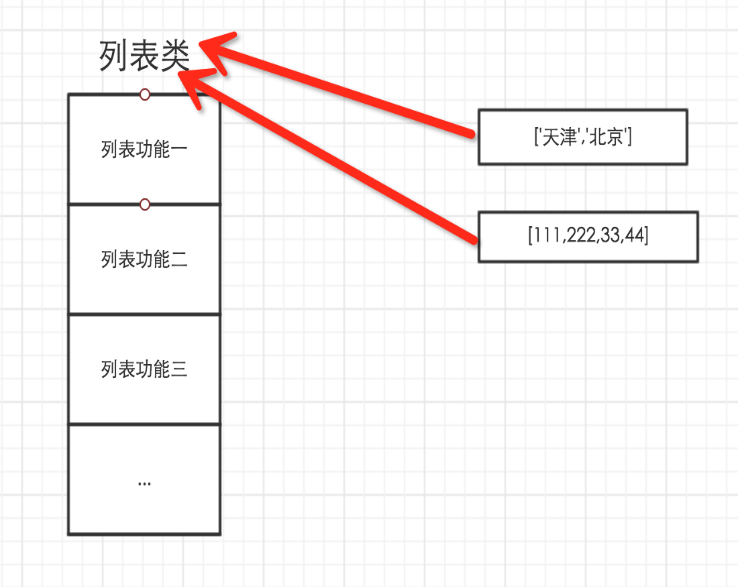
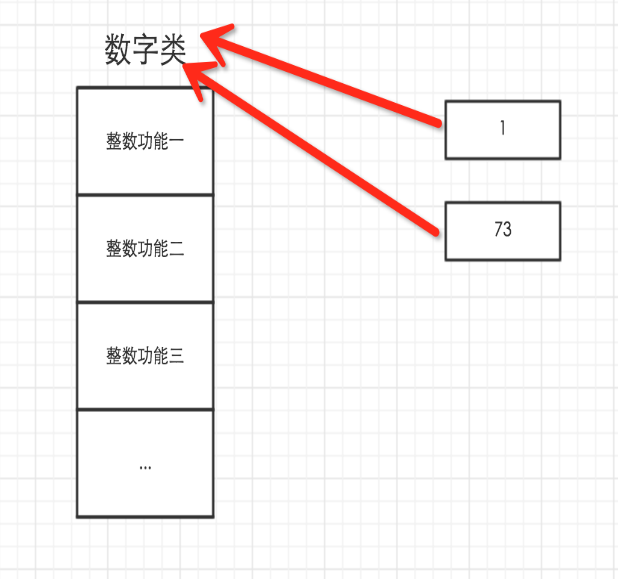
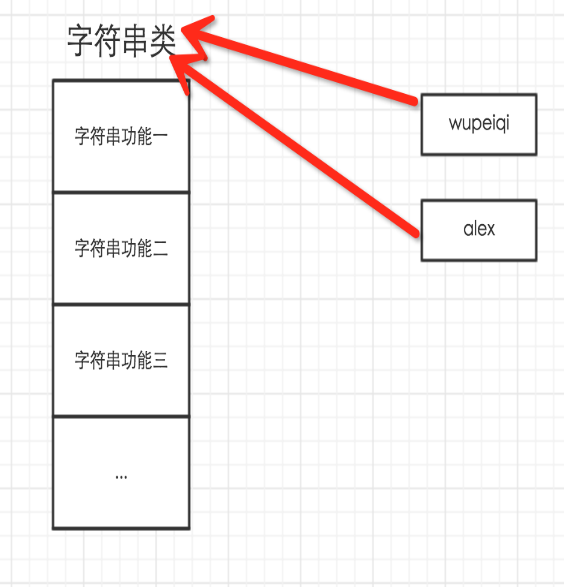
3、一些常用值的定义，比如：我们经常用到的html中color表达，就是用的16进制方式，4个16进制位可以表达好几百万的颜色信息。

**四、 一切皆对象**

**对于Python，一切事物都是对象，对象基于类创建**



所以，以下这些值都是对象： "wupeiqi"、38、['北京', '上海', '深圳']，并且是根据不同的类生成的对象。



# [Python之路,Day2 - Python基础2](http://www.cnblogs.com/alex3714/articles/5717620.html)

<http://www.cnblogs.com/alex3714/articles/5717620.html>

本节内容

1. 列表、元组操作
2. 字符串操作
3. 字典操作
4. 集合操作
5. 文件操作
6. 字符编码与转码

### 1. 列表、元组操作

列表是我们最以后最常用的数据类型之一，通过列表可以对数据实现最方便的存储、修改等操作

定义列表

|  |  |
| --- | --- |
| 1 | names = ['Alex',"Tenglan",'Eric'] |

通过下标访问列表中的元素，下标从0开始计数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | >>> names[0]  'Alex'  >>> names[2]  'Eric'  >>> names[-1]  'Eric'  >>> names[-2] #还可以倒着取  'Tenglan' |

**切片:取多个元素**

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

>>> names = ["Alex","Tenglan","Eric","Rain","Tom","Amy"]

>>> names[1:4] #取下标1至下标4之间的数字，包括1，不包括4

['Tenglan', 'Eric', 'Rain']

>>> names[1:-1] #取下标1至-1的值，不包括-1

['Tenglan', 'Eric', 'Rain', 'Tom']

>>> names[0:3]

['Alex', 'Tenglan', 'Eric']

>>> names[:3] #如果是从头开始取，0可以忽略，跟上句效果一样

['Alex', 'Tenglan', 'Eric']

>>> names[3:] #如果想取最后一个，必须不能写-1，只能这么写

['Rain', 'Tom', 'Amy']

>>> names[3:-1] #这样-1就不会被包含了

['Rain', 'Tom']

>>> names[0::2] #后面的2是代表，每隔一个元素，就取一个

['Alex', 'Eric', 'Tom']

>>> names[::2] #和上句效果一样

['Alex', 'Eric', 'Tom']

[复制代码](javascript:void(0);)

**追加**

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

>>> names

['Alex', 'Tenglan', 'Eric', 'Rain', 'Tom', 'Amy']

>>> names.append("我是新来的")

>>> names

['Alex', 'Tenglan', 'Eric', 'Rain', 'Tom', 'Amy', '我是新来的']

**插入**

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

>>> names

['Alex', 'Tenglan', 'Eric', 'Rain', 'Tom', 'Amy', '我是新来的']

>>> names.insert(2,"强行从Eric前面插入")

>>> names

['Alex', 'Tenglan', '强行从Eric前面插入', 'Eric', 'Rain', 'Tom', 'Amy', '我是新来的']

>>> names.insert(5,"从eric后面插入试试新姿势")

>>> names

['Alex', 'Tenglan', '强行从Eric前面插入', 'Eric', 'Rain', '从eric后面插入试试新姿势', 'Tom', 'Amy', '我是新来的']

[复制代码](javascript:void(0);)

**修改**

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

>>> names

['Alex', 'Tenglan', '强行从Eric前面插入', 'Eric', 'Rain', '从eric后面插入试试新姿势', 'Tom', 'Amy', '我是新来的']

>>> names[2] = "该换人了"

>>> names

['Alex', 'Tenglan', '该换人了', 'Eric', 'Rain', '从eric后面插入试试新姿势', 'Tom', 'Amy', '我是新来的']

**删除**

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

>>> del names[2]

>>> names

['Alex', 'Tenglan', 'Eric', 'Rain', '从eric后面插入试试新姿势', 'Tom', 'Amy', '我是新来的']

>>> del names[4]

>>> names

['Alex', 'Tenglan', 'Eric', 'Rain', 'Tom', 'Amy', '我是新来的']

>>>

>>> names.remove("Eric") #删除指定元素

>>> names

['Alex', 'Tenglan', 'Rain', 'Tom', 'Amy', '我是新来的']

>>> names.pop() #删除列表最后一个值

'我是新来的'

>>> names

['Alex', 'Tenglan', 'Rain', 'Tom', 'Amy']

[复制代码](javascript:void(0);)

**扩展**

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

>>> names

['Alex', 'Tenglan', 'Rain', 'Tom', 'Amy']

>>> b = [1,2,3]

>>> names.extend(b)

>>> names

['Alex', 'Tenglan', 'Rain', 'Tom', 'Amy', 1, 2, 3]

[复制代码](javascript:void(0);)

**拷贝**

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

>>> names

['Alex', 'Tenglan', 'Rain', 'Tom', 'Amy', 1, 2, 3]

>>> name\_copy = names.copy()

>>> name\_copy

['Alex', 'Tenglan', 'Rain', 'Tom', 'Amy', 1, 2, 3]

[复制代码](javascript:void(0);)

copy真的这么简单么？那我还讲个屁。。。

**统计**

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

>>> names

['Alex', 'Tenglan', 'Amy', 'Tom', 'Amy', 1, 2, 3]

>>> names.count("Amy")

2

**排序&翻转**

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

>>> names

['Alex', 'Tenglan', 'Amy', 'Tom', 'Amy', 1, 2, 3]

>>> names.sort() #排序

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unorderable types: int() < str() #3.0里不同数据类型不能放在一起排序了，擦

>>> names[-3] = '1'

>>> names[-2] = '2'

>>> names[-1] = '3'

>>> names

['Alex', 'Amy', 'Amy', 'Tenglan', 'Tom', '1', '2', '3']

>>> names.sort()

>>> names

['1', '2', '3', 'Alex', 'Amy', 'Amy', 'Tenglan', 'Tom']

>>> names.reverse() #反转

>>> names

['Tom', 'Tenglan', 'Amy', 'Amy', 'Alex', '3', '2', '1']

[复制代码](javascript:void(0);)

**获取下标**

**http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif**

**>>> names**

**['Tom', 'Tenglan', 'Amy', 'Amy', 'Alex', '3', '2', '1']**

**>>> names.index("Amy")**

**2 #只返回找到的第一个下标**

#### 元组

元组其实跟列表差不多，也是存一组数，只不是它一旦创建，便不能再修改，所以又叫只读列表

语法

|  |  |
| --- | --- |
| 1 | names = ("alex","jack","eric") |

它只有2个方法，一个是count,一个是index，完毕。

#### 程序练习

请闭眼写出以下程序。

程序：购物车程序

需求:

1. 启动程序后，让用户输入工资，然后打印商品列表
2. 允许用户根据商品编号购买商品
3. 用户选择商品后，检测余额是否够，够就直接扣款，不够就提醒
4. 可随时退出，退出时，打印已购买商品和余额

### ****2. 字符串操作****

**特性：不可修改**

[复制代码](javascript:void(0);)

name.capitalize() 首字母大写

name.casefold() 大写全部变小写

name.center(50,"-") 输出 '---------------------Alex Li----------------------'

name.count('lex') 统计 lex出现次数

name.encode() 将字符串编码成bytes格式

name.endswith("Li") 判断字符串是否以 Li结尾

"Alex\tLi".expandtabs(10) 输出'Alex Li'， 将\t转换成多长的空格

name.find('A') 查找A,找到返回其索引， 找不到返回-1

format :

>>> msg = "my name is {}, and age is {}"

>>> msg.format("alex",22)

'my name is alex, and age is 22'

>>> msg = "my name is {1}, and age is {0}"

>>> msg.format("alex",22)

'my name is 22, and age is alex'

>>> msg = "my name is {name}, and age is {age}"

>>> msg.format(age=22,name="ale")

'my name is ale, and age is 22'

format\_map

>>> msg.format\_map({'name':'alex','age':22})

'my name is alex, and age is 22'

msg.index('a') 返回a所在字符串的索引

'9aA'.isalnum() True

'9'.isdigit() 是否整数

name.isnumeric

name.isprintable

name.isspace

name.istitle

name.isupper

"|".join(['alex','jack','rain'])

'alex|jack|rain'

maketrans

>>> intab = "aeiou" #This is the string having actual characters.

>>> outtab = "12345" #This is the string having corresponding mapping character

>>> trantab = str.maketrans(intab, outtab)

>>>

>>> str = "this is string example....wow!!!"

>>> str.translate(trantab)

'th3s 3s str3ng 2x1mpl2....w4w!!!'

msg.partition('is') 输出 ('my name ', 'is', ' {name}, and age is {age}')

>>> "alex li, chinese name is lijie".replace("li","LI",1)

'alex LI, chinese name is lijie'

msg.swapcase 大小写互换

>>> msg.zfill(40)

'00000my name is {name}, and age is {age}'

>>> n4.ljust(40,"-")

'Hello 2orld-----------------------------'

>>> n4.rjust(40,"-")

'-----------------------------Hello 2orld'

>>> b="ddefdsdff\_哈哈"

>>> b.isidentifier() #检测一段字符串可否被当作标志符，即是否符合变量命名规则

True

[复制代码](javascript:void(0);)

### 3. 字典操作

字典一种key - value 的数据类型，使用就像我们上学用的字典，通过笔划、字母来查对应页的详细内容。

语法:

info = {

'stu1101': "TengLan Wu",

'stu1102': "LongZe Luola",

'stu1103': "XiaoZe Maliya",

}

字典的特性：

* dict是无序的
* key必须是唯一的,so 天生去重

增加

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

>>> info["stu1104"] = "苍井空"

>>> info

{'stu1102': 'LongZe Luola', 'stu1104': '苍井空', 'stu1103': 'XiaoZe Maliya', 'stu1101': 'TengLan Wu'}

修改

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

>>> info['stu1101'] = "武藤兰"

>>> info

{'stu1102': 'LongZe Luola', 'stu1103': 'XiaoZe Maliya', 'stu1101': '武藤兰'}

删除

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

>>> info

{'stu1102': 'LongZe Luola', 'stu1103': 'XiaoZe Maliya', 'stu1101': '武藤兰'}

>>> info.pop("stu1101") #标准删除姿势

'武藤兰'

>>> info

{'stu1102': 'LongZe Luola', 'stu1103': 'XiaoZe Maliya'}

>>> del info['stu1103'] #换个姿势删除

>>> info

{'stu1102': 'LongZe Luola'}

>>>

>>> info = {'stu1102': 'LongZe Luola', 'stu1103': 'XiaoZe Maliya'}

>>> info

{'stu1102': 'LongZe Luola', 'stu1103': 'XiaoZe Maliya'} #随机删除

>>> info.popitem()

('stu1102', 'LongZe Luola')

>>> info

{'stu1103': 'XiaoZe Maliya'}

[复制代码](javascript:void(0);)

查找

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

>>> info = {'stu1102': 'LongZe Luola', 'stu1103': 'XiaoZe Maliya'}

>>>

>>> "stu1102" in info #标准用法

True

>>> info.get("stu1102") #获取

'LongZe Luola'

>>> info["stu1102"] #同上，但是看下面

'LongZe Luola'

>>> info["stu1105"] #如果一个key不存在，就报错，get不会，不存在只返回None

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'stu1105'

[复制代码](javascript:void(0);)

多级字典嵌套及操作

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

av\_catalog = {

"欧美":{

"www.youporn.com": ["很多免费的,世界最大的","质量一般"],

"www.pornhub.com": ["很多免费的,也很大","质量比yourporn高点"],

"letmedothistoyou.com": ["多是自拍,高质量图片很多","资源不多,更新慢"],

"x-art.com":["质量很高,真的很高","全部收费,屌比请绕过"]

},

"日韩":{

"tokyo-hot":["质量怎样不清楚,个人已经不喜欢日韩范了","听说是收费的"]

},

"大陆":{

"1024":["全部免费,真好,好人一生平安","服务器在国外,慢"]

}

}

av\_catalog["大陆"]["1024"][1] += ",可以用爬虫爬下来"

print(av\_catalog["大陆"]["1024"])

#ouput

['全部免费,真好,好人一生平安', '服务器在国外,慢,可以用爬虫爬下来']

[复制代码](javascript:void(0);)

其它姿势

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

#values

>>> info.values()

dict\_values(['LongZe Luola', 'XiaoZe Maliya'])

#keys

>>> info.keys()

dict\_keys(['stu1102', 'stu1103'])

#setdefault

>>> info.setdefault("stu1106","Alex")

'Alex'

>>> info

{'stu1102': 'LongZe Luola', 'stu1103': 'XiaoZe Maliya', 'stu1106': 'Alex'}

>>> info.setdefault("stu1102","龙泽萝拉")

'LongZe Luola'

>>> info

{'stu1102': 'LongZe Luola', 'stu1103': 'XiaoZe Maliya', 'stu1106': 'Alex'}

#update

>>> info

{'stu1102': 'LongZe Luola', 'stu1103': 'XiaoZe Maliya', 'stu1106': 'Alex'}

>>> b = {1:2,3:4, "stu1102":"龙泽萝拉"}

>>> info.update(b)

>>> info

{'stu1102': '龙泽萝拉', 1: 2, 3: 4, 'stu1103': 'XiaoZe Maliya', 'stu1106': 'Alex'}

#items

info.items()

dict\_items([('stu1102', '龙泽萝拉'), (1, 2), (3, 4), ('stu1103', 'XiaoZe Maliya'), ('stu1106', 'Alex')])

#通过一个列表生成默认dict,有个没办法解释的坑，少用吧这个

>>> dict.fromkeys([1,2,3],'testd')

{1: 'testd', 2: 'testd', 3: 'testd'}

[复制代码](javascript:void(0);)

循环dict

[复制代码](javascript:void(0);)

#方法1

for key in info:

print(key,info[key])

#方法2

for k,v in info.items(): #会先把dict转成list,数据量大时莫用

print(k,v)

[复制代码](javascript:void(0);)

**程序练习**

程序: 三级菜单

要求:

1. 打印省、市、县三级菜单
2. 可返回上一级
3. 可随时退出程序

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

menu = {

'北京':{

'海淀':{

'五道口':{

'soho':{},

'网易':{},

'google':{}

},

'中关村':{

'爱奇艺':{},

'汽车之家':{},

'youku':{},

},

'上地':{

'百度':{},

},

},

'昌平':{

'沙河':{

'老男孩':{},

'北航':{},

},

'天通苑':{},

'回龙观':{},

},

'朝阳':{},

'东城':{},

},

'上海':{

'闵行':{

"人民广场":{

'炸鸡店':{}

}

},

'闸北':{

'火车战':{

'携程':{}

}

},

'浦东':{},

},

'山东':{},

}

exit\_flag = False

current\_layer = menu

layers = [menu]

while not exit\_flag:

for k in current\_layer:

print(k)

choice = input(">>:").strip()

if choice == "b":

current\_layer = layers[-1]

#print("change to laster", current\_layer)

layers.pop()

elif choice not in current\_layer:continue

else:

layers.append(current\_layer)

current\_layer = current\_layer[choice]

[复制代码](javascript:void(0);)

### 4.集合操作

集合是一个无序的，不重复的数据组合，它的主要作用如下：

* 去重，把一个列表变成集合，就自动去重了
* 关系测试，测试两组数据之前的交集、差集、并集等关系

常用操作

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

s = set([3,5,9,10]) #创建一个数值集合

t = set("Hello") #创建一个唯一字符的集合

a = t | s # t 和 s的并集

b = t & s # t 和 s的交集

c = t – s # 求差集（项在t中，但不在s中）

d = t ^ s # 对称差集（项在t或s中，但不会同时出现在二者中）

基本操作：

t.add('x') # 添加一项

s.update([10,37,42]) # 在s中添加多项

使用remove()可以删除一项：

t.remove('H')

len(s)

set 的长度

x in s

测试 x 是否是 s 的成员

x not in s

测试 x 是否不是 s 的成员

s.issubset(t)

s <= t

测试是否 s 中的每一个元素都在 t 中

s.issuperset(t)

s >= t

测试是否 t 中的每一个元素都在 s 中

s.union(t)

s | t

返回一个新的 set 包含 s 和 t 中的每一个元素

s.intersection(t)

s & t

返回一个新的 set 包含 s 和 t 中的公共元素

s.difference(t)

s - t

返回一个新的 set 包含 s 中有但是 t 中没有的元素

s.symmetric\_difference(t)

s ^ t

返回一个新的 set 包含 s 和 t 中不重复的元素

s.copy()

返回 set “s”的一个浅复制

[复制代码](javascript:void(0);)

### ****5. 文件操作****

**对文件操作流程**

1. 打开文件，得到文件句柄并赋值给一个变量
2. 通过句柄对文件进行操作
3. 关闭文件

现有文件如下

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64 | Somehow, it seems the love I knew was always the most destructive kind  不知为何，我经历的爱情总是最具毁灭性的的那种  Yesterday when I was young  昨日当我年少轻狂  The taste of life was sweet  生命的滋味是甜的  As rain upon my tongue  就如舌尖上的雨露  I teased at life as if it were a foolish game  我戏弄生命 视其为愚蠢的游戏  The way the evening breeze  就如夜晚的微风  May tease the candle flame  逗弄蜡烛的火苗  The thousand dreams I dreamed  我曾千万次梦见  The splendid things I planned  那些我计划的绚丽蓝图  I always built to last on weak and shifting sand  但我总是将之建筑在易逝的流沙上  I lived by night and shunned the naked light of day  我夜夜笙歌 逃避白昼赤裸的阳光  And only now I see how the time ran away  事到如今我才看清岁月是如何匆匆流逝  Yesterday when I was young  昨日当我年少轻狂  So many lovely songs were waiting to be sung  有那么多甜美的曲儿等我歌唱  So many wild pleasures lay in store for me  有那么多肆意的快乐等我享受  And so much pain my eyes refused to see  还有那么多痛苦 我的双眼却视而不见  I ran so fast that time and youth at last ran out  我飞快地奔走 最终时光与青春消逝殆尽  I never stopped to think what life was all about  我从未停下脚步去思考生命的意义  And every conversation that I can now recall  如今回想起的所有对话  Concerned itself with me and nothing else at all  除了和我相关的 什么都记不得了  The game of love I played with arrogance and pride  我用自负和傲慢玩着爱情的游戏  And every flame I lit too quickly, quickly died  所有我点燃的火焰都熄灭得太快  The friends I made all somehow seemed to slip away  所有我交的朋友似乎都不知不觉地离开了  And only now I'm left alone to end the play, yeah  只剩我一个人在台上来结束这场闹剧  Oh, yesterday when I was young  噢 昨日当我年少轻狂  So many, many songs were waiting to be sung  有那么那么多甜美的曲儿等我歌唱  So many wild pleasures lay in store for me  有那么多肆意的快乐等我享受  And so much pain my eyes refused to see  还有那么多痛苦 我的双眼却视而不见  There are so many songs in me that won't be sung  我有太多歌曲永远不会被唱起  I feel the bitter taste of tears upon my tongue  我尝到了舌尖泪水的苦涩滋味  The time has come for me to pay for yesterday  终于到了付出代价的时间 为了昨日  When I was young  当我年少轻狂 |

**基本操作**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | f = open('lyrics') #打开文件  first\_line = f.readline()  print('first line:',first\_line) #读一行  print('我是分隔线'.center(50,'-'))  data = f.read()# 读取剩下的所有内容,文件大时不要用  print(data) #打印文件    f.close() #关闭文件 |

打开文件的模式有：

* r，只读模式（默认）。
* w，只写模式。【不可读；不存在则创建；存在则删除内容；】
* a，追加模式。【可读；   不存在则创建；存在则只追加内容；】

"+" 表示可以同时读写某个文件

* r+，可读写文件。【可读；可写；可追加】
* w+，写读
* a+，同a

"U"表示在读取时，可以将 \r \n \r\n自动转换成 \n （与 r 或 r+ 模式同使用）

* rU
* r+U

"b"表示处理二进制文件（如：FTP发送上传ISO镜像文件，linux可忽略，windows处理二进制文件时需标注）

* rb
* wb
* ab

**其它语法**

[复制代码](javascript:void(0);)

def close(self): # real signature unknown; restored from \_\_doc\_\_

"""

Close the file.

A closed file cannot be used for further I/O operations. close() may be

called more than once without error.

"""

pass

def fileno(self, \*args, \*\*kwargs): # real signature unknown

""" Return the underlying file descriptor (an integer). """

pass

def isatty(self, \*args, \*\*kwargs): # real signature unknown

""" True if the file is connected to a TTY device. """

pass

def read(self, size=-1): # known case of \_io.FileIO.read

"""

注意，不一定能全读回来

Read at most size bytes, returned as bytes.

Only makes one system call, so less data may be returned than requested.

In non-blocking mode, returns None if no data is available.

Return an empty bytes object at EOF.

"""

return ""

def readable(self, \*args, \*\*kwargs): # real signature unknown

""" True if file was opened in a read mode. """

pass

def readall(self, \*args, \*\*kwargs): # real signature unknown

"""

Read all data from the file, returned as bytes.

In non-blocking mode, returns as much as is immediately available,

or None if no data is available. Return an empty bytes object at EOF.

"""

pass

def readinto(self): # real signature unknown; restored from \_\_doc\_\_

""" Same as RawIOBase.readinto(). """

pass #不要用,没人知道它是干嘛用的

def seek(self, \*args, \*\*kwargs): # real signature unknown

"""

Move to new file position and return the file position.

Argument offset is a byte count. Optional argument whence defaults to

SEEK\_SET or 0 (offset from start of file, offset should be >= 0); other values

are SEEK\_CUR or 1 (move relative to current position, positive or negative),

and SEEK\_END or 2 (move relative to end of file, usually negative, although

many platforms allow seeking beyond the end of a file).

Note that not all file objects are seekable.

"""

pass

def seekable(self, \*args, \*\*kwargs): # real signature unknown

""" True if file supports random-access. """

pass

def tell(self, \*args, \*\*kwargs): # real signature unknown

"""

Current file position.

Can raise OSError for non seekable files.

"""

pass

def truncate(self, \*args, \*\*kwargs): # real signature unknown

"""

Truncate the file to at most size bytes and return the truncated size.

Size defaults to the current file position, as returned by tell().

The current file position is changed to the value of size.

"""

pass

def writable(self, \*args, \*\*kwargs): # real signature unknown

""" True if file was opened in a write mode. """

pass

def write(self, \*args, \*\*kwargs): # real signature unknown

"""

Write bytes b to file, return number written.

Only makes one system call, so not all of the data may be written.

The number of bytes actually written is returned. In non-blocking mode,

returns None if the write would block.

"""

pass

[复制代码](javascript:void(0);)

**with语句**

为了避免打开文件后忘记关闭，可以通过管理上下文，即：

|  |  |
| --- | --- |
| 1  2  3 | with open('log','r') as f:        ... |

如此方式，当with代码块执行完毕时，内部会自动关闭并释放文件资源。

在Python 2.7 后，with又支持同时对多个文件的上下文进行管理，即：

|  |  |
| --- | --- |
| 1  2 | with open('log1') as obj1, open('log2') as obj2:      pass |

**程序练习**

**程序1: 实现简单的shell sed替换功能**

**程序2:修改haproxy配置文件**

**需求：**

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

1、查

输入：www.oldboy.org

获取当前backend下的所有记录

2、新建

输入：

arg = {

'bakend': 'www.oldboy.org',

'record':{

'server': '100.1.7.9',

'weight': 20,

'maxconn': 30

}

}

3、删除

输入：

arg = {

'bakend': 'www.oldboy.org',

'record':{

'server': '100.1.7.9',

'weight': 20,

'maxconn': 30

}

}

[复制代码](javascript:void(0);)

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

global

log 127.0.0.1 local2

daemon

maxconn 256

log 127.0.0.1 local2 info

defaults

log global

mode http

timeout connect 5000ms

timeout client 50000ms

timeout server 50000ms

option dontlognull

listen stats :8888

stats enable

stats uri /admin

stats auth admin:1234

frontend oldboy.org

bind 0.0.0.0:80

option httplog

option httpclose

option forwardfor

log global

acl www hdr\_reg(host) -i www.oldboy.org

use\_backend www.oldboy.org if www

backend www.oldboy.org

server 100.1.7.9 100.1.7.9 weight 20 maxconn 3000

[复制代码](javascript:void(0);)

### 6. 字符编码与转码

详细文章:

http://www.cnblogs.com/luotianshuai/articles/5735051.html

http://www.cnblogs.com/yuanchenqi/articles/5956943.html

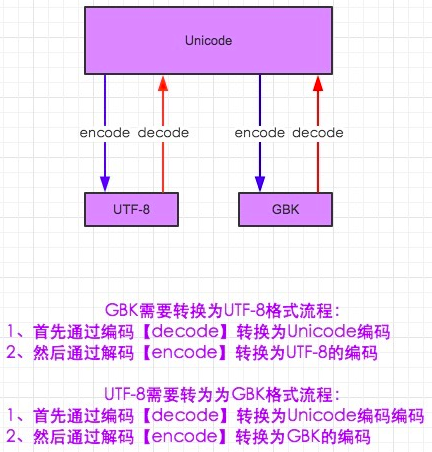
http://www.diveintopython3.net/strings.html

需知:

1.在python2默认编码是ASCII, python3里默认是unicode

2.unicode 分为 utf-32(占4个字节),utf-16(占两个字节)，utf-8(占1-4个字节)， so utf-16就是现在最常用的unicode版本， 不过在文件里存的还是utf-8，因为utf8省空间

3.在py3中encode,在转码的同时还会把string 变成bytes类型，decode在解码的同时还会把bytes变回string



 上图仅适用于py2

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

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

\_\_author\_\_ = 'Alex Li'

import sys

print(sys.getdefaultencoding())

msg = "我爱北京天安门"

msg\_gb2312 = msg.decode("utf-8").encode("gb2312")

gb2312\_to\_gbk = msg\_gb2312.decode("gbk").encode("gbk")

print(msg)

print(msg\_gb2312)

print(gb2312\_to\_gbk)

[复制代码](javascript:void(0);)

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

#-\*-coding:gb2312 -\*- #这个也可以去掉

\_\_author\_\_ = 'Alex Li'

import sys

print(sys.getdefaultencoding())

msg = "我爱北京天安门"

#msg\_gb2312 = msg.decode("utf-8").encode("gb2312")

msg\_gb2312 = msg.encode("gb2312") #默认就是unicode,不用再decode,喜大普奔

gb2312\_to\_unicode = msg\_gb2312.decode("gb2312")

gb2312\_to\_utf8 = msg\_gb2312.decode("gb2312").encode("utf-8")

print(msg)

print(msg\_gb2312)

print(gb2312\_to\_unicode)

print(gb2312\_to\_utf8)

[复制代码](javascript:void(0);)

7.  内置函数

# [Python之路,Day3 - Python基础3](http://www.cnblogs.com/alex3714/articles/5740985.html)

<http://www.cnblogs.com/alex3714/articles/5740985.html>

本节内容

1. 函数基本语法及特性

2. 参数与局部变量

3. 返回值

嵌套函数

4.递归

5.匿名函数

6.函数式编程介绍

7.高阶函数

8.内置函数

### 温故知新

#### 1. 集合

主要作用:

1. 去重
2. 关系测试， 交集＼差集＼并集＼反向(对称)差集

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | >>> a = {1,2,3,4}  >>> b ={3,4,5,6}  >>> a  {1, 2, 3, 4}  >>> type(a)  <class 'set'>  >>> a.symmetric\_difference(b)  {1, 2, 5, 6}  >>> b.symmetric\_difference(a)  {1, 2, 5, 6}  >>>  >>>  >>> a.difference(b)  {1, 2}  >>> a.union(b)  {1, 2, 3, 4, 5, 6}  >>> a.issu  a.issubset(   a.issuperset(  >>> a.issubset(b)  False |

#### 2. 元组

只读列表，只有count, index 2 个方法

作用：如果一些数据不想被人修改， 可以存成元组，比如身份证列表

#### 3. 字典

#### key-value对

1. 特性：
2. 无顺序
3. 去重
4. 查询速度快，比列表快多了
5. 比list占用内存多

为什么会查询速度会快呢？因为他是hash类型的，那什么是hash呢？

哈希算法将任意长度的二进制值映射为较短的固定长度的二进制值，这个小的二进制值称为哈希值。哈希值是一段数据唯一且极其紧凑的数值表示形式。如果散列一段明文而且哪怕只更改该段落的一个字母，随后的哈希都将产生不同的值。要找到散列为同一个值的两个不同的输入，在计算上是不可能的，所以数据的哈希值可以检验数据的完整性。一般用于快速查找和加密算法

dict会把所有的key变成hash 表，然后将这个表进行排序，这样，你通过data[key]去查data字典中一个key的时候，python会先把这个key hash成一个数字，然后拿这个数字到hash表中看没有这个数字， 如果有，拿到这个key在hash表中的索引，拿到这个索引去与此key对应的value的内存地址那取值就可以了。

上面依然没回答这样做查找一个数据为什么会比列表快，对不对？ 呵呵，等我课上揭晓。

#### 4. 字符编码

**先说python2**

1. py2里默认编码是ascii
2. 文件开头那个编码声明是告诉解释这个代码的程序 以什么编码格式 把这段代码读入到内存，因为到了内存里，这段代码其实是以bytes二进制格式存的，不过即使是2进制流，也可以按不同的编码格式转成2进制流，你懂么？
3. 如果在文件头声明了#\_\*\_coding:utf-8\*\_，就可以写中文了， 不声明的话，python在处理这段代码时按ascii，显然会出错， 加了这个声明后，里面的代码就全是utf-8格式了
4. 在有#\_\*\_coding:utf-8\*\_的情况下，你在声明变量如果写成name=u"大保健"，那这个字符就是unicode格式，不加这个u,那你声明的字符串就是utf-8格式
5. utf-8 to gbk怎么转，utf8先decode成unicode,再encode成gbk

**再说python3**

1. py3里默认文件编码就是utf-8,所以可以直接写中文，也不需要文件头声明编码了，干的漂亮
2. 你声明的变量默认是unicode编码，不是utf-8, 因为默认即是unicode了（不像在py2里，你想直接声明成unicode还得在变量前加个u）, 此时你想转成gbk的话，直接your\_str.encode("gbk")即可以
3. 但py3里，你在your\_str.encode("gbk")时，感觉好像还加了一个动作，就是就是encode的数据变成了bytes里，我擦，这是怎么个情况，因为在py3里，str and bytes做了明确的区分，你可以理解为bytes就是2进制流，你会说，我看到的不是010101这样的2进制呀， 那是因为python为了让你能对数据进行操作而在内存级别又帮你做了一层封装，否则让你直接看到一堆2进制，你能看出哪个字符对应哪段2进制么？什么？自己换算，得了吧，你连超过2位数的数字加减运算都费劲，还还是省省心吧。
4. 那你说，在py2里好像也有bytes呀，是的，不过py2里的bytes只是对str做了个别名(python2里的str就是bytes, py3里的str是unicode)，没有像py3一样给你显示的多出来一层封装，但其实其内部还是封装了的。 这么讲吧， 无论是2还是三， 从硬盘到内存，数据格式都是 010101二进制到-->b'\xe4\xbd\xa0\xe5\xa5\xbd' bytes类型－－>按照指定编码转成你能看懂的文字

编码应用比较多的场景应该是爬虫了，互联网上很多网站用的编码格式很杂，虽然整体趋向都变成utf-8，但现在还是很杂，所以爬网页时就需要你进行各种编码的转换，不过生活正在变美好，期待一个不需要转码的世界。

最后，编码is a piece of fucking shit, noboby likes it.

### 1.函数基本语法及特性

#### 背景提要

现在老板让你写一个监控程序，监控服务器的系统状况，当cpu＼memory＼disk等指标的使用量超过阀值时即发邮件报警，你掏空了所有的知识量，写出了以下代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | while True：      if cpu利用率 > 90%:          #发送邮件提醒          连接邮箱服务器          发送邮件          关闭连接        if 硬盘使用空间 > 90%:          #发送邮件提醒          连接邮箱服务器          发送邮件          关闭连接        if 内存占用 > 80%:          #发送邮件提醒          连接邮箱服务器          发送邮件          关闭连接 |

上面的代码实现了功能，但即使是邻居老王也看出了端倪，老王亲切的摸了下你家儿子的脸蛋，说，你这个重复代码太多了，每次报警都要重写一段发邮件的代码，太low了，这样干存在2个问题：

1. 代码重复过多，一个劲的copy and paste不符合高端程序员的气质
2. 如果日后需要修改发邮件的这段代码，比如加入群发功能，那你就需要在所有用到这段代码的地方都修改一遍

你觉得老王说的对，你也不想写重复代码，但又不知道怎么搞，老王好像看出了你的心思，此时他抱起你儿子，笑着说，其实很简单，只需要把重复的代码提取出来，放在一个公共的地方，起个名字，以后谁想用这段代码，就通过这个名字调用就行了，如下

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | 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') |

你看着老王写的代码，气势恢宏、磅礴大气，代码里透露着一股内敛的傲气，心想，老王这个人真是不一般，突然对他的背景更感兴趣了，问老王，这些花式玩法你都是怎么知道的？ 老王亲了一口你儿子，捋了捋不存在的胡子，淡淡的讲，“老夫，年少时，师从京西沙河淫魔银角大王 ”， 你一听“银角大王”这几个字，不由的娇躯一震，心想，真nb,怪不得代码写的这么6, 这“银角大王”当年在江湖上可是数得着的响当当的名字，只可惜后期纵欲过度，卒于公元2016年， 真是可惜了，只留下其哥哥孤守当年兄弟俩一起打下来的江山。 此时你看着的老王离开的身影，感觉你儿子跟他越来越像了。。。

#### 函数是什么?

函数一词来源于数学，但编程中的「函数」概念，与数学中的函数是有很大不同的，具体区别，我们后面会讲，编程中的函数在英文中也有很多不同的叫法。在BASIC中叫做subroutine(子过程或子程序)，在Pascal中叫做procedure(过程)和function，在C中只有function，在Java里面叫做method。

#### 定义: 函数是指将一组语句的集合通过一个名字(函数名)封装起来，要想执行这个函数，只需调用其函数名即可

**特性:**

1. 减少重复代码
2. 使程序变的可扩展
3. 使程序变得易维护

#### 语法定义

|  |  |
| --- | --- |
| 1  2  3  4 | def sayhi():#函数名      print("Hello, I'm nobody!")    sayhi() #调用函数 |

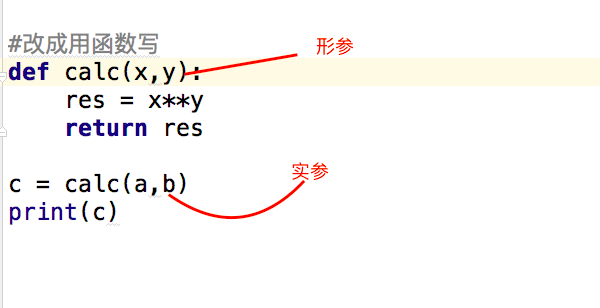
可以带参数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | #下面这段代码  a,b = 5,8  c = a\*\*b  print(c)      #改成用函数写  def calc(x,y):      res = x\*\*y      return res #返回函数执行结果    c = calc(a,b) ＃结果赋值给c变量  print(c) |

### 2.函数参数与局部变量

形参变量只有在被调用时才分配内存单元，在调用结束时，即刻释放所分配的内存单元。因此，形参只在函数内部有效。函数调用结束返回主调用函数后则不能再使用该形参变量

实参可以是常量、变量、表达式、函数等，无论实参是何种类型的量，在进行函数调用时，它们都必须有确定的值，以便把这些值传送给形参。因此应预先用赋值，输入等办法使参数获得确定值



**默认参数**

**看下面代码**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def stu\_register(name,age,country,course):      print("----注册学生信息------")      print("姓名:",name)      print("age:",age)      print("国籍:",country)      print("课程:",course)    stu\_register("王山炮",22,"CN","python\_devops")  stu\_register("张叫春",21,"CN","linux")  stu\_register("刘老根",25,"CN","linux") |

发现 country 这个参数 基本都 是"CN", 就像我们在网站上注册用户，像国籍这种信息，你不填写，默认就会是 中国， 这就是通过默认参数实现的，把country变成默认参数非常简单

|  |  |
| --- | --- |
| 1 | def stu\_register(name,age,course,country="CN"): |

这样，这个参数在调用时不指定，那默认就是CN，指定了的话，就用你指定的值。

另外，你可能注意到了，在把country变成默认参数后，我同时把它的位置移到了最后面，为什么呢？

#### 关键参数

正常情况下，给函数传参数要按顺序，不想按顺序就可以用关键参数，只需指定参数名即可，***但记住一个要求就是，关键参数必须放在位置参数之后。***

|  |  |
| --- | --- |
| 1 | stu\_register(age=22,name='alex',course="python",) |

**非固定参数**

若你的函数在定义时不确定用户想传入多少个参数，就可以使用非固定参数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def stu\_register(name,age,\*args): # \*args 会把多传入的参数变成一个元组形式      print(name,age,args)    stu\_register("Alex",22)  #输出  #Alex 22 () #后面这个()就是args,只是因为没传值,所以为空    stu\_register("Jack",32,"CN","Python")  #输出  # Jack 32 ('CN', 'Python') |

还可以有一个\*\*kwargs

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def stu\_register(name,age,\*args,\*\*kwargs): # \*kwargs 会把多传入的参数变成一个dict形式      print(name,age,args,kwargs)    stu\_register("Alex",22)  #输出  #Alex 22 () {}#后面这个{}就是kwargs,只是因为没传值,所以为空    stu\_register("Jack",32,"CN","Python",sex="Male",province="ShanDong")  #输出  # Jack 32 ('CN', 'Python') {'province': 'ShanDong', 'sex': 'Male'} |

#### 局部变量

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | name = "Alex Li"    def change\_name(name):      print("before change:",name)      name = "金角大王,一个有Tesla的男人"      print("after change", name)      change\_name(name)    print("在外面看看name改了么?",name) |

输出

|  |  |
| --- | --- |
| 1  2  3 | before change: Alex Li  after change 金角大王,一个有Tesla的男人  在外面看看name改了么? Alex Li |

**全局与局部变量**

在子程序中定义的变量称为局部变量，在程序的一开始定义的变量称为全局变量。

全局变量作用域是整个程序，局部变量作用域是定义该变量的子程序。

当全局变量与局部变量同名时：

在定义局部变量的子程序内，局部变量起作用；在其它地方全局变量起作用。

### 3.返回值

要想获取函数的执行结果，就可以用return语句把结果返回

注意:

1. 函数在执行过程中只要遇到return语句，就会停止执行并返回结果，so 也可以理解为 return 语句代表着函数的结束
2. 如果未在函数中指定return,那这个函数的返回值为None

### 强行插入知识点: 嵌套函数

看上面的标题的意思是，函数还能套函数？of course

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | name = "Alex"    def change\_name():      name = "Alex2"        def change\_name2():          name = "Alex3"          print("第3层打印",name)        change\_name2() #调用内层函数      print("第2层打印",name)      change\_name()  print("最外层打印",name) |

此时，在最外层调用change\_name2()会出现什么效果？

没错， 出错了， 为什么呢？

嵌套函数的用法会了，但它有什么用呢？下节课揭晓。。。

### 4. 递归

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | def calc(n):      print(n)      if int(n/2) ==0:          return n      return calc(int(n/2))    calc(10)    输出：  10  5  2  1 |

递归特性:

1. 必须有一个明确的结束条件

2. 每次进入更深一层递归时，问题规模相比上次递归都应有所减少

3. 递归效率不高，递归层次过多会导致栈溢出（在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出）

堆栈扫盲http://www.cnblogs.com/lln7777/archive/2012/03/14/2396164.html

递归函数实际应用案例，二分查找

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | data = [1, 3, 6, 7, 9, 12, 14, 16, 17, 18, 20, 21, 22, 23, 30, 32, 33, 35]      def binary\_search(dataset,find\_num):      print(dataset)        if len(dataset) >1:          mid = int(len(dataset)/2)          if dataset[mid] == find\_num:  #find it              print("找到数字",dataset[mid])          elif dataset[mid] > find\_num :# 找的数在mid左面              print("\033[31;1m找的数在mid[%s]左面\033[0m" % dataset[mid])              return binary\_search(dataset[0:mid], find\_num)          else:# 找的数在mid右面              print("\033[32;1m找的数在mid[%s]右面\033[0m" % dataset[mid])              return binary\_search(dataset[mid+1:],find\_num)      else:          if dataset[0] == find\_num:  #find it              print("找到数字啦",dataset[0])          else:              print("没的分了,要找的数字[%s]不在列表里" % find\_num)      binary\_search(data,66) |

### 5. 匿名函数

匿名函数就是不需要显式的指定函数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | #这段代码  def calc(n):      return n\*\*n  print(calc(10))    #换成匿名函数  calc = lambda n:n\*\*n  print(calc(10)) |

你也许会说，用上这个东西没感觉有毛方便呀， 。。。。呵呵，如果是这么用，确实没毛线改进，不过匿名函数主要是和其它函数搭配使用的呢，如下

|  |  |
| --- | --- |
| 1  2  3 | res = map(lambda x:x\*\*2,[1,5,7,4,8])  for i in res:      print(i) |

输出

1  
25  
49  
16  
64

### 6.函数式编程介绍

函数是Python内建支持的一种封装，我们通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之为面向过程的程序设计。函数就是面向过程的程序设计的基本单元。

**函数式编程中的函数这个术语不是指计算机中的函数（实际上是Subroutine），而是指数学中的函数，即自变量的映射。也就是说一个函数的值仅决定于函数参数的值，不依赖其他状态。比如sqrt(x)函数计算x的平方根，只要x不变，不论什么时候调用，调用几次，值都是不变的。**

Python对函数式编程提供部分支持。由于Python允许使用变量，因此，Python不是纯函数式编程语言。

一、定义

简单说，"函数式编程"是一种["编程范式"](http://en.wikipedia.org/wiki/Programming_paradigm)（programming paradigm），也就是如何编写程序的方法论。

主要思想是把运算过程尽量写成一系列嵌套的函数调用。举例来说，现在有这样一个数学表达式：

　　(1 + 2) \* 3 - 4

传统的过程式编程，可能这样写：

　　var a = 1 + 2;

　　var b = a \* 3;

　　var c = b - 4;

函数式编程要求使用函数，我们可以把运算过程定义为不同的函数，然后写成下面这样：

　　var result = subtract(multiply(add(1,2), 3), 4);

这段代码再演进以下，可以变成这样

add(1,2).multiply(3).subtract(4)

这基本就是自然语言的表达了。再看下面的代码，大家应该一眼就能明白它的意思吧：

merge([1,2],[3,4]).sort().search("2")

因此，函数式编程的代码更容易理解。

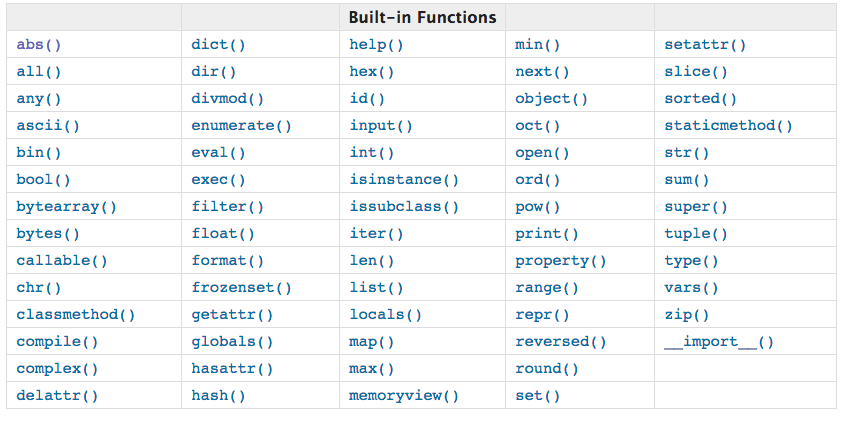
要想学好函数式编程，不要玩py,玩Erlang,Haskell, 好了，我只会这么多了。。。

### 7.高阶函数

变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | def add(x,y,f):      return f(x) + f(y)      res = add(3,-6,abs)  print(res) |

### 8. 内置参数



内置参数详解 https://docs.python.org/3/library/functions.html?highlight=built#ascii

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

#compile

f = open("函数递归.py")

data =compile(f.read(),'','exec')

exec(data)

#print

msg = "又回到最初的起点"

f = open("tofile","w")

print(msg,"记忆中你青涩的脸",sep="|",end="",file=f)

# #slice

# a = range(20)

# pattern = slice(3,8,2)

# for i in a[pattern]: #等于a[3:8:2]

# print(i)

#

#

#memoryview

#usage:

#>>> memoryview(b'abcd')

#<memory at 0x104069648>

#在进行切片并赋值数据时，不需要重新copy原列表数据，可以直接映射原数据内存，

import time

for n in (100000, 200000, 300000, 400000):

data = b'x'\*n

start = time.time()

b = data

while b:

b = b[1:]

print('bytes', n, time.time()-start)

for n in (100000, 200000, 300000, 400000):

data = b'x'\*n

start = time.time()

b = memoryview(data)

while b:

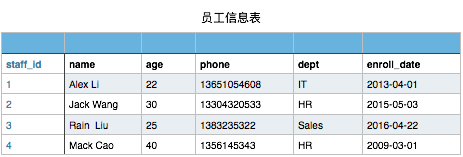
b = b[1:]

print('memoryview', n, time.time()-start)

[复制代码](javascript:void(0);)

### 本节作业

有以下员工信息表



当然此表你在文件存储时可以这样表示

|  |  |
| --- | --- |
| 1 | 1,Alex Li,22,13651054608,IT,2013-04-01 |

现需要对这个员工信息文件，实现增删改查操作

1. 可进行模糊查询，语法至少支持下面3种:
   1. select name,age from staff\_table where age > 22
   2. select  \* from staff\_table where dept = "IT"
   3. select  \* from staff\_table where enroll\_date like "2013"
   4. 查到的信息，打印后，最后面还要显示查到的条数
2. 可创建新员工纪录，以phone做唯一键，staff\_id需自增
3. 可删除指定员工信息纪录，输入员工id，即可删除
4. 可修改员工信息，语法如下:
   1. UPDATE staff\_table SET dept="Market" WHERE where dept = "IT"

**注意：以上需求，要充分使用函数，请尽你的最大限度来减少重复代码！**

# [Python之路,Day4](http://www.cnblogs.com/alex3714/articles/5143440.html)

<http://www.cnblogs.com/alex3714/articles/5143440.html>

**本节大纲**

1. 迭代器&生成器
2. 装饰器
   1. 基本装饰器
   2. 多参数装饰器
3. 递归
4. 算法基础：二分查找、二维数组转换
5. 正则表达式
6. 常用模块学习
7. 作业：计算器开发
   1. 实现加减乘除及拓号优先级解析
   2. 用户输入 1 - 2 \* ( (60-30 +(-40/5) \* (9-2\*5/3 + 7 /3\*99/4\*2998 +10 \* 568/14 )) - (-4\*3)/ (16-3\*2) )等类似公式后，必须自己解析里面的(),+,-,\*,/符号和公式，运算后得出结果，结果必须与真实的计算器所得出的结果一致

# ****迭代器&生成器****

**迭代器**

迭代器是访问集合元素的一种方式。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退，不过这也没什么，因为人们很少在迭代途中往后退。另外，迭代器的一大优点是不要求事先准备好整个迭代过程中所有的元素。迭代器仅仅在迭代到某个元素时才计算该元素，而在这之前或之后，元素可以不存在或者被销毁。这个特点使得它特别适合用于遍历一些巨大的或是无限的集合，比如几个G的文件

特点：

1. 访问者不需要关心迭代器内部的结构，仅需通过next()方法不断去取下一个内容
2. 不能随机访问集合中的某个值 ，只能从头到尾依次访问
3. 访问到一半时不能往回退
4. 便于循环比较大的数据集合，节省内存

生成一个迭代器：

[复制代码](javascript:void(0);)

>>> a = iter([1,2,3,4,5])

>>> a

<list\_iterator object at 0x101402630>

>>> a.\_\_next\_\_()

1

>>> a.\_\_next\_\_()

2

>>> a.\_\_next\_\_()

3

>>> a.\_\_next\_\_()

4

>>> a.\_\_next\_\_()

5

>>> a.\_\_next\_\_()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

[复制代码](javascript:void(0);)

Repeated calls to the iterator’s [\_\_next\_\_()](https://docs.python.org/3/library/stdtypes.html#iterator.__next__) method (or passing it to the built-in function [next()](https://docs.python.org/3/library/functions.html#next)) return successive items in the stream. When no more data are available a [StopIteration](https://docs.python.org/3/library/exceptions.html#StopIteration) exception is raised instead. At this point, the iterator object is exhausted and any further calls to its \_\_next\_\_() method just raise [StopIteration](https://docs.python.org/3/library/exceptions.html#StopIteration) again.

**生成器generator**

定义：一个函数调用时返回一个迭代器，那这个函数就叫做生成器（generator），如果函数中包含yield语法，那这个函数就会变成生成器

代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | def cash\_out(amount):      while amount >0:          amount -= 1          yield 1          print("擦，又来取钱了。。。败家子！")        ATM = cash\_out(5)    print("取到钱 %s 万" % ATM.\_\_next\_\_())  print("花掉花掉!")  print("取到钱 %s 万" % ATM.\_\_next\_\_())  print("取到钱 %s 万" % ATM.\_\_next\_\_())  print("花掉花掉!")  print("取到钱 %s 万" % ATM.\_\_next\_\_())  print("取到钱 %s 万" % ATM.\_\_next\_\_())  print("取到钱 %s 万" % ATM.\_\_next\_\_()) #到这时钱就取没了,再取就报错了  print("取到钱 %s 万" % ATM.\_\_next\_\_()) |

作用：

这个yield的主要效果呢，就是可以使函数中断，并保存中断状态，中断后，代码可以继续往下执行，过一段时间还可以再重新调用这个函数，从上次yield的下一句开始执行。

另外，还可通过yield实现在单线程的情况下实现并发运算的效果

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | import time  def consumer(name):      print("%s 准备吃包子啦!" %name)      while True:         baozi = yield           print("包子[%s]来了,被[%s]吃了!" %(baozi,name))    def producer(name):      c = consumer('A')      c2 = consumer('B')      c.\_\_next\_\_()      c2.\_\_next\_\_()      print("老子开始准备做包子啦!")      for i in range(10):          time.sleep(1)          print("做了2个包子!")          c.send(i)          c2.send(i)    producer("alex") |

**装饰器**

直接 看银角大王写的文档 http://www.cnblogs.com/wupeiqi/articles/4980620.html

**递归**

### 特点

递归算法是一种直接或者间接地调用自身算法的过程。在计算机编写程序中，递归算法对解决一大类问题是十分有效的，它往往使算法的描述简洁而且易于理解。

递归算法解决问题的特点：

(1) 递归就是在过程或函数里调用自身。

(2) 在使用递归策略时，必须有一个明确的递归结束条件，称为递归出口。

(3) 递归算法解题通常显得很简洁，但递归算法解题的运行效率较低。所以一般不提倡用递归算法设计程序。

(4) 在[递归调用](http://baike.baidu.com/view/1265506.htm)的过程当中系统为每一层的返回点、局部量等开辟了栈来存储。递归次数过多容易造成[栈溢出](http://baike.baidu.com/view/2061755.htm)等。所以一般不提倡用递归算法设计程序。

### 要求

递归算法所体现的“重复”一般有三个要求：

一是每次调用在规模上都有所缩小(通常是减半)；

二是相邻两次重复之间有紧密的联系，前一次要为后一次做准备(通常前一次的输出就作为后一次的输入)；

三是在问题的规模极小时必须用直接给出解答而不再进行[递归调用](http://baike.baidu.com/view/1265506.htm)，因而每次递归调用都是有条件的(以规模未达到直接解答的大小为条件)，无条件递归调用将会成为死循环而不能正常结束。

实现

1. 通过递归实现2分查找

　　现有列表 primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]， 要求尔等 用最快的方式 找出**23** 。 请Low B, Low 2B ,Low 3B 三个同学来回答这个问题。

　　Low B: 这个很简单，直接用 if 41 in primes:print("found it!") , 话音未落就被老师打了，让你自己实现，不是让你用现成提供的功能， Low B于是说，那只能从头开始一个个数了，然后Low B被 开除了。。。

　　Low 2B: 因为这个列表是有序的， 我可以把列表从中截取一半，大概如下：

　　　　p1 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,41]

　　　　p2 = [ 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

　　　　然后看p1[-1]也就是41是否比23大， 如果比23大就代表23肯定在p1里面，否则那就肯定在p2里面。现在我们知道23比41小，所以23肯定在p1里，但p1里依然有很多元素， 怎么找到23呢？很简单，依然按上一次的方法，把p1分成2部分，如下：

　　　　p1\_a = [2, 3, 5, 7, 11, 13,17]

　　　　p1\_b = [19, 23, 29, 31, 37,41]

　　　　然后我们发现，23 比p1\_a最后一个值 17 大，那代表23肯定在p1\_b中， p1\_b中依然有很多元素，那就再按之前的方法继续分半，最终用不了几次，肯定就把23找出来了！

　　　　说完，Low 2B满有成就感的甩了下头上的头皮屑。

　　老师：很好，确实较Low B的方案强很多。 然后转头问Low 3B ，你有更好的想法 么？

      Low 3B: 啊。。。噢 ，我。。。我跟Low 2B的想法一样，结果被他说了。

      老师：噢，那你帮我把代码写出来吧。

      Low 3B此时冷汗直冒，因为他根本没思路，但还是硬着头皮去写了。。。。虽然自己没思路，但是会谷歌呀，三个小时过去了，终于憋出了以下代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | def binary\_search(data\_list,find\_num):      mid\_pos = int(len(data\_list) /2 ) #find the middle position of the list      mid\_val = data\_list[mid\_pos] # get the value by it's position      print(data\_list)      if len(data\_list) >1:          if mid\_val > find\_num: # means the find\_num is in left hand of mid\_val              print("[%s] should be in left of [%s]" %(find\_num,mid\_val))              binary\_search(data\_list[:mid\_pos],find\_num)          elif mid\_val < find\_num: # means the find\_num is in the right hand of mid\_val              print("[%s] should be in right of [%s]" %(find\_num,mid\_val))              binary\_search(data\_list[mid\_pos:],find\_num)          else: # means the mid\_val == find\_num              print("Find ", find\_num)        else:          print("cannot find [%s] in data\_list" %find\_num)    if \_\_name\_\_ == '\_\_main\_\_':      primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]      binary\_search(primes,67) |

　　在后面的故事我就编不下去啦，哈哈！but anyway,以上就是典型的递归用法，在程序里自己调用自己。

### ****算法基础****

要求：生成一个4\*4的2维数组并将其顺时针旋转90度

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | #!\_\*\_coding:utf-8\_\*\_      array=[[col for col in range(5)] for row in range(5)] #初始化一个4\*4数组  #array=[[col for col in 'abcde'] for row in range(5)]    for row in array: #旋转前先看看数组长啥样      print(row)    print('-------------')  for i,row in enumerate(array):        for index in range(i,len(row)):          tmp = array[index][i] #get each rows' data by column's index          array[index][i] = array[i][index] #          print tmp,array[i][index]  #= tmp          array[i][index] = tmp      for r in array:print r        print('--one big loop --') |

**冒泡排序**

将一个不规则的数组按从小到大的顺序进行排序

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | data = [10,4,33,21,54,3,8,11,5,22,2,1,17,13,6]    print("before sort:",data)    previous = data[0]  for j in range(len(data)):      tmp = 0      for i in range(len(data)-1):          if data[i] > data[i+1]:              tmp=data[i]              data[i] = data[i+1]              data[i+1] = tmp      print(data)    print("after sort:",data) |

**时间复杂度   
（1）时间频度** 一个算法执行所耗费的时间，从理论上是不能算出来的，必须上机运行测试才能知道。但我们不可能也没有必要对每个算法都上机测试，只需知道哪个算法花费的时间多，哪个算法花费的时间少就可以了。并且一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。一个算法中的语句执行次数称为语句频度或时间频度。记为T(n)。

**（2）时间复杂度** 在刚才提到的时间频度中，n称为问题的规模，当n不断变化时，时间频度T(n)也会不断变化。但有时我们想知道它变化时呈现什么规律。为此，我们引入时间复杂度概念。 一般情况下，算法中基本操作重复执行的次数是问题规模n的某个函数，用T(n)表示，若有某个辅助函数f(n),使得当n趋近于无穷大时，**T(n)/f(n)**的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函数。记作**T(n)=Ｏ(f(n)),**称**Ｏ(f(n))** 为算法的渐进时间复杂度，简称时间复杂度。

**指数时间**

指的是一个问题求解所需要的[计算时间](https://zh.wikipedia.org/wiki/%E8%A8%88%E7%AE%97%E6%99%82%E9%96%93)m(n)，依输入数据的大小而呈[指数](https://zh.wikipedia.org/wiki/%E6%8C%87%E6%95%B0)成长（即输入数据的数量依[线性](https://zh.wikipedia.org/wiki/%E7%B7%9A%E6%80%A7)成长，所花的时间将会以指数成长）

|  |  |
| --- | --- |
| 1  2  3  4  5 | for (i=1; i<=n; i++)         x++;  for (i=1; i<=n; i++)      　for (j=1; j<=n; j++)            x++; |

第一个for循环的时间复杂度为Ο(n)，第二个for循环的时间复杂度为Ο(n2)，则整个算法的时间复杂度为Ο(n+n2)=Ο(n2)。

**常数时间**

若对于一个算法，的上界与输入大小无关，则称其具有**常数时间**，记作时间。一个例子是访问[数组](https://zh.wikipedia.org/wiki/%E6%95%B0%E7%BB%84)中的单个元素，因为访问它只需要一条[指令](https://zh.wikipedia.org/wiki/%E6%8C%87%E4%BB%A4)。但是，找到无序数组中的最小元素则不是，因为这需要遍历所有元素来找出最小值。这是一项线性时间的操作，或称时间。但如果预先知道元素的数量并假设数量保持不变，则该操作也可被称为具有常数时间。

**对数时间**

若算法的T(n) = O(log n)，则称其具有对数时间

常见的具有对数时间的算法有[二叉树](https://zh.wikipedia.org/wiki/%E4%BA%8C%E5%8F%89%E6%A0%91)的相关操作和[二分搜索](https://zh.wikipedia.org/wiki/%E4%BA%8C%E5%88%86%E6%90%9C%E7%B4%A2)。

对数时间的算法是非常有效的，因为每增加一个输入，其所需要的额外计算时间会变小。

递归地将字符串砍半并且输出是这个类别函数的一个简单例子。它需要O（log n）的时间因为每次输出之前我们都将字符串砍半。 这意味着，如果我们想增加输出的次数，我们需要将字符串长度加倍。

**线性时间**

如果一个算法的时间复杂度为O(n)，则称这个算法具有线性时间，或O(n)时间。非正式地说，这意味着对于足够大的输入，运行时间增加的大小与输入成线性关系。例如，一个计算列表所有元素的和的程序，需要的时间与列表的长度成正比。

### ****正则表达式****

语法:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | import re #导入模块名    p = re.compile("^[0-9]")  #生成要匹配的正则对象 ， ^代表从开头匹配，[0-9]代表匹配0至9的任意一个数字， 所以这里的意思是对传进来的字符串进行匹配，如果这个字符串的开头第一个字符是数字，就代表匹配上了    m = p.match('14534Abc')   #按上面生成的正则对象 去匹配 字符串， 如果能匹配成功，这个m就会有值， 否则m为None<br><br>if m: #不为空代表匹配上了  　　print(m.group())　　　　#m.group()返回匹配上的结果，此处为1，因为匹配上的是1这个字符<br>else:<br>　　print("doesn't match.")<br> |

上面的第2 和第3行也可以合并成一行来写：

|  |  |
| --- | --- |
| 1 | m = p.match("^[0-9]",'14534Abc') |

效果是一样的，区别在于，第一种方式是提前对要匹配的格式进行了编译（对匹配公式进行解析），这样再去匹配的时候就不用在编译匹配的格式，第2种简写是每次匹配的时候 都 要进行一次匹配公式的编译，所以，如果你需要从一个5w行的文件中匹配出所有以数字开头的行，建议先把正则公式进行编译再匹配，这样速度会快点。

匹配格式

|  |  |
| --- | --- |
| **模式** | **描述** |
| ^ | 匹配字符串的开头 |
| $ | 匹配字符串的末尾。 |
| . | 匹配任意字符，除了换行符，当re.DOTALL标记被指定时，则可以匹配包括换行符的任意字符。 |
| [...] | 用来表示一组字符,单独列出：[amk] 匹配 'a'，'m'或'k' |
| [^...] | 不在[]中的字符：[^abc] 匹配除了a,b,c之外的字符。 |
| re\* | 匹配0个或多个的表达式。 |
| re+ | 匹配1个或多个的表达式。 |
| re? | 匹配0个或1个由前面的正则表达式定义的片段，非贪婪方式 |
| re{ n} |  |
| re{ n,} | 精确匹配n个前面表达式。 |
| re{ n, m} | 匹配 n 到 m 次由前面的正则表达式定义的片段，贪婪方式 |
| a| b | 匹配a或b |
| (re) | G匹配括号内的表达式，也表示一个组 |
| (?imx) | 正则表达式包含三种可选标志：i, m, 或 x 。只影响括号中的区域。 |
| (?-imx) | 正则表达式关闭 i, m, 或 x 可选标志。只影响括号中的区域。 |
| (?: re) | 类似 (...), 但是不表示一个组 |
| (?imx: re) | 在括号中使用i, m, 或 x 可选标志 |
| (?-imx: re) | 在括号中不使用i, m, 或 x 可选标志 |
| (?#...) | 注释. |
| (?= re) | 前向肯定界定符。如果所含正则表达式，以 ... 表示，在当前位置成功匹配时成功，否则失败。但一旦所含表达式已经尝试，匹配引擎根本没有提高；模式的剩余部分还要尝试界定符的右边。 |
| (?! re) | 前向否定界定符。与肯定界定符相反；当所含表达式不能在字符串当前位置匹配时成功 |
| (?> re) | 匹配的独立模式，省去回溯。 |
| \w | 匹配字母数字 |
| \W | 匹配非字母数字 |
| \s | 匹配任意空白字符，等价于 [\t\n\r\f]. |
| \S | 匹配任意非空字符 |
| \d | 匹配任意数字，等价于 [0-9]. |
| \D | 匹配任意非数字 |
| \A | 匹配字符串开始 |
| \Z | 匹配字符串结束，如果是存在换行，只匹配到换行前的结束字符串。c |
| \z | 匹配字符串结束 |
| \G | 匹配最后匹配完成的位置。 |
| \b | 匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。 |
| \B | 匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。 |
| \n, \t, 等. | 匹配一个换行符。匹配一个制表符。等 |
| \1...\9 | 匹配第n个分组的子表达式。 |
| \10 | 匹配第n个分组的子表达式，如果它经匹配。否则指的是八进制字符码的表达式。 |

**正则表达式常用5种操作**

re.match(pattern, string)　　　　 # 从头匹配

re.search(pattern, string)　　　　# 匹配整个字符串，直到找到一个匹配

re.split()　　　　　　　　　　　　# 将匹配到的格式当做分割点对字符串分割成列表

|  |  |
| --- | --- |
| 1  2 | >>>m = re.split("[0-9]", "alex1rain2jack3helen rachel8")  >>>print(m) |

输出：　['alex', 'rain', 'jack', 'helen rachel', '']

re.findall()　　　　　　　　　　# 找到所有要匹配的字符并返回列表格式

|  |  |
| --- | --- |
| 1  2 | >>>m = re.findall("[0-9]", "alex1rain2jack3helen rachel8")  >>>print(m)<br> |

输出：['1', '2', '3', '8']

re.sub(pattern, repl, string, count,flag) 　　　# 替换匹配到的字符

|  |  |
| --- | --- |
| 1  2 | m=re.sub("[0-9]","|", "alex1rain2jack3helen rachel8",count=2 )  print(m) |

输出：alex|rain|jack3helen rachel8

#### 正则表达式实例

#### 字符匹配

|  |  |
| --- | --- |
| **实例** | **描述** |
| python | 匹配 "python". |

#### 字符类

|  |  |
| --- | --- |
| **实例** | **描述** |
| [Pp]ython | 匹配 "Python" 或 "python" |
| rub[ye] | 匹配 "ruby" 或 "rube" |
| [aeiou] | 匹配中括号内的任意一个字母 |
| [0-9] | 匹配任何数字。类似于 [0123456789] |
| [a-z] | 匹配任何小写字母 |
| [A-Z] | 匹配任何大写字母 |
| [a-zA-Z0-9] | 匹配任何字母及数字 |
| [^aeiou] | 除了aeiou字母以外的所有字符 |
| [^0-9] | 匹配除了数字外的字符 |

#### 特殊字符类

|  |  |
| --- | --- |
| **实例** | **描述** |
| . | 匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用象 '[.\n]' 的模式。 |
| \d | 匹配一个数字字符。等价于 [0-9]。 |
| \D | 匹配一个非数字字符。等价于 [^0-9]。 |
| \s | 匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [ \f\n\r\t\v]。 |
| \S | 匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。 |
| \w | 匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9\_]'。 |
| \W | 匹配任何非单词字符。等价于 '[^A-Za-z0-9\_]'。 |

## re.match与re.search的区别

re.match只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回None；而re.search匹配整个字符串，直到找到一个匹配。

## Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these −

|  |  |
| --- | --- |
| **Modifier** | **Description** |
| re.I | Performs case-insensitive matching. |
| re.L | Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior (\b and \B). |
| re.M | Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string). |
| re.S | Makes a period (dot) match any character, including a newline. |
| re.U | Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B. |
| re.X | Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker. |

**几个常见正则例子：**

匹配手机号

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | phone\_str = "hey my name is alex, and my phone number is 13651054607, please call me if you are pretty!"  phone\_str2 = "hey my name is alex, and my phone number is 18651054604, please call me if you are pretty!"    m = re.search("(1)([358]\d{9})",phone\_str2)  if m:      print(m.group()) |

匹配IP V4

|  |  |
| --- | --- |
| 1  2  3  4  5 | ip\_addr = "inet 192.168.60.223 netmask 0xffffff00 broadcast 192.168.60.255"    m = re.search("\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}", ip\_addr)    print(m.group()) |

分组匹配地址

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | contactInfo = 'Oldboy School, Beijing Changping Shahe: 010-8343245'  match = re.search(r'(\w+), (\w+): (\S+)', contactInfo) #分组  """  >>> match.group(1)    'Doe'    >>> match.group(2)    'John'    >>> match.group(3)    '555-1212'  """  match = re.search(r'(?P<last>\w+), (?P<first>\w+): (?P<phone>\S+)', contactInfo)  """   >>> match.group('last')    'Doe'    >>> match.group('first')    'John'    >>> match.group('phone')    '555-1212'  """ |

匹配email

|  |  |
| --- | --- |
| 1  2  3  4 | email = "alex.li@126.com   http://www.oldboyedu.com"    m = re.search(r"[0-9.a-z]{0,26}@[0-9.a-z]{0,20}.[0-9a-z]{0,8}", email)  print(m.group()) |

**json 和 pickle**

用于序列化的两个模块

* json，用于字符串 和 python数据类型间进行转换
* pickle，用于python特有的类型 和 python的数据类型间进行转换

Json模块提供了四个功能：dumps、dump、loads、load

pickle模块提供了四个功能：dumps、dump、loads、load



**其它常用模块学习**

**http://www.cnblogs.com/wupeiqi/articles/4963027.html**

# [Python之路,Day4 - Python基础4 (new版)](http://www.cnblogs.com/alex3714/articles/5765046.html)

<http://www.cnblogs.com/alex3714/articles/5765046.html>

本节内容

1. 迭代器&生成器
2. 装饰器
3. Json & pickle 数据序列化
4. 软件目录结构规范
5. 作业:ATM项目开发

### 1.列表生成式，迭代器&生成器

#### 列表生成式

孩子，我现在有个需求，看列表[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],我要求你把列表里的每个值加1，你怎么实现？你可能会想到2种方式

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

>>> a

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> b = []

>>> for i in a:b.append(i+1)

...

>>> b

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> a = b

>>> a

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

a = [1,3,4,6,7,7,8,9,11]  
  
for index,i in enumerate(a):  
 a[index] +=1  
print(a)  
  
原值修改

[复制代码](javascript:void(0);)

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

>>> a

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> a = map(lambda x:x+1, a)

>>> a

<map object at 0x101d2c630>

>>> for i in a:print(i)

...

2

3

4

5

6

7

8

9

10

11

[复制代码](javascript:void(0);)

其实还有一种写法，如下

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

>>> a = [i+1 for i in range(10)]

>>> a

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

这就叫做列表生成

#### 生成器

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器：generator。

要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的[]改成()，就创建了一个generator：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | >>> L = [x \* x for x in range(10)]  >>> L  [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]  >>> g = (x \* x for x in range(10))  >>> g  <generator object <genexpr> at 0x1022ef630> |

创建L和g的区别仅在于最外层的[]和()，L是一个list，而g是一个generator。

我们可以直接打印出list的每一个元素，但我们怎么打印出generator的每一个元素呢？

如果要一个一个打印出来，可以通过next()函数获得generator的下一个返回值：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | >>> next(g)  0  >>> next(g)  1  >>> next(g)  4  >>> next(g)  9  >>> next(g)  16  >>> next(g)  25  >>> next(g)  36  >>> next(g)  49  >>> next(g)  64  >>> next(g)  81  >>> next(g)  Traceback (most recent call last):    File "<stdin>", line 1, in <module>  StopIteration |

我们讲过，generator保存的是算法，每次调用next(g)，就计算出g的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误。

当然，上面这种不断调用next(g)实在是太变态了，正确的方法是使用for循环，因为generator也是可迭代对象：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | >>> g = (x \* x for x in range(10))  >>> for n in g:  ...     print(n)  ...  0  1  4  9  16  25  36  49  64  81 |

所以，我们创建了一个generator后，基本上永远不会调用next()，而是通过for循环来迭代它，并且不需要关心StopIteration的错误。

generator非常强大。如果推算的算法比较复杂，用类似列表生成式的for循环无法实现的时候，还可以用函数来实现。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def fib(max):      n, a, b = 0, 0, 1      while n < max:          print(b)          a, b = b, a + b          n = n + 1      return 'done' |

注意，赋值语句：

|  |  |
| --- | --- |
| 1 | a, b = b, a + b |

相当于：

|  |  |
| --- | --- |
| 1  2  3 | t = (b, a + b) # t是一个tuple  a = t[0]  b = t[1] |

但不必显式写出临时变量t就可以赋值。

上面的函数可以输出斐波那契数列的前N个数：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | >>> fib(10)  1  1  2  3  5  8  13  21  34  55  done |

仔细观察，可以看出，fib函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。

也就是说，上面的函数和generator仅一步之遥。要把fib函数变成generator，只需要把print(b)改为yield b就可以了：

[复制代码](javascript:void(0);)

def fib(max):

n,a,b = 0,0,1

while n < max:

#print(b)

yield b

a,b = b,a+b

n += 1

return 'done'

[复制代码](javascript:void(0);)

这就是定义generator的另一种方法。如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator：

>>> f = fib(6)

>>> f

<generator object fib at 0x104feaaa0>

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

[复制代码](javascript:void(0);)

data = fib(10)

print(data)

print(data.\_\_next\_\_())

print(data.\_\_next\_\_())

print("干点别的事")

print(data.\_\_next\_\_())

print(data.\_\_next\_\_())

print(data.\_\_next\_\_())

print(data.\_\_next\_\_())

print(data.\_\_next\_\_())

#输出

<generator object fib at 0x101be02b0>

1

1

干点别的事

2

3

5

8

13

[复制代码](javascript:void(0);)

在上面fib的例子，我们在循环过程中不断调用yield，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator后，我们基本上从来不会用next()来获取下一个返回值，而是直接使用for循环来迭代：

[复制代码](javascript:void(0);)

>>> for n in fib(6):

... print(n)

...

1

1

2

3

5

8

[复制代码](javascript:void(0);)

但是用for循环调用generator时，发现拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | >>> g = fib(6)  >>> while True:  ...     try:  ...         x = next(g)  ...         print('g:', x)  ...     except StopIteration as e:  ...         print('Generator return value:', e.value)  ...         break  ...  g: 1  g: 1  g: 2  g: 3  g: 5  g: 8  Generator return value: done |

关于如何捕获错误，后面的错误处理还会详细讲解。

还可通过yield实现在单线程的情况下实现并发运算的效果

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

#\_\*\_coding:utf-8\_\*\_

\_\_author\_\_ = 'Alex Li'

import time

def consumer(name):

print("%s 准备吃包子啦!" %name)

while True:

baozi = yield

print("包子[%s]来了,被[%s]吃了!" %(baozi,name))

def producer(name):

c = consumer('A')

c2 = consumer('B')

c.\_\_next\_\_()

c2.\_\_next\_\_()

print("老子开始准备做包子啦!")

for i in range(10):

time.sleep(1)

print("做了2个包子!")

c.send(i)

c2.send(i)

producer("alex")

[复制代码](javascript:void(0);)

#### 迭代器

我们已经知道，可以直接作用于for循环的数据类型有以下几种：

一类是集合数据类型，如list、tuple、dict、set、str等；

一类是generator，包括生成器和带yield的generator function。

这些可以直接作用于for循环的对象统称为可迭代对象：Iterable。

可以使用isinstance()判断一个对象是否是Iterable对象：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | >>> from collections import Iterable  >>> isinstance([], Iterable)  True  >>> isinstance({}, Iterable)  True  >>> isinstance('abc', Iterable)  True  >>> isinstance((x for x in range(10)), Iterable)  True  >>> isinstance(100, Iterable)  False |

而生成器不但可以作用于for循环，还可以被next()函数不断调用并返回下一个值，直到最后抛出StopIteration错误表示无法继续返回下一个值了。

**\*可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator**。

可以使用isinstance()判断一个对象是否是Iterator对象：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | >>> from collections import Iterator  >>> isinstance((x for x in range(10)), Iterator)  True  >>> isinstance([], Iterator)  False  >>> isinstance({}, Iterator)  False  >>> isinstance('abc', Iterator)  False |

生成器都是Iterator对象，但list、dict、str虽然是Iterable，却不是Iterator。

把list、dict、str等Iterable变成Iterator可以使用iter()函数：

|  |  |
| --- | --- |
| 1  2  3  4 | >>> isinstance(iter([]), Iterator)  True  >>> isinstance(iter('abc'), Iterator)  True |

你可能会问，为什么list、dict、str等数据类型不是Iterator？

这是因为Python的Iterator对象表示的是一个数据流，Iterator对象可以被next()函数调用并不断返回下一个数据，直到没有数据时抛出StopIteration错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过next()函数实现按需计算下一个数据，所以Iterator的计算是惰性的，只有在需要返回下一个数据时它才会计算。

Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。

**小结**

凡是可作用于for循环的对象都是Iterable类型；

凡是可作用于next()函数的对象都是Iterator类型，它们表示一个惰性计算的序列；

集合数据类型如list、dict、str等是Iterable但不是Iterator，不过可以通过iter()函数获得一个Iterator对象。

Python的for循环本质上就是通过不断调用next()函数实现的，例如：

|  |  |
| --- | --- |
| 1  2 | for x in [1, 2, 3, 4, 5]:      pass |

实际上完全等价于：

[复制代码](javascript:void(0);)

# 首先获得Iterator对象:

it = iter([1, 2, 3, 4, 5])

# 循环:

while True:

try:

# 获得下一个值:

x = next(it)

except StopIteration:

# 遇到StopIteration就退出循环

break

[复制代码](javascript:void(0);)

### ****2.装饰器****

你是一家视频网站的后端开发工程师，你们网站有以下几个版块

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | def home():      print("---首页----")    def america():      print("----欧美专区----")    def japan():      print("----日韩专区----")    def henan():      print("----河南专区----") |

视频刚上线初期，为了吸引用户，你们采取了免费政策，所有视频免费观看，迅速吸引了一大批用户，免费一段时间后，每天巨大的带宽费用公司承受不了了，所以准备对比较受欢迎的几个版块收费，其中包括“欧美” 和 “河南”专区，你拿到这个需求后，想了想，想收费得先让其进行用户认证，认证通过后，再判定这个用户是否是VIP付费会员就可以了，是VIP就让看，不是VIP就不让看就行了呗。 你觉得这个需求很是简单，因为要对多个版块进行认证，那应该把认证功能提取出来单独写个模块，然后每个版块里调用 就可以了，与是你轻轻的就实现了下面的功能 。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | #\_\*\_coding:utf-8\_\*\_      user\_status = False #用户登录了就把这个改成True    def login():      \_username = "alex" #假装这是DB里存的用户信息      \_password = "abc!23" #假装这是DB里存的用户信息      global user\_status        if user\_status == False:          username = input("user:")          password = input("pasword:")            if username == \_username and password == \_password:              print("welcome login....")              user\_status = True          else:              print("wrong username or password!")      else:          print("用户已登录，验证通过...")    def home():      print("---首页----")    def america():      login() #执行前加上验证      print("----欧美专区----")    def japan():      print("----日韩专区----")    def henan():      login() #执行前加上验证      print("----河南专区----")        home()  america()  henan() |

此时你信心满满的把这个代码提交给你的TEAM LEADER审核，没成想，没过5分钟，代码就被打回来了， TEAM LEADER给你反馈是，我现在有很多模块需要加认证模块，你的代码虽然实现了功能，但是需要更改需要加认证的各个模块的代码，这直接违反了软件开发中的一个原则“开放-封闭”原则，简单来说，它规定已经实现的功能代码不允许被修改，但可以被扩展，即：

* 封闭：已实现的功能代码块不应该被修改
* 开放：对现有功能的扩展开放

这个原则你还是第一次听说，我擦，再次感受了自己这个野生程序员与正规军的差距，BUT ANYWAY,老大要求的这个怎么实现呢？如何在不改原有功能代码的情况下加上认证功能呢？你一时想不出思路，只好带着这个问题回家继续憋，媳妇不在家，去隔壁老王家串门了，你正好落的清静，一不小心就想到了解决方案，不改源代码可以呀，

你师从沙河金角大王时，记得他教过你，高阶函数，就是把一个函数当做一个参数传给另外一个函数，当时大王说，有一天，你会用到它的，没想到这时这个知识点突然从脑子 里蹦出来了，我只需要写个认证方法，每次调用 需要验证的功能 时，直接 把这个功能 的函数名当做一个参数 传给 我的验证模块不就行了么，哈哈，机智如我，如是你啪啪啪改写了之前的代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | #\_\*\_coding:utf-8\_\*\_      user\_status = False #用户登录了就把这个改成True    def login(func): #把要执行的模块从这里传进来      \_username = "alex" #假装这是DB里存的用户信息      \_password = "abc!23" #假装这是DB里存的用户信息      global user\_status        if user\_status == False:          username = input("user:")          password = input("pasword:")            if username == \_username and password == \_password:              print("welcome login....")              user\_status = True          else:              print("wrong username or password!")        if user\_status == True:          func() # 看这里看这里，只要验证通过了，就调用相应功能    def home():      print("---首页----")    def america():      #login() #执行前加上验证      print("----欧美专区----")    def japan():      print("----日韩专区----")    def henan():      #login() #执行前加上验证      print("----河南专区----")        home()  login(america) #需要验证就调用 login，把需要验证的功能 当做一个参数传给login  # home()  # america()  login(henan) |

你很开心，终于实现了老板的要求，不改变原功能代码的前提下，给功能加上了验证，此时，媳妇回来了，后面还跟着老王，你两家关系 非常 好，老王经常来串门，老王也是码农，你跟他分享了你写的代码，兴奋的等他看完 夸奖你NB,没成想，老王看后，并没有夸你，抱起你的儿子，笑笑说，你这个代码还是改改吧， 要不然会被开除的，WHAT? 会开除，明明实现了功能 呀， 老王讲，没错，你功能 是实现了，但是你又犯了一个大忌，什么大忌？

你改变了调用方式呀， 想一想，现在没每个需要认证的模块，都必须调用你的login()方法，并把自己的函数名传给你，人家之前可不是这么调用 的， 试想，如果 有100个模块需要认证，那这100个模块都得更改调用方式，这么多模块肯定不止是一个人写的，让每个人再去修改调用方式 才能加上认证，你会被骂死的。。。。

你觉得老王说的对，但问题是，如何即不改变原功能代码，又不改变原有调用方式，还能加上认证呢？ 你苦思了一会，还是想不出，老王在逗你的儿子玩，你说，老王呀，快给我点思路 ，实在想不出来，老王背对着你问，

老王：学过匿名函数没有？

你：学过学过，就是lambda嘛

老王：那lambda与正常函数的区别是什么？

你：最直接的区别是，正常函数定义时需要写名字，但lambda不需要

老王：没错，那lambda定好后，为了多次调用 ，可否也给它命个名？

你：可以呀，可以写成plus = lambda x:x+1类似这样，以后再调用plus就可以了，但这样不就失去了lambda的意义了，明明人家叫匿名函数呀，你起了名字有什么用呢？

老王：我不是要跟你讨论它的意义 ，我想通过这个让你明白一个事实

说着，老王拿起你儿子的画板，在上面写了以下代码：

|  |  |
| --- | --- |
| 1  2  3  4 | def plus(n):      return n+1    plus2 = lambda x:x+1 |

老王： 上面这两种写法是不是代表 同样的意思？

你：是的

老王：我给lambda x:x+1 起了个名字叫plus2，是不是相当于def plus2(x) ?

你：我擦，你别说，还真是，但老王呀，你想说明什么呢？

老王： 没啥，只想告诉你，给函数赋值变量名就像def func\_name　是一样的效果，如下面的plus(n)函数，你调用时可以用plus名，还可以再起个其它名字，如

|  |  |
| --- | --- |
| 1  2  3 | calc = plus    calc(n) |

你明白我想传达什么意思了么？

你：。。。。。。。。。。。这。。。。。。嗯 。。。。。不太。。。。明白 。。

老王：。。。。这。。。。。呵呵。。。。。。好吧。。。。，那我在给你点一下，你之前写的下面这段调用 认证的代码

|  |  |
| --- | --- |
| 1  2  3  4  5 | home()  login(america) #需要验证就调用 login，把需要验证的功能 当做一个参数传给login  # home()  # america()  login(henan) |

你之所改变了调用方式，是因为用户每次调用时需要执行login(henan)，类似的。其实稍一改就可以了呀

|  |  |
| --- | --- |
| 1  2  3 | home()  america = login(america)  henan = login(henan) |

这样你，其它人调用henan时，其实相当于调用了login(henan), 通过login里的验证后，就会自动调用henan功能。

你：我擦，还真是唉。。。，老王，还是你nb。。。不过，等等， 我这样写了好，那用户调用时，应该是下面这个样子

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | home()  america = login(america) #你在这里相当于把america这个函数替换了  henan = login(henan)    #那用户调用时依然写  america() |

但问题在于，还不等用户调用 ，你的america = login(america)就会先自己把america执行了呀。。。。，你应该等我用户调用 的时候 再执行才对呀，不信我试给你看。。。

老王：哈哈，你说的没错，这样搞会出现这个问题？ 但你想想有没有解决办法 呢？

你：我擦，你指的思路呀，大哥。。。我哪知道 下一步怎么走。。。

老王：算了，估计你也想不出来。。。 学过嵌套函数没有？

你：yes,然后呢？

老王：想实现一开始你写的america = login(america)不触发你函数的执行，只需要在这个login里面再定义一层函数，第一次调用america = login(america)只调用到外层login，这个login虽然会执行，但不会触发认证了，因为认证的所有代码被封装在login里层的新定义 的函数里了，login只返回 里层函数的函数名，这样下次再执行america()时， 就会调用里层函数啦。。。

你：。。。。。。什么？ 什么个意思，我蒙逼了。。。

老王：还是给你看代码吧。。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | def login(func): #把要执行的模块从这里传进来        def inner():#再定义一层函数          \_username = "alex" #假装这是DB里存的用户信息          \_password = "abc!23" #假装这是DB里存的用户信息          global user\_status            if user\_status == False:              username = input("user:")              password = input("pasword:")                if username == \_username and password == \_password:                  print("welcome login....")                  user\_status = True              else:                  print("wrong username or password!")            if user\_status == True:              func() # 看这里看这里，只要验证通过了，就调用相应功能        return inner #用户调用login时，只会返回inner的内存地址，下次再调用时加上()才会执行inner函数 |

此时你仔细着了老王写的代码　，感觉老王真不是一般人呀，连这种奇淫巧技都能想出来。。。，心中默默感谢上天赐你一个大牛邻居。

你: 老王呀，你这个姿势很nb呀，你独创的？

此时你媳妇噗嗤的笑出声来，你也不知道 她笑个球。。。

老王：呵呵， 这不是我独创的呀当然 ，这是开发中一个常用的玩法，叫语法糖，官方名称“装饰器”，其实上面的写法，还可以更简单

可以把下面代码去掉

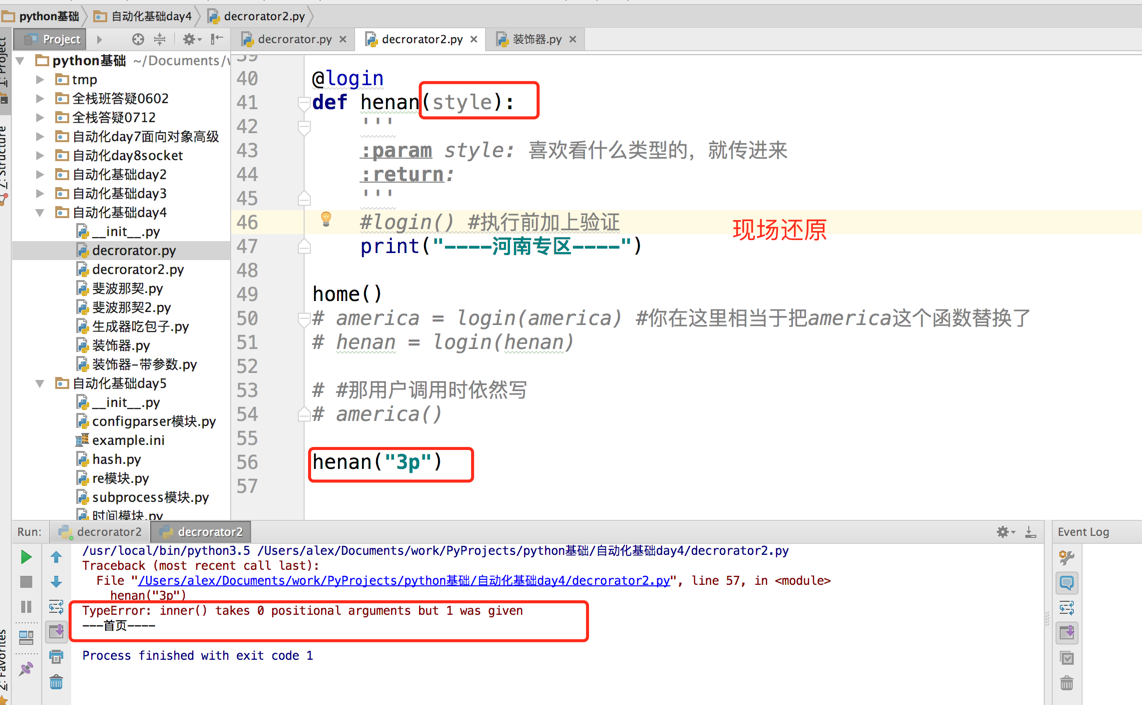
|  |  |
| --- | --- |
| 1 | america = login(america) #你在这里相当于把america这个函数替换了 |

只在你要装饰的函数上面加上下面代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @login  def america():      #login() #执行前加上验证      print("----欧美专区----")    def japan():      print("----日韩专区----")    @login  def henan():      #login() #执行前加上验证      print("----河南专区----") |

效果是一样的。

你开心的玩着老王教你的新姿势 ，玩着玩着就手贱给你的“河南专区”版块 加了个参数，然后，结果 出错了。。。



你：老王，老王，怎么传个参数就不行了呢？

老王：那必然呀，你调用henan时，其实是相当于调用的login，你的henan第一次调用时henan = login(henan)， login就返回了inner的内存地址，第2次用户自己调用henan("3p"),实际上相当于调用的时inner,但你的inner定义时并没有设置参数，但你给他传了个参数，所以自然就报错了呀

你：但是我的 版块需要传参数呀，你不让我传不行呀。。。

老王：没说不让你传，稍做改动便可。。



老王：你再试试就好了 。

你： 果然好使，大神就是大神呀。 。。 不过，如果有多个参数呢？

老王：。。。。老弟，你不要什么都让我教你吧，非固定参数你没学过么？ \*args,\*\*kwargs...

你：噢 。。。还能这么搞?,nb,我再试试。

你身陷这种新玩法中无法自拔，竟没注意到老王已经离开，你媳妇告诉你说为了不打扰你加班，今晚带孩子去跟她姐妹住 ，你觉得媳妇真体贴，最终，你终于搞定了所有需求，完全遵循开放-封闭原则，最终代码如下 。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | #\_\*\_coding:utf-8\_\*\_      user\_status = False #用户登录了就把这个改成True    def login(func): #把要执行的模块从这里传进来        def inner(\*args,\*\*kwargs):#再定义一层函数          \_username = "alex" #假装这是DB里存的用户信息          \_password = "abc!23" #假装这是DB里存的用户信息          global user\_status            if user\_status == False:              username = input("user:")              password = input("pasword:")                if username == \_username and password == \_password:                  print("welcome login....")                  user\_status = True              else:                  print("wrong username or password!")            if user\_status == True:              func(\*args,\*\*kwargs) # 看这里看这里，只要验证通过了，就调用相应功能        return inner #用户调用login时，只会返回inner的内存地址，下次再调用时加上()才会执行inner函数      def home():      print("---首页----")    @login  def america():      #login() #执行前加上验证      print("----欧美专区----")    def japan():      print("----日韩专区----")    # @login  def henan(style):      '''      :param style: 喜欢看什么类型的，就传进来      :return:      '''      #login() #执行前加上验证      print("----河南专区----")    home()  # america = login(america) #你在这里相当于把america这个函数替换了  henan = login(henan)    # #那用户调用时依然写  america()    henan("3p") |

此时，你已累的不行了，洗洗就抓紧睡了，半夜，上厕所，隐隐听到隔壁老王家有微弱的女人的声音传来，你会心一笑，老王这家伙，不声不响找了女朋友也不带给我看看，改天一定要见下真人。。。。

 第二2天早上，产品经理又提了新的需求，要允许用户选择用qq\weibo\weixin认证，此时的你，已深谙装饰器各种装逼技巧，轻松的就实现了新的需求。

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

#\_\*\_coding:utf-8\_\*\_

user\_status = False #用户登录了就把这个改成True

def login(auth\_type): #把要执行的模块从这里传进来

def auth(func):

def inner(\*args,\*\*kwargs):#再定义一层函数

if auth\_type == "qq":

\_username = "alex" #假装这是DB里存的用户信息

\_password = "abc!23" #假装这是DB里存的用户信息

global user\_status

if user\_status == False:

username = input("user:")

password = input("pasword:")

if username == \_username and password == \_password:

print("welcome login....")

user\_status = True

else:

print("wrong username or password!")

if user\_status == True:

return func(\*args,\*\*kwargs) # 看这里看这里，只要验证通过了，就调用相应功能

else:

print("only support qq ")

return inner #用户调用login时，只会返回inner的内存地址，下次再调用时加上()才会执行inner函数

return auth

def home():

print("---首页----")

@login('qq')

def america():

#login() #执行前加上验证

print("----欧美专区----")

def japan():

print("----日韩专区----")

@login('weibo')

def henan(style):

'''

:param style: 喜欢看什么类型的，就传进来

:return:

'''

#login() #执行前加上验证

print("----河南专区----")

home()

# america = login(america) #你在这里相当于把america这个函数替换了

#henan = login(henan)

# #那用户调用时依然写

america()

# henan("3p")

[复制代码](javascript:void(0);)

### 3.Json & pickle 数据序列化

参考 http://www.cnblogs.com/alex3714/articles/5161349.html

### 4.软件目录结构规范

#### 为什么要设计好目录结构?

"设计项目目录结构"，就和"代码编码风格"一样，属于个人风格问题。对于这种风格上的规范，一直都存在两种态度:

1. 一类同学认为，这种个人风格问题"无关紧要"。理由是能让程序work就好，风格问题根本不是问题。
2. 另一类同学认为，规范化能更好的控制程序结构，让程序具有更高的可读性。

我是比较偏向于后者的，因为我是前一类同学思想行为下的直接受害者。我曾经维护过一个非常不好读的项目，其实现的逻辑并不复杂，但是却耗费了我非常长的时间去理解它想表达的意思。从此我个人对于提高项目可读性、可维护性的要求就很高了。"项目目录结构"其实也是属于"可读性和可维护性"的范畴，我们设计一个层次清晰的目录结构，就是为了达到以下两点:

1. 可读性高: 不熟悉这个项目的代码的人，一眼就能看懂目录结构，知道程序启动脚本是哪个，测试目录在哪儿，配置文件在哪儿等等。从而非常快速的了解这个项目。
2. 可维护性高: 定义好组织规则后，维护者就能很明确地知道，新增的哪个文件和代码应该放在什么目录之下。这个好处是，随着时间的推移，代码/配置的规模增加，项目结构不会混乱，仍然能够组织良好。

所以，我认为，保持一个层次清晰的目录结构是有必要的。更何况组织一个良好的工程目录，其实是一件很简单的事儿。

#### 目录组织方式

关于如何组织一个较好的Python工程目录结构，已经有一些得到了共识的目录结构。在Stackoverflow的[这个问题](http://stackoverflow.com/questions/193161/what-is-the-best-project-structure-for-a-python-application)上，能看到大家对Python目录结构的讨论。

这里面说的已经很好了，我也不打算重新造轮子列举各种不同的方式，这里面我说一下我的理解和体会。

假设你的项目名为foo, 我比较建议的最方便快捷目录结构这样就足够了:

Foo/

|-- bin/

| |-- foo

|

|-- foo/

| |-- tests/

| | |-- \_\_init\_\_.py

| | |-- test\_main.py

| |

| |-- \_\_init\_\_.py

| |-- main.py

|

|-- docs/

| |-- conf.py

| |-- abc.rst

|

|-- setup.py

|-- requirements.txt

|-- README

简要解释一下:

1. bin/: 存放项目的一些可执行文件，当然你可以起名script/之类的也行。
2. foo/: 存放项目的所有源代码。(1) 源代码中的所有模块、包都应该放在此目录。不要置于顶层目录。(2) 其子目录tests/存放单元测试代码； (3) 程序的入口最好命名为main.py。
3. docs/: 存放一些文档。
4. setup.py: 安装、部署、打包的脚本。
5. requirements.txt: 存放软件依赖的外部Python包列表。
6. README: 项目说明文件。

除此之外，有一些方案给出了更加多的内容。比如LICENSE.txt,ChangeLog.txt文件等，我没有列在这里，因为这些东西主要是项目开源的时候需要用到。如果你想写一个开源软件，目录该如何组织，可以参考[这篇文章](http://www.jeffknupp.com/blog/2013/08/16/open-sourcing-a-python-project-the-right-way/)。

下面，再简单讲一下我对这些目录的理解和个人要求吧。

#### 关于README的内容

**这个我觉得是每个项目都应该有的一个文件**，目的是能简要描述该项目的信息，让读者快速了解这个项目。

它需要说明以下几个事项:

1. 软件定位，软件的基本功能。
2. 运行代码的方法: 安装环境、启动命令等。
3. 简要的使用说明。
4. 代码目录结构说明，更详细点可以说明软件的基本原理。
5. 常见问题说明。

我觉得有以上几点是比较好的一个README。在软件开发初期，由于开发过程中以上内容可能不明确或者发生变化，并不是一定要在一开始就将所有信息都补全。但是在项目完结的时候，是需要撰写这样的一个文档的。

可以参考Redis源码中[Readme](https://github.com/antirez/redis#what-is-redis)的写法，这里面简洁但是清晰的描述了Redis功能和源码结构。

#### 关于requirements.txt和setup.py

#### setup.py

一般来说，用setup.py来管理代码的打包、安装、部署问题。业界标准的写法是用Python流行的打包工具[setuptools](https://pythonhosted.org/setuptools/setuptools.html#developer-s-guide)来管理这些事情。这种方式普遍应用于开源项目中。不过这里的核心思想不是用标准化的工具来解决这些问题，而是说，**一个项目一定要有一个安装部署工具**，能快速便捷的在一台新机器上将环境装好、代码部署好和将程序运行起来。

这个我是踩过坑的。

我刚开始接触Python写项目的时候，安装环境、部署代码、运行程序这个过程全是手动完成，遇到过以下问题:

1. 安装环境时经常忘了最近又添加了一个新的Python包，结果一到线上运行，程序就出错了。
2. Python包的版本依赖问题，有时候我们程序中使用的是一个版本的Python包，但是官方的已经是最新的包了，通过手动安装就可能装错了。
3. 如果依赖的包很多的话，一个一个安装这些依赖是很费时的事情。
4. 新同学开始写项目的时候，将程序跑起来非常麻烦，因为可能经常忘了要怎么安装各种依赖。

setup.py可以将这些事情自动化起来，提高效率、减少出错的概率。"复杂的东西自动化，能自动化的东西一定要自动化。"是一个非常好的习惯。

setuptools的[文档](https://pythonhosted.org/setuptools/setuptools.html#developer-s-guide)比较庞大，刚接触的话，可能不太好找到切入点。学习技术的方式就是看他人是怎么用的，可以参考一下Python的一个Web框架，flask是如何写的: [setup.py](https://github.com/mitsuhiko/flask/blob/master/setup.py)

当然，简单点自己写个安装脚本（deploy.sh）替代setup.py也未尝不可。

#### requirements.txt

这个文件存在的目的是:

1. 方便开发者维护软件的包依赖。将开发过程中新增的包添加进这个列表中，避免在setup.py安装依赖时漏掉软件包。
2. 方便读者明确项目使用了哪些Python包。

这个文件的格式是每一行包含一个包依赖的说明，通常是flask>=0.10这种格式，要求是这个格式能被pip识别，这样就可以简单的通过 pip install -r requirements.txt来把所有Python包依赖都装好了。具体格式说明： [点这里](https://pip.readthedocs.org/en/1.1/requirements.html)。

#### 关于配置文件的使用方法

#### 注意，在上面的目录结构中，没有将conf.py放在源码目录下，而是放在docs/目录下。

很多项目对配置文件的使用做法是:

1. 配置文件写在一个或多个python文件中，比如此处的conf.py。
2. 项目中哪个模块用到这个配置文件就直接通过import conf这种形式来在代码中使用配置。

这种做法我不太赞同:

1. 这让单元测试变得困难（因为模块内部依赖了外部配置）
2. 另一方面配置文件作为用户控制程序的接口，应当可以由用户自由指定该文件的路径。
3. 程序组件可复用性太差，因为这种贯穿所有模块的代码硬编码方式，使得大部分模块都依赖conf.py这个文件。

所以，我认为配置的使用，更好的方式是，

1. 模块的配置都是可以灵活配置的，不受外部配置文件的影响。
2. 程序的配置也是可以灵活控制的。

能够佐证这个思想的是，用过nginx和mysql的同学都知道，nginx、mysql这些程序都可以自由的指定用户配置。

所以，不应当在代码中直接import conf来使用配置文件。上面目录结构中的conf.py，是给出的一个配置样例，不是在写死在程序中直接引用的配置文件。可以通过给main.py启动参数指定配置路径的方式来让程序读取配置内容。当然，这里的conf.py你可以换个类似的名字，比如settings.py。或者你也可以使用其他格式的内容来编写配置文件，比如settings.yaml之类的。

### 5.本节作业

作业需求：

模拟实现一个ATM + 购物商城程序

1. 额度 15000或自定义
2. 实现购物商城，买东西加入 购物车，调用信用卡接口结账
3. 可以提现，手续费5%
4. ~~每月22号出账单，每月10号为还款日，过期未还，按欠款总额 万分之5 每日计息~~
5. 支持多账户登录
6. 支持账户间转账
7. 记录每月日常消费流水
8. 提供还款接口
9. ATM记录操作日志
10. 提供管理接口，包括添加账户、用户额度，冻结账户等。。。
11. 用户认证用装饰器

示例代码 https://github.com/triaquae/py3\_training/tree/master/atm

简易流程图：https://www.processon.com/view/link/589eb841e4b0999184934329