Contents

[1、 How to run automation 2](#_Toc454802818)

[A、Stress test 2](#_Toc454802819)

[B、Unit test 3](#_Toc454802820)

[2、Process of automation 5](#_Toc454802821)

[**A、大鹏工作流程** 5](#_Toc454802822)

[B、interact.py是沟通大鹏与automation的桥梁 6](#_Toc454802823)

[**C、以pytest为框架的automation工作流程** 7](#_Toc454802824)

[3、命令方式使用jlink 8](#_Toc454802825)

[4、GNU工具的使用 10](#_Toc454802826)

[5、Python语言相关 12](#_Toc454802827)

[Code Checked at Runtime 13](#_Toc454802828)

[Online help, help(), and dir() 13](#_Toc454802829)

[序列 17](#_Toc454802830)

[For 循环 19](#_Toc454802831)

[词典 23](#_Toc454802832)

[引入模块 25](#_Toc454802833)

[路径搜索 26](#_Toc454802834)

[文本输入输出 27](#_Toc454802835)

[动态类型 28](#_Toc454802836)

[文件操作 29](#_Toc454802837)

[Pipe操作 30](#_Toc454802838)

[Pytest 31](#_Toc454802839)

[Fixture 34](#_Toc454802840)

[OS 37](#_Toc454802841)

[Chdir() 37](#_Toc454802842)

[6、Python常用代码 38](#_Toc454802843)

[1、文件操作 38](#_Toc454802844)

[获取当前文件路径 38](#_Toc454802845)

[判断是否为文件夹\文件 39](#_Toc454802846)

[列出文件夹里面的内容 39](#_Toc454802847)

[文件读写 39](#_Toc454802848)

[文件查找拷贝 40](#_Toc454802849)

[处理二进制数据 40](#_Toc454802850)

[实例：读取PC and stack 40](#_Toc454802851)

# 1、How to run automation

此处所讲解的是单机版（没有使用大鹏）的automation。简单来讲大鹏就是调用我们automation。只不过一些配置，比如com口，jlink号等等，通过大鹏提供。

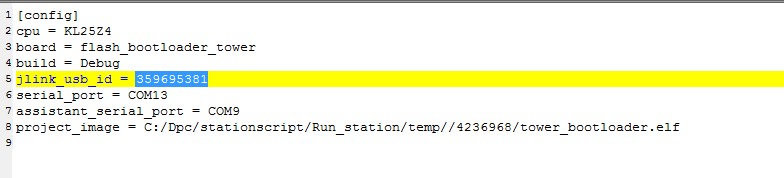
## A、Stress test

1、执行测试前的配置工作：

参数配置目录：

E:\Git\_Repos\app\_data\2.0\kibble\mcu\_kibble\_val\targets

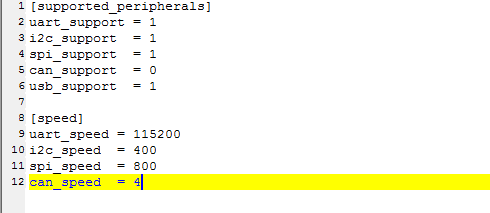
test\_config.ini里面配置参数，cpu跟target目录名称一样，jlink ID使用usb 2抓取jlink的ID，配置板子跟buspal的com口



打开target板子目录，例如：

E:\Git\_Repos\app\_data\2.0\kibble\mcu\_kibble\_val\targets\KL25Z4

peripheral\_config.ini里面配置需要测试的项目



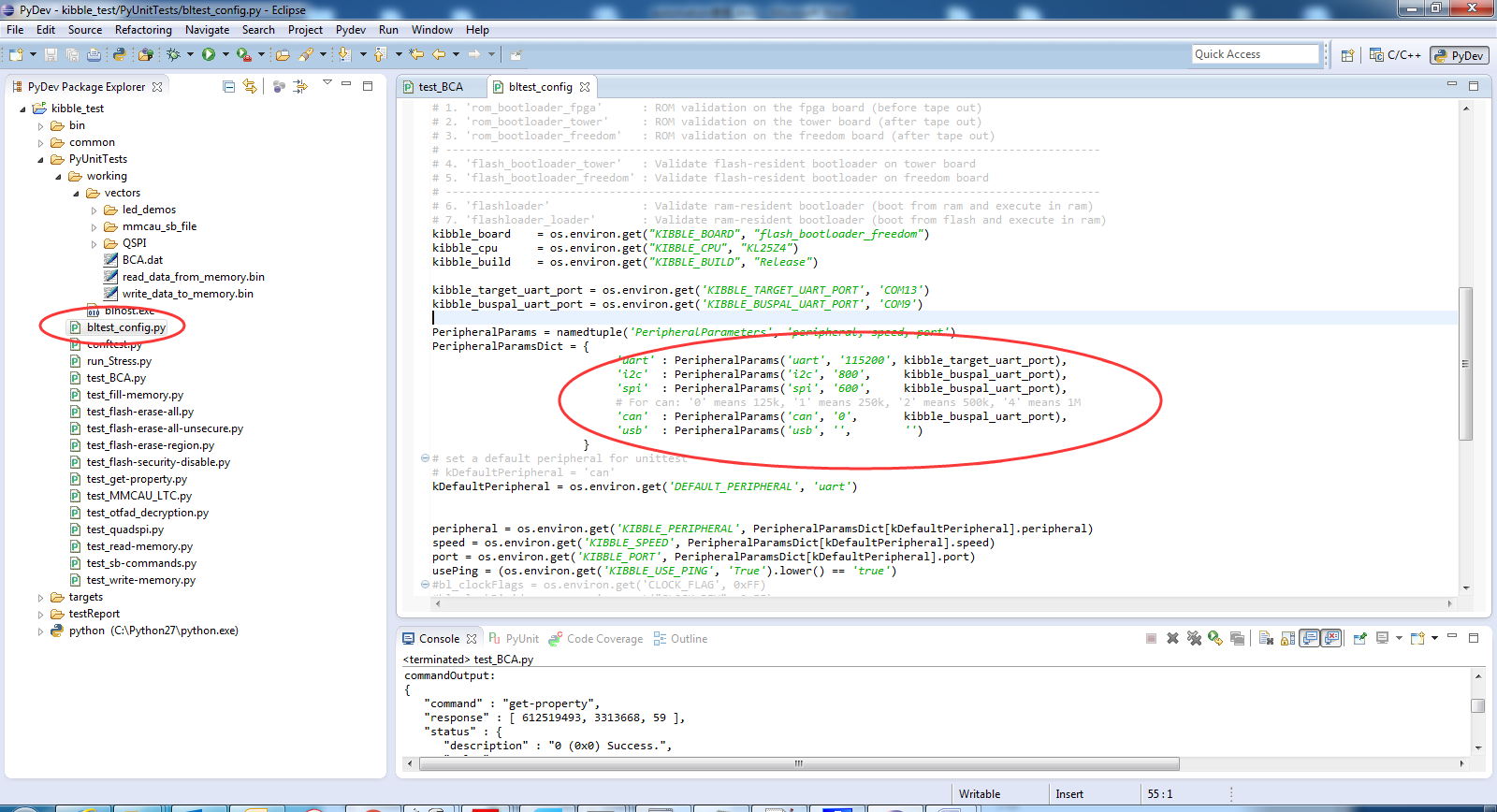
2、执行文档目录：

E:\Git\_Repos\app\_data\2.0\kibble\common

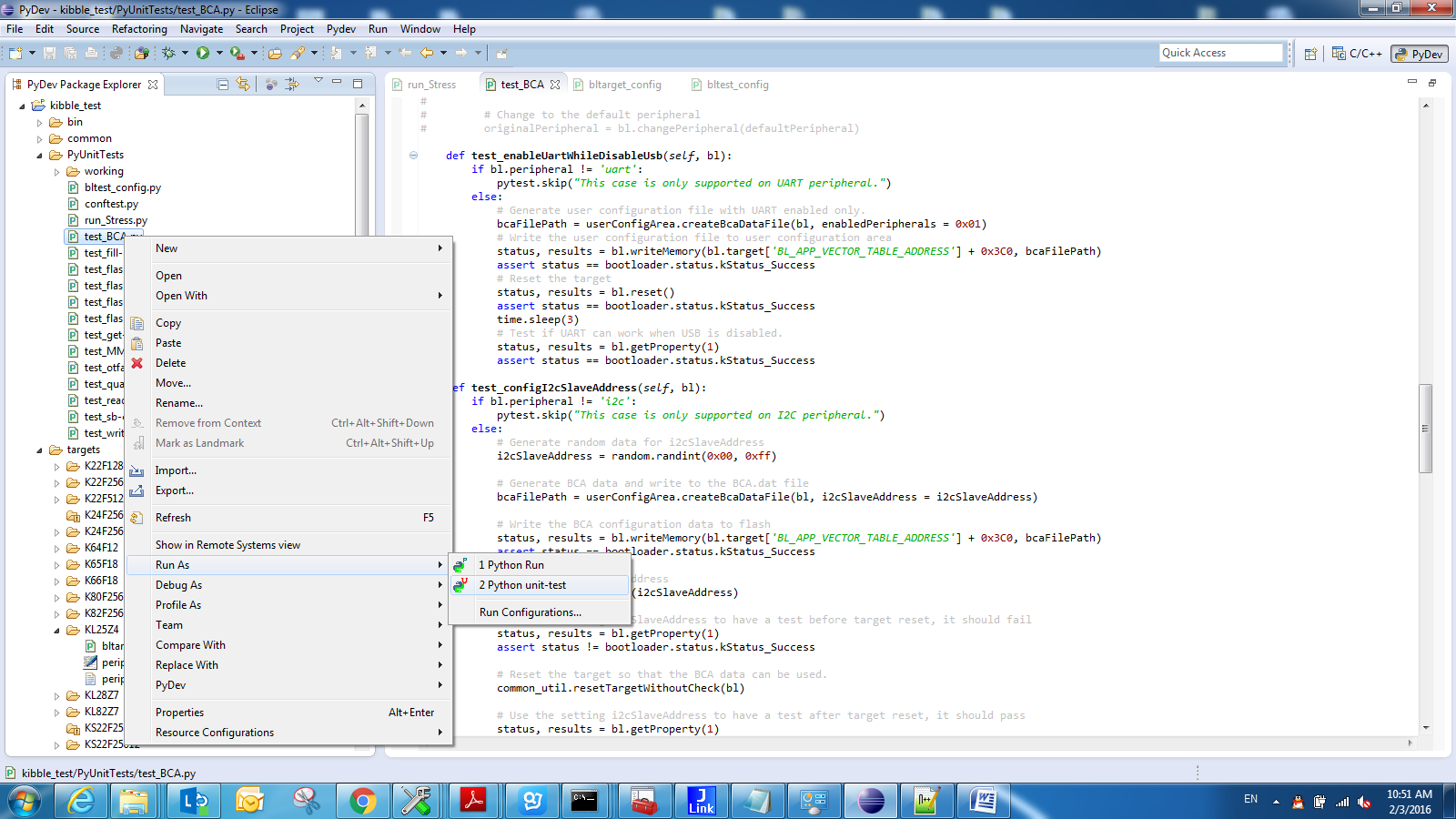
当前目录shift+ctrl+右击 ，点击open command window here，然后输入python interact.py。执行测试 。

## B、Unit test

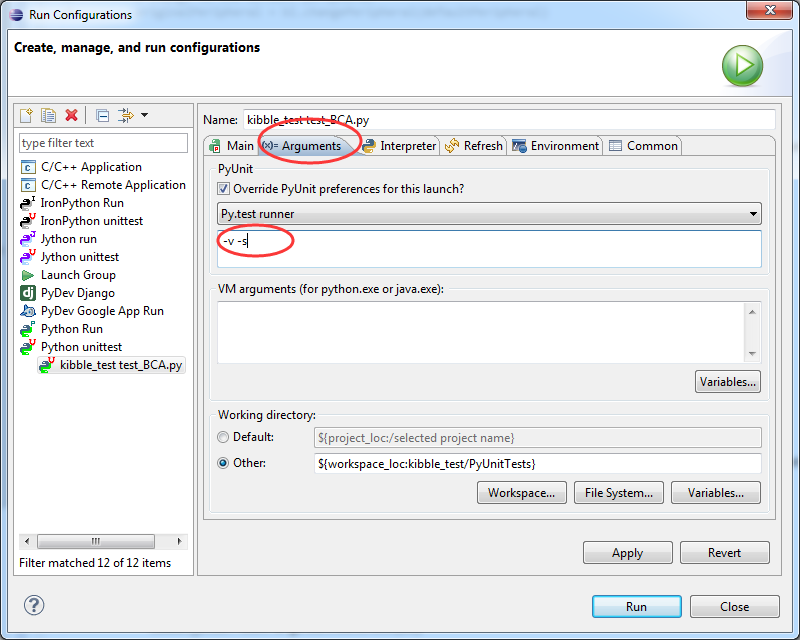
1、测试前，需要对板子进行配置。比如配置测试的板子型号，串口号等。打开test\_config.py



2、配置Eclipse。



Run configuration进行下面的配置：-s 表示将log显示



# 2、Process of automation

这里划分为三个部分：

大鹏 🡪 interact.py 🡪 automation执行部分

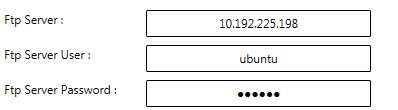
## A、大鹏工作流程

1、首先从git上面抓取kibble release包

2、Build：

python freemv.py C:/Dpc/FreeMV/cfg/TWR\_K65F18\_build\_cfg.ini

这部分工作完成后，生成elf文件。同时传送到FTP上面



3、run :

python freemv.py C:/Dpc/FreeMV/cfg/TWR\_K65F18\_run\_cfg.ini

Elf文件会从ftp上面抓取下来存放到：

C:/Dpc/stationscript/tmp/8583569/tower\_bootloader.elf

run log（cmd窗口log）跟blhost跑出来的log存放地址：

C:\Dpc\FreeMV\result\parallel\_run\TWR\_K65F18\latest\log

跑一个case，log全部清空，更新一次

这个log是将pytest生成的log复制过去的

## B、interact.py是沟通大鹏与automation的桥梁

interact所做的工作：

1. CfgStore获取基本配置：

例如：使用额外设，com号，jlink号等等

1. 使用pytest.exe –v --collect-only抓取所有的case。根据大鹏提供得case name，进行测试。

case测试其实就是调用了如下的语句：

py.test.exe -v -s test\_BCA.py::TestBootloaderConfigurationArea::test\_tag

1. 根据pytest跑出来的结果传送给freemv

Pass ：FREEMV\_INTERACT\_RESULT = 0

skipped、error、Fail ：FREEMV\_INTERACT\_RESULT = 1

1. 将测试log保存到：

C:\Dpc\FreeMV\result\parallel\_run\TWR\_K65F18\latest\log\result.txt

大鹏会从这里抓取log，然后显示到网页上

## C、以pytest为框架的automation工作流程

以test\_BCA.py为例。

1、程序从第一条语句执行，找到class TestBCA后，进行connecting，收集所有以test\_打头的函数。收集完后，log会打印出收集了多少个item，同时开始测试

collected 1 items

test\_BCA.py::TestBootloaderConfigurationArea::test\_tag

2、接下来执行conftest.py中的tgt()函数

1. 判断以CPU命名的文件夹中是否存在bltarget\_config.py

B、执行bltarget\_config.py 同时将所有变量存放到targetConfig字典中

targetConfig = locals().copy()

# Add the file path for bltarget\_config.py to targetConfig

targetConfig[*'\_\_file\_\_'*] = targetConfigFile

# Add the script file name to targetConfig

targetConfig[*'\_\_name\_\_'*] = *'bltarget\_config'*

# Note that, all the variables in bltarget\_config.py will store in the dictionary targetConfig

execfile(targetConfigFile, globals(), targetConfig)

下面说说bltarget\_config.py这个文件

执行完bltarget\_config.py后，我们将一些变量存放到targetconfig字典中，这是这个文件存在的最主要目的。

变量有如下几种：

A、使用get\_property得到的值，比如说flashtotalsize、bootloaderversion等。

使用一个循环进行get\_property的动作，将得到的result值存放到表propertyValues中。最后将表中的值存放到bootloaderVersion等变量中。

所以我们在测试的log中会发现有好多get-property的动作

while propertyTag < bootloader.properties.kPropertyTag\_ExternalMemoryAttrubutes:

status, result = bl.getProperty(propertyTag)

B、板子相关的属性

板子是否支持CRC校验，app的起始地址，flash对齐的字节，CPU的名字等等

3、执行conftest.py中bl函数

BL = bltest.BootloaderDevice(tgt, bltest\_config.vectorsDir, bltest\_config.peripheral, bltest\_config.speed, bltest\_config.port,

bltest\_config.loadTarget, bltest\_config.resetTarget, bltest\_config.usePing)

这段的意思是，创建一个对象，名字叫bl。类的定义在bltest.py中bootloaderDevice。

到bltest.py中查找bootloaderDevice这个类，发现继承与bootloader类。所以当执行完Conftest.py中的bl函数后，以类名bl就可以使用bltest.py中bootloaderDevice跟bootloader两个类中的函数了

4、执行test\_BCA.py中定义的函数

如果有setup()函数，每个函数执行前先执行这个函数，然后执行以test\_开头的函数。

如果有teardown()函数，执行完每个函数有函数的测试后，最后执行teardown()函数

此代码用来添加一个路径，或者一个包含完整路径的文件

targetBaseDir **=** os.path.join**(**os.path.dirname**(**\_\_file\_\_**),** **'..',** **'targets',** cpu**)**

# 3、命令方式使用jlink

比较好的一个讲解网址：

http://blog.163.com/nesnil@126/blog/static/380812982012223104416462/

GDB server

A 使用C:\Program Files (x86)\SEGGER\JLink\_V502h\目录下面的JlinkGDB server通过TCP的方式开启一个GDB server，进行监听

JLinkGDBServerCL.exe -select usb=359695381 -if SWD -port 2724 -device MKV46F256XXX16 -speed auto –singlerun



B 使用C:\MinGW\bin\gdb.exe 发送指令链接到GDB server

set tcp connect-timeout 10

set remotetimeout 10

target remote localhost:2724

monitor speed auto

monitor flash device = MKV46F256XXX16

monitor endian little

monitor reset 0

monitor waithalt 5000

load"E:/Git\_Repos/app\_data/2.0/kibble/mcu\_kibble\_val/BootloaderBinary/IAR/release\_tower.elf" //下载elf文件

monitor reg SP = (0x0)

monitor reg PC = (0x4)

monitor go//运行bootloader

q

C:\MinGW\bin\gdb.exe -x E:\Git\_Repos\app\_data\2.0\kibble\mcu\_kibble\_val\common\flash\_image\_HhQyknCwPDbLgBFNUtqi.init

# 4、GNU工具的使用

Windows下安装MinGW

以下都是linux环境下

前提：装了objdump

1.将a.elf 文件反汇编为a.txt文件 (txt的内容是进elf反汇编后的汇编代码)

命令如下:

[如果](http://blog.sina.com.cn/s/blog_574d08530100i0aj.html)你装的是arm-elf-objdump

那么arm-elf-objdump -d a.elf > 1.txt

如果你装的是arm-eabi-objdump

那么arm-eabi-objdump -d a.elf > 1.txt

arm-elf-objdump:反汇编工具，将elf工具，转变成汇编代码。重要程度：\*\*\*\*\*  
arm-elf-objdump -d a.out > 1.txt将a.out 转变成汇编代码并输出到1.txt文件

2.将elf文件转换为bin文件，将bin文件转换为hex文件  
arm-elf-objcopy：代码转换工具，可以将elf文件转变成hex或bin文件。重要程度:\*\*\*\*\*  
arm-elf-objcopy -O binary a.out a.bin,将a.out转换成a.bin  
arm-elf-objcopy -I binary -O ihex a.bin a.hex将abin转换成a.hex

# 5、Python文件转换为exe文件

1.安装pywin，如果提示找不到python2.7，则运行register.py

2.解压pyinstaller

3.将写好的.py文件放到2中解压的文件夹中

4.运行python pyinstaller.py -F XX.py（3中的.py文件）

5.2中文件夹里面会成3同名的文件夹，里面有exe文件

# 6、Python语言相关

Google python入门学习：

<https://developers.google.com/edu/python/introduction?hl=zh-CN&csw=1>

python比较全面的学习网址

<http://sofar.blog.51cto.com/353572/1381685>

中文快捷入门教程：

<http://www.cnblogs.com/vamei/archive/2012/09/13/2682778.html>

http://www.yiibai.com/python/python\_quick\_guide.html

带编译环境的入门学习：

<https://www.shiyanlou.com/courses/running/649>

1、看如下的代码

#!/usr/bin/env python

# import modules used here -- sys is a very standard one

import sys

# Gather our code in a main() function

def main():

print 'Hello there', sys.argv[1]

# Command line args are in sys.argv[1], sys.argv[2] ...

# sys.argv[0] is the script name itself and can be ignored

# Standard boilerplate to call the main() function to begin

# the program.

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

main()

Running this program from the command line looks like:

$ python hello.py Guido

Hello there Guido

$ ./hello.py Alice ## without needing 'python' first (Unix)

Hello there Alice

当.py文件直接运行时，特殊变量\_\_name\_\_就是\_\_main\_\_.若.py文件作为其他文件的输入而非直接运行时，\_\_name\_\_就不是“\_\_main\_\_”而是本身文件名

## Code Checked at Runtime

def main():

if name == 'Guido':

print repeeeet(name) + '!!!'

else:

print repeat(name)

如上的代码中，没有定义repeeeet（）函数，但是如果name不等于’Guido’ ，此程序也能够顺利运行。因为Python 在编译的时候只做一点点的检查，只有在代码执行到这个地方的时候进行错误检查。

## Online help, help(), and dir()

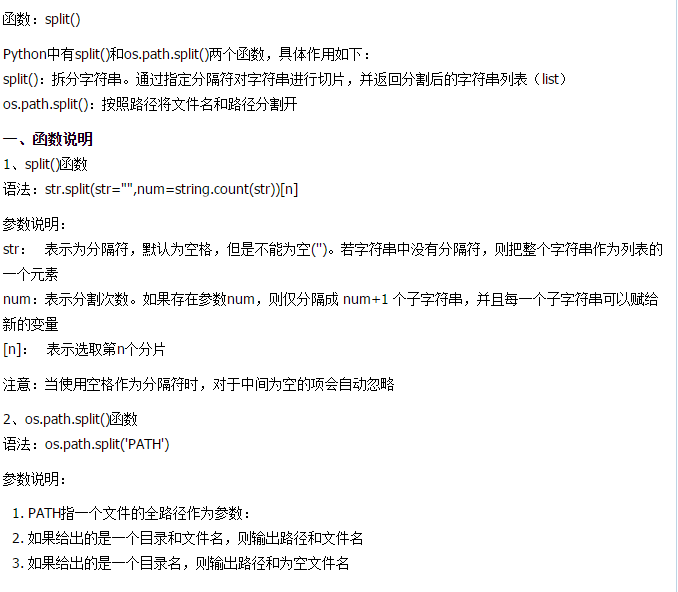
There are a variety ways to get help for Python.

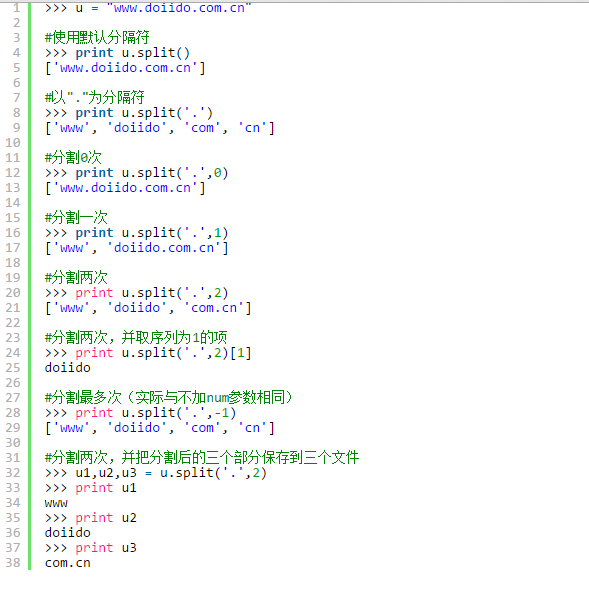
* Do a Google search, starting with the word "python", like "python list" or "python string lowercase". The first hit is often the answer. This technique seems to work better for Python than it does for other languages for some reason.
* The official Python docs site — [docs.python.org](http://docs.python.org/) — has high quality docs. Nonetheless, I often find a Google search of a couple words to be quicker.
* There is also an [official Tutor mailing list](http://mail.python.org/mailman/listinfo/tutor) specifically designed for those who are new to Python and/or programming!
* Many questions (and answers) can be found on [StackOverflow](http://stackoverflow.com/questions/tagged/python) and [Quora](http://quora.com/Python-programming-language).
* Use the help() and dir() functions (see below).

Inside the Python interpreter, the help() function pulls up documentation strings for various modules, functions, and methods. These doc strings are similar to Java's javadoc. The dir() function tells you what the attributes of an object are. Below are some ways to call help() and dir() from the interpreter:

* help(len) — help string for the built-in len() function; note that it's "len" not "len()", which is a **call** to the function, which we don't want
* help(sys) — help string for the sys module (must do an import sys first)
* dir(sys) — dir() is like help() but just gives a quick list of its defined symbols, or "attributes"
* help(sys.exit) — help string for the exit() function in the sys module
* help('xyz'.split) — help string for the split() method for string objects. You can call help() with that object itself or an **example** of that object, plus its attribute. For example, calling help('xyz'.split) is the same as calling help(str.split).
* help(list) — help string for list objects
* dir(list) — displays list object attributes, including its methods
* help(list.append) — help string for the append() method for list objects

split







## 序列

序列是一组有顺序元素的集合。以C语言为例，从内存的角度来讲，数组跟结构体都是组织内存的一种形式，数组中的元素类型必须要统一，结构体则可以包含多种类型的数据。

序列有两种：

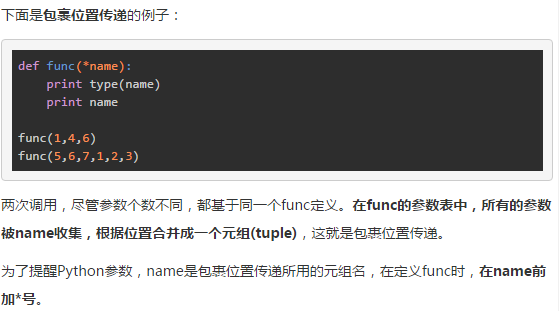
1、tuple（元组）

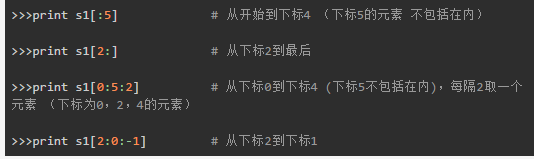
S1 = （2 ,1.3 ,’love’ ,False）元组中的元素定义完后不可以修改

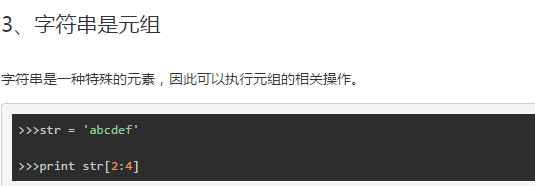
S1[0] = 2 //error

2、list（表）

S1 = [2 ,1.3 ,’love’ ,False]  
 S1[0] = 5 //ok



1 





## For 循环

For a in [1,2,4.5,6]

Print a//输出表中的所有数据，每次从表[1,2,4.5,6]中取一个值，然后打印出来



# data = ‘fdasfdsf’

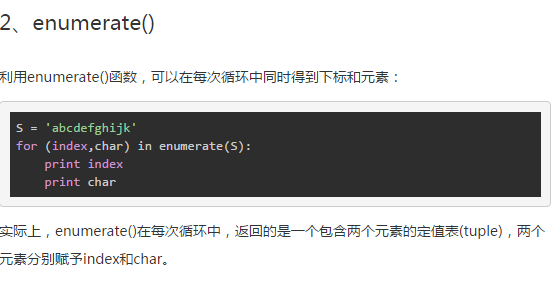
# for i in range(0,len(data),1):

# print ord(data[i])

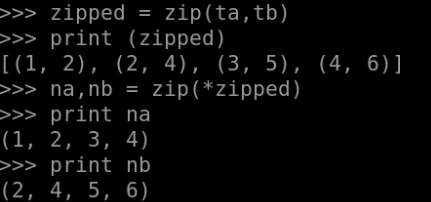
使用了range函数，步长为1，挨个输出字符的十进制数











## 词典

Dic = {‘li’:95,’wang’:45,’jiang’:36}

每一个key对应一个值

内置函数Locals（）返回本地变量

内置函数Globals（）返回全局变量

内置函数dir（）返回内置跟定义的变量

A = 10

Dir(a) 返回a的所有方法跟属性

当词典中有两个相同的键值时，len（dic）将不计入重复键值的个数。

例如：

Dic = {‘li’:95,’wang’:45,’jiang’:36，‘li’：23}

Len（dic）=3而不是4

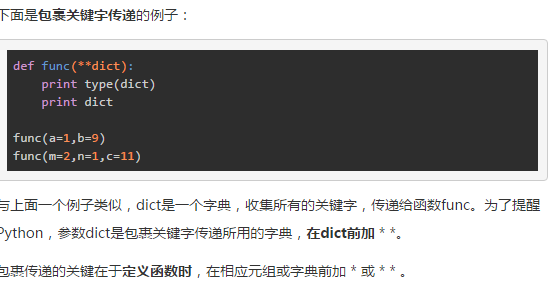
但是list不会有这个问题，相同的元素也会计数

Python内置(built-in)函数随着python解释器的运行而创建。在Python的程序中，你可以随时调用这些函数，不需要定义。最常见的内置函数是:

print("Hello World!")

http://www.cnblogs.com/vamei/archive/2012/11/09/2762224.html





## Namedtumple

使用nametumple的好处就是能知道给摸个变量赋值，能够用名字记录，每个变量都有了自己的名字，便于记忆。而dic的方式，只能用下标去记录，不够形象化

##namedtumple

from collections import namedtuple

student = namedtuple(*'hah'*,*'age, height, weight'*)

# b = student('33', 35, 69)

# print b.age

b = {*'bb'*: student}

b[*'bb'*].age = 22222

print b[*'bb'*].age

##使用dictionary的方式

a = [1,2,3]

dic1 = {*'aa'*: a}

dic1[*'aa'*][0] = 33333

print dic1[*'aa'*][0]

#打印结果：

22222

33333

## 引入模块

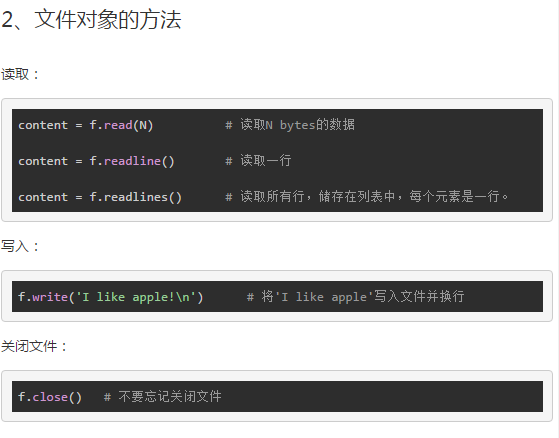


## 路径搜索



## 文本输入输出





## 动态类型

表类似于C中的数组，因此表名字就相当于地址，使用到表的时候要注意，此时使用的是地址传递，会修改表中参数的值。

字典类似于C中的二维数组，key跟键值对应两个维度



## 文件操作

Python中的路径采用跟linux一样的做法，使用正斜杠/

反斜杠\有转义字符的意思，所以在得到系统文件路径的时候，需要使用\\来处理文件路径。在windows 平台下面得到的路径是反斜杠\

直接使用路径’c\\windows’也是可以的，注意要是[\\，第一个\](file:///\\，第一个\)表示转义字符

抓取当前文件的路径：

print (sys.path[0])（当前文件夹或者文件的路径）

或者

this\_file = inspect.getfile(inspect.currentframe())

得到的文件路径是windows下面以反斜杠\为路径分隔符的，所以必须进行处理转换成python的文件格式/

current\_path = os.path.dirname(this\_file).replace('\\','/')

print current\_path

得到路径后判断是否为python认定的路径或者文件

print (os.path.isdir(current\_path))

print (os.path.isfile(test\_file))

添加文件

test\_file = os.path.join(current\_path,'test.txt')

## Pipe操作

使用到subprocess.py

<http://blog.csdn.net/g457499940/article/details/17068277>

得到进程返回值

process = subprocess.Popen(theArgs, stdout=subprocess.PIPE, stderr=subprocess.STDOUT)

输出全部数据

process.communicate() //数据连接在一起

process.communicate() //数据按照cmd里面的格式输出

process.stdout.readline //数据每次输出一行

得到的数据如何处理？

print process.communicate()[0] // communicate（）返回的是一个元祖（stdoutdata， stderrordata）

print process.stdout.read()//读出所有的out内容

详见split

### 让命令行窗口暂停

raw\_input('Press Enter to exit...')

## Pytest

Pytest使用说明：

<http://www.cnblogs.com/fnng/p/4765112.html>

<http://pytest.org/latest/example/index.html>

<http://pythontesting.net/start-here/>

最后，pytest是如果识别测试用例的呢？

它默认使用检查以test\_ \*.py 或\*\_test.py命名的文件名，在文件内部查找以test\_打头的方法或函数，并执行它们。

py.test 框架会在它测试的项目中寻找 conftest.py 文件，然后在这个文件中寻找针对整个项目的选项，比如是否检测并运行 doctest 以及应该使用哪种模式检测测试文件和函数。

从Python代码中调用pytest

pytest中同样提供了main() 来函数来执行测试用例。

pytest/

├── test\_sample.py

├── test\_class.py

└── test\_main.py

此目录为我们练习的目录，打开test\_mian.py

[复制代码](javascript:void(0);)

import pytest

def test\_main():

assert 5 != 5

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

pytest.main()

[复制代码](javascript:void(0);)

 直接运行该程序，sublime 中按Ctrl+B 运行。结果如下：

[复制代码](javascript:void(0);)

============================= test session starts =============================

platform win32 -- Python 2.7.10 -- py-1.4.30 -- pytest-2.7.2

rootdir: D:\pyse\pytest, inifile:

collected 4 items

test\_class.py .F

test\_main.py F

test\_sample.py F

================================== FAILURES ===================================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ TestClass.test\_two \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

self = <test\_class.TestClass instance at 0x000000000304F548>

def test\_two(self):

x = "hello"

> assert x == "hi"

E assert 'hello' == 'hi'

E - hello

E + hi

test\_class.py:11: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_main \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_main():

> assert 5 != 5

E assert 5 != 5

test\_main.py:4: AssertionError

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_answer \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_answer():

> assert func(3) == 5

E assert 4 == 5

E + where 4 = func(3)

test\_sample.py:9: AssertionError

===================== 3 failed, 1 passed in 0.03 seconds ======================

[Finished in 0.3s]

　　从执行结果看到，main() 默认执行了当前文件所在的目录下的所有测试文件。

　　那么，如果我们只想运行某个测试文件呢？可以向main()中添加参数，就像在cmd命令提示符下面一样：

[复制代码](javascript:void(0);)

#coding=utf-8

import pytest

def test\_main():

assert 5 != 5

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

pytest.main("-q test\_main.py") # 指定测试文件

-q  为quiet。表示在安静的模式输出报告诉。加不加这个参有什么区别呢？ 读者可以对比一下两次输出的日志。其实，就是少了一些pytest的版本信息。

那如果我想运行某个目录下的测试用例呢？指定测试目录即可。

[复制代码](javascript:void(0);)

#coding=utf-8

import pytest

def test\_main():

assert 5 != 5

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

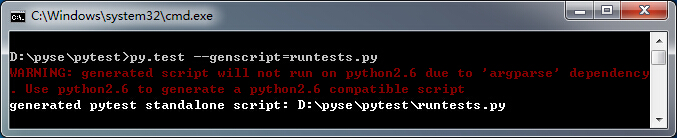
pytest.main("d:/pyse/pytest/") # 指定测试目录

**创建运行测试脚本**

　　有时候我们的测试用例文件分散在不同的层级目录下，通过命令行的方式运行测试显示不太方便，如何编写一个运行所有测试用例的脚本呢？ pytest可以自动帮我们生成这样的脚本。

**>py.test --genscript=runtests.py**

http://www.cnblogs.com/fnng/p/4765112.html



### Fixture

比较好的资料：

http://blog.csdn.net/xibeichengf/article/details/50589235

这里重点说下pytest中的fixture

Fixture的功用就是在模块、函数、类、方法，执行前后调用一段设计好的代码。

http://pytest.org/latest/fixture.html

经典的xUnit

<http://pytest.org/latest/xunit_setup.html#xunitsetup>

首先谈谈经典的fixture，叫做xUnit：

def **multiply**(a,b):

return a \* b

def **setup\_module**(module):

print (*"\n"*)

print (*"setup\_module================>"*)

def **teardown\_module**(module):

print (*"teardown\_module=============>"*)

class **Test**:

*@classmethod*

def **setup\_class**(cls):

print (*"setup\_class------>"*)

*@classmethod*

def **teardown\_class**(cls):

print (*"teardown\_class--->"*)

def **setup\_method**(*self*,method):

print (*"setup\_method------>"*)

def **teardown\_method**(*self*,method):

print (*"teardown\_method--->"*)

def **test\_numbers\_3\_4**(*self*):

print *'test\_numbers\_3\_4'*

assert multiply(3,4) == 12

def **test\_strings\_a\_3**(*self*):

print *'test\_strings\_a\_3'*

assert multiply(*'a'*,3) == *'aaa'*

如上这段代码执行的效果如下

setup\_module================>

setup\_class------>

setup\_method------>

test\_numbers\_3\_4

.teardown\_method--->

setup\_method------>

test\_strings\_a\_3

.teardown\_method--->

teardown\_class--->

teardown\_module=============>

可以发现：

1、setup/tear\_down class/method 都必须写在类的内部，

2、setup/tear\_down module写在类的外面

3、setup\_class跟teardown\_class在类的前后执行一次，module在module调用前后执行一次，method在每个方法的前后分别执行一次。

再看下面一段代码：

将如上的代码中加入function跟function相关的fixture

def **multiply**(a,b):

return a \* b

# =====fixtures========

def **setup\_module**(module):

print (*"\n"*)

print (*"setup\_module================>"*)

def **teardown\_module**(module):

print (*"teardown\_module=============>"*)

def **setup\_function**(function):

print (*"setup\_function------>"*)

def **teardown\_function**(function):

print (*"teardown\_function--->"*)

def **test\_numbers\_3\_4**():

print *'test\_numbers\_3\_4'*

assert multiply(3,4) == 12

def **test\_strings\_a\_3**():

print *'test\_strings\_a\_3'*

assert multiply(*'a'*,3) == *'aaa'*

运行结果如下：

setup\_module================>

setup\_function------>

test\_numbers\_3\_4

.teardown\_function--->

setup\_function------>

test\_strings\_a\_3

.teardown\_function--->

teardown\_module=============>

可以发现：

Function在函数的执行前后分别执行一次，module只会在程序开始前后各执行一次

接下来，。。。。过段时间再写吧，专项ultra

### 总结

Fixture比较好的资料：

http://blog.csdn.net/xibeichengf/article/details/50589235

conftest.py就是一个fixture的集合。。

里面定义的fixture，如果在其他的.py文件中用到的话，都会执行。。

#### Pytest.skip的使用

import pytest

*@pytest.fixture*(autouse=True)

def **ss**(request):

pytest.skip(*"should not be tested"*)

def **test**(bl):

print *'hah'*

bl.output(*'==============happy new year=================='*)

执行.py的时候，pytest 执行到这里的时候，就会skip整个.py文件。。即便.py文件中有test item，也不care，直接skip。。

#### Conftest的使用

新建一个contest.py,内容为：

import pytest

class **hah**(object):

def **\_\_init\_\_**(*self*):

*self*.a = *'aaa'*

*self*.b = *'bbb'*

def **output**(*self*,argument):

print *'==================='*

print *self*.a

print *self*.b

print argument

*@pytest.fixture*(scope = *'function'*)

def **bl**(request):

ee = hah()

return ee

创建一个test.py，内容为：

def **test**(bl):

print *'hah'*

bl.output(*'==============happy new year=================='*)

当pytest执行test fuction的时候，会找从conftest中查找bl。

bl是一个function，功能是返回一个类的实例。。

一般的，如果我们在python中调用bl的话，用法为bl().output()

但是pytest框架，只需要bl.output()就能调用output function了

## OS

### Chdir()

os.chdir() 方法用于改变当前工作目录到指定的路径。

语法

**chdir()**方法语法格式如下：

os.chdir(path)

参数

* **path** -- 要切换到的新路径。

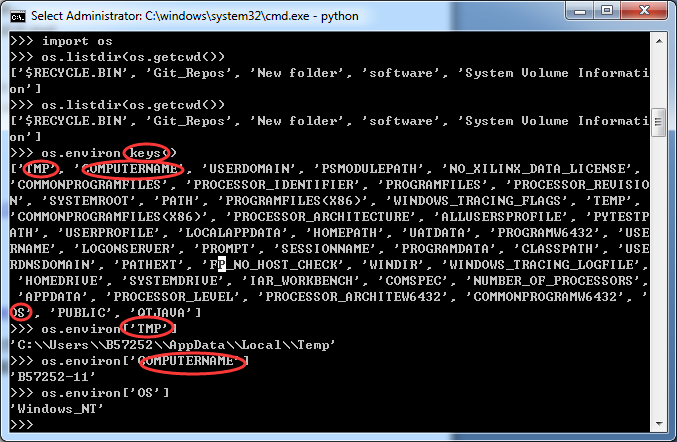
返回值

如果允许访问返回 True , 否则返回False。

### OS environ

<http://blog.csdn.net/junweifan/article/details/7615591>

使用对于官方的解释，environ是一个字符串所对应环境的映像对象。这是什么意思呢？举个例子来说，environ['HOME']就代表了当前这个用户的主目录。



### \_\_name\_\_

\_\_name\_\_系统变量指示模块应如何被加载，他的值为"\_\_main\_\_"时表示当前模块是被直接执行。  
由于主程序代码无论模块是被导入还是直接被执行都会运行，所以我们需要一种方式在运行时检测该模块是被导入还是被直接执行。该方式也就是\_\_name\_\_系统变量。如果模块是被导入，\_\_name\_\_的值为模块名字；如果是被直接执行，\_\_name\_\_的值为"\_\_main\_\_"。  
此处所说的模块可以简单理解为一个.py文件。

  举个例子，下面在test.py中写入如下代码：

print "I'm the first."

if \_\_name\_\_=="\_\_main\_\_":

    print "I'm the second."

并直接执行test.py，结果如下图，可以成功print两行字符串。即，if \_\_name\_\_=="\_\_main\_\_": 语句之前和之后的代码都被执行。

然后在同一文件夹新建名称为import\_test.py的脚本，只输入如代码：

import test  
执行import\_test.py脚本，输出结果如下：

只输出了第一行字符串。即，if \_\_name\_\_=="\_\_main\_\_": 之前的语句被执行，之后的没有被执行。

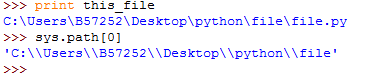
# 7、Python常用代码

## 1、文件操作

### 获取当前文件路径

this\_file = inspect.getfile(inspect.currentframe())

this\_file = sys.path[0]



如上可见，使用ispect得到的文件路径是‘\’，python承认的文件路径是‘/’所以必须对这个这个路径进行处理。\\ 中第一个\是表示转义字符

current\_path = os.path.dirname(this\_file).replace('\\','/')

test\_file = os.path.join(current\_path,'test.txt')



### 判断是否为文件夹\文件

print (os.path.isdir(current\_path))

print (os.path.isfile(test\_file))

添加文件

os.path.join(current\_path , '..' , 'test2.py') //上一层文件夹里面的文件



### 列出文件夹里面的内容

os.listdir(current\_path)



### 文件读写

w     以写方式打开，  
a     以追加模式打开 (从 EOF 开始, 必要时创建新文件)  
r+     以读写模式打开  
w+     以读写模式打开 (参见 w )  
a+     以读写模式打开 (参见 a )  
rb     以二进制读模式打开  
wb     以二进制写模式打开 (参见 w )  
ab     以二进制追加模式打开 (参见 a )  
rb+    以二进制读写模式打开 (参见 r+ )  
wb+    以二进制读写模式打开 (参见 w+ )  
ab+    以二进制读写模式打开 (参见 a+ )

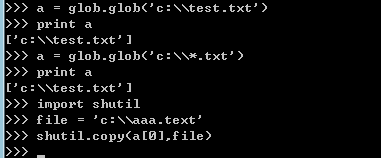
file = os.path.join(self.current\_path,'test.yml') //文件不存在则创建此文件

f = open(file,'w')

f.write(kibble\_release\_package + ':' + '\n')

f.close()

### 文件查找拷贝



Glob.glob(‘c:\\*txt’) //查找所有c根目录下面.txt文件

Shutil.copy(A,B)将文件A的内容复制到文件B中.文件B可以不存在，如上

file = C：[\\aaa.txt](file:///\\aaa.txt) 制定了文件名称，但是文件不存在，使用了copy函数后生成aaa为文件名的文件

详见上一章节

### 处理二进制数据

struct模块中最重要的三个函数是pack(), unpack(), calcsize()

pack(fmt, v1, v2, ...)     按照给定的格式(fmt)，把数据封装成字符串(实际上是类似于c结构体的字节流)

unpack(fmt, string)       按照给定的格式(fmt)解析字节流string，返回解析出来的tuple

calcsize(fmt)                 计算给定的格式(fmt)占用多少字节的内存

http://www.cnblogs.com/gala/archive/2011/09/22/2184801.html

**Flash中读取数据，然后按照高位到低位依照十六进制排序。**

#### 实例：读取PC and stack

法一：

def **read\_register**(bl, address, length):

kFilePath = file\_path(bl)

status, results = bl.readMemory(address, length, kFilePath)

assert status == bootloader.status.kStatus\_Success

readBackBytes = None

with open (kFilePath, *'rb'*) as file:

readBackBytes = file.read(int(length))

file.close()

if length == 1:

format = *'B'*

elif length == 2:

format = *'H'*

elif length == 4:

format = *'L'*

assert readBackBytes != None

value, = struct.unpack(format, readBackBytes)

return value

def **test\_read**(bl):

print hex(read\_register(bl,0x0,4))

print hex(read\_register(bl,0x04,4))

**法二：**

实例化一个字符数组，然后将值存放到数组中，然后从数组中取值

TempArr = array.array(*'c'*,[chr(0x11)])

with open(demoFile, *'rb'*) as file:

TempArr = file.read(8)

file.close()

Stack = (ord(TempArr[3])<<24) + (ord(TempArr[2])<<16) + (ord(TempArr[1])<<8) + (ord(TempArr[0]))

PC = (ord(TempArr[7])<<24) + (ord(TempArr[6])<<16) + (ord(TempArr[5])<<8) + (ord(TempArr[4]))

#### 向二进制文件制定位置写入数据

appDataArray = range(0,4)

with open(binFile, *'rb+'*) as fileObj:

for i in range(0,4):

data = fileObj.read(int(1))

appDataArray[i] = (ord(data))

#value, = struct.unpack('L', data)

print appDataArray

crcResult = crc32.ComputeCRC32().calculate(0xffffffff, appDataArray[0:4])

print crcResult

fileObj.seek(0x3c0,0)

byte.append(crcResult & 0x000000ff)

byte.append((crcResult & 0x0000ff00)>>8)

byte.append((crcResult & 0x00ff0000)>>16)

byte.append((crcResult & 0xff000000)>>24)

fileObj.write(byte)

fileObj.close()

## 从网页下载文件

Python test.py <http://www.python.org/ftp/python/2.7.5/Python-2.7.5.tar.bz2>

import urllib

import os

import sys

import inspect

# a:已经下载的数据块

# b:数据块的大小

# c:远程文件的大小

def **Schedule**(a,b,c):

per = 100.0 \* a \* b / c

if per > 100 :

per = 100

print *'%.2f%%'* % per

#url = 'http://www.python.org/ftp/python/2.7.5/Python-2.7.5.tar.bz2'

#local = url.split('/')[-1]

#this\_file = sys.path[0]

print *'---------------------------------------------------'*

print *'---------Downloading resource from network---------'*

print *'-----------------Richard 20160920------------------'*

print *'---------------------------------------------------'*

print *'Download address:'*

print sys.argv[1]

url = str(sys.argv[1])

this\_file = inspect.getfile(inspect.currentframe())

current\_path = os.path.dirname(this\_file).replace(*'\\'*,*'/'*)

print current\_path

file = os.path.join(current\_path, *'python-2.7.5.tar.bz2'*)

print *'Downloading file:'*

print file

# local = os.path.join('/data/software','Python-2.7.5.tar.bz2')

urllib.urlretrieve(url,file,Schedule)

## 命令行参数

Sys.argv[0]代表运行的python文件名称

Sys.argv[1]代表第一个参数

import sys,getopt

def main():

args = '-a aa -b bb'

opt,args = getopt.getopt(args.split(), 'a:b:')

print opt

#options = dict(opt)

#print options['-a']

for a,b in opt:

if a == '-a':

opt\_a = b

elif a == '-b':

opt\_b = b

else:

print "nothing"

print opt\_a,opt\_b

## socket

<https://www.cnblogs.com/weiman3389/p/6044955.html>

摘要：  只要和网络服务涉及的，就离不开Socket以及Socket编程，下面就说说Python Socket通信的基本原理。 1.Socket     socket也称作“套接字”，用于描述IP地址和端口，是一个通信链的句柄。应用程序通常通过“套接字”向网络发出请求或者应答网络请求。可以列举中国移动或...

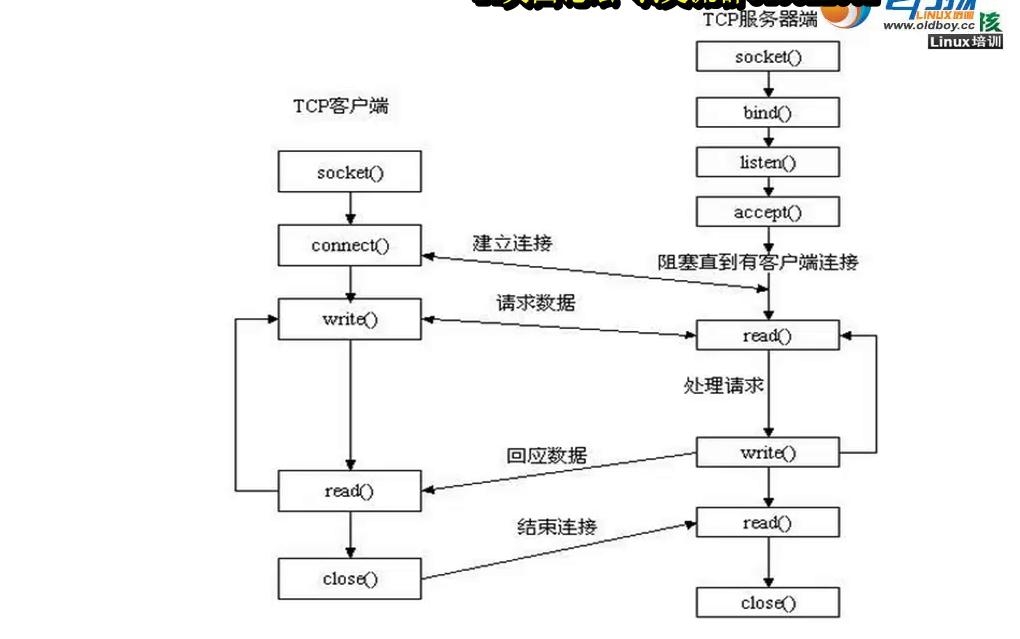
 只要和网络服务涉及的，就离不开Socket以及Socket编程，下面就说说Python Socket通信的基本原理。

1.Socket

    socket也称作“套接字”，用于描述IP地址和端口，是一个通信链的句柄。应用程序通常通过“套接字”向网络发出请求或者应答网络请求。可以列举中国移动或者是中国电信等的电话客服，当然，也可以看下面的图片来作形象的说明。

    socket起源于Unix，而Unix/Linux基本哲学之一就是：一切皆文件，即都可以用“打开open—>读写write/read—>关闭close”模式来操作。Socket就是该模式的一个实现，socket即是一种特殊的文件，一些socket函数就是对其进行的操作（读/写IO、打开、关闭）。

    可以看下面的图示来形象说明：



示例程序：

Server：

Socket –》 bind （绑定）-》listen （监听）-》accept –》communication

import socket

# address = ('127.0.0.1', 10010)

address = socket.gethostname()# get address from local

port = 10010

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # s = socket.socket()

s.bind((address, port))

s.listen(5)

ss, addr = s.accept()

print *'got connect form'*, addr

while True:

ss.send(*'host say hello'*)

re = ss.recv(512)

print re

ss.close()

s.close()

Client ：

Socket –》connect –》 communication

import socket

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

address = socket.gethostname()

port = 10010

s.connect((address, port)) #parameter for connect should be a tumple

while True:

re = s.recv(512)

print re

s.send(*'hah from client'*)

s.close()

## Eval\bin

1 可以把list,tuple,dict和string相互转化。

2 #################################################

3 字符串转换成列表

4 >>>a = "[[1,2], [3,4], [5,6], [7,8], [9,0]]"

5 >>>type(a)

6 <type 'str'>

7 >>> b = eval(a)

8 >>> print b

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

10 >>> type(b)

11 <type 'list'>

12 #################################################

13 字符串转换成字典

14 >>> a = "{1: 'a', 2: 'b'}"

15 >>> type(a)

16 <type 'str'>

17 >>> b = eval(a)

18 >>> print b

19 {1: 'a', 2: 'b'}

20 >>> type(b)

21 <type 'dict'>

22 #################################################

23 字符串转换成元组

24 >>> a = "([1,2], [3,4], [5,6], [7,8], (9,0))"

25 >>> type(a)

26 <type 'str'>

27 >>> b = eval(a)

28 >>> print b

29 ([1, 2], [3, 4], [5, 6], [7, 8], (9, 0))

30 >>> type(b)

31 <type 'tuple'>

test = *'0x1003'*

print type(eval(test))

print bin(eval(test))

print bin(eval(test))[-2:]

<type 'int'>

0b1000000000011

11

## Yaml

安装：

Pip install pyyaml

将字典保存为yaml文件

with open(yaml\_file, *'r'*) as f:

yaml.dump(pattern\_info\_dict, f)

从yaml文件中读取数据，读出来的是一个字典

with open(yaml\_file, *'r'*) as f:

content = yaml.load(f)

## json

actualResults = json.loads(result)

result为json格式的数据，loads返回一个字典

## list组合为string

comport = *'com10 '*

blhost = r*'C:\Users\nxa28190\Desktop\Tools\blhost\win\blhost.exe '*

command = [blhost,*'-p '*,*'-j '* ,*'-- '* ,*'get-property 1'*]

command.insert((command.index(*'-p '*)+1),comport)

print type(command)

command = [str(x) for x in command]

command = *''*.join(command)

print command

print type(command)

<type 'list'>

C:\Users\nxa28190\Desktop\Tools\blhost\win\blhost.exe -p com10 -j -- get-property 1

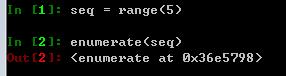
<type 'str'>

## Enumerate

给对象增加索引

* enumerate()是python的内置函数
* enumerate在字典上是枚举、列举的意思
* 对于一个可迭代的（iterable）/可遍历的对象（如列表、字符串），enumerate将其组成一个索引序列，利用它可以同时获得索引和值
* enumerate多用于在for循环中得到计数
* 例如对于一个seq，得到：

(0, seq[0]), (1, seq[1]), (2, seq[2])

* + 1
* enumerate()返回的是一个enumerate对象，例如：   
  

## enumerate()使用

* 如果对一个列表，既要遍历索引又要遍历元素时，首先可以这样写：

list1 = ["这", "是", "一个", "测试"]

for i in range (len(list1)):

print i ,list1[i]

* 上述方法有些累赘，利用enumerate()会更加直接和优美：

list1 = ["这", "是", "一个", "测试"]

for index, item in enumerate(list1):

print index, item

>>>

0 这

1 是

2 一个

3 测试

* enumerate还可以接收第二个参数，用于指定索引起始值，如：

list1 = ["这", "是", "一个", "测试"]

for index, item in enumerate(list1, 1):

print index, item

>>>

1 这

2 是

3 一个

4 测试

### 补充，统计文件行数

如果要统计文件的行数，可以这样写：

count = len(open(filepath, 'r').readlines())

* 1

这种方法简单，但是可能比较慢，当**文件比较大**时甚至不能工作。

可以利用enumerate()：

count = 0

for index, line in enumerate(open(filepath,'r'))：

count += 1

## Multiprocessing

Python 运行后，会有一个main进程。

使用了multiprocessing后可以开启多个进程。。每个进程的空间是独立的，如果需要数据共享的话需要使用到pipe，queue，value等方法

### 新建进程

import os

import multiprocessing

import time

#==================

# input worker

def **inputQ**(queue):

info = str(os.getpid()) + *'(put):'* + *'str(time.time())'*

queue.put(info)

print *'input'* + str(os.getpid())

# output worker

def **outputQ**(queue,lock):

info = queue.get()

# lock.acquire()

print (str(os.getpid()) + *'(get):'* + info)

# lock.release()

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

#===================

# Main

record1 = [] # store input processes

record2 = [] # store output processes

lock = multiprocessing.Lock() # To prevent messy print

queue = multiprocessing.Queue()

# input processes

for i in range(10):

process = multiprocessing.Process(target=inputQ,args=(queue,))

process.start()

record1.append(process)

for p in record1:

p.join()

# output processes

for i in range(10):

process = multiprocessing.Process(target=outputQ,args=(queue,lock))

process.start()

print i

record2.append(process)

# queue.close() # No more object will come, close the queue

for p in record2:

p.join()

print 'main'

这段代码开启了10个进程，向queue里面传送自己进程的ID。。

使用P.join阻塞其他进程，让这10个进程执行完毕后

再开10个进程，用来从queue里面获取上面进程输入的信息。

使用P.join阻塞其他进行，让这10个进程执行完毕。然后程序往下走

### Queue

Multiprocessing.queue(3) //三个队列，只允许最多存放三个数据

Queue是先进先出的结构

一个进程向里面传送数据，另一个进程获取数据。。先传送的先获取

### Value

Value = Multiprocessing.value(‘i’, 1)

设置一个int 型变量，值为1.。其他进程都可以操作这个数据

调用方法：

Value.value

### Pipe

进程间通过管道传递数据

def loger( pp):

pp.send(“hahh”)

def record(pp):

print 'entring record'

while True:

buffer = pp.recv()

if buffer:

with open('record.txt', 'a') as f:

f.write(buffer)

pp = multiprocessing.Pipe()

logr = mp.Process(target=loger, args=(running\_flag, q, mode, pp[0]))

recorder = mp.Process(target=record, args=(pp[1],))

logr.start()

recorder.start()

## os

### os.environ.items()

环境变量，字典的形式呈现出来

[('TMP', 'C:\\Users\\nxa28190\\AppData\\Local\\Temp\\1'), ('OS', 'Windows\_NT'), ('PUBLIC', 'C:\\Users\\Public'), ('USERPROFILE', 'C:\\Users\\nxa28190')]

设置环境变量：

Os.environ[‘hah’] = ‘123’

增加一个环境变量，名字为hah，内容为123

os.environ.get(‘hah’,’567’)

如果存在环境变量hah，则hah内容为123.。

如果不存在环境变量hah，则创建环境变量hah，内容为567

### pipe, system

for line in os.popen("ps -aux").read().splitlines():

if "nios" in line:

os.system("kill -9 " + line.split()[1])

### getcwd

cur\_dir = os.getcwd()

### chdir

os.chdir("../zebu\_tb\_mx8\_qx/zebu.emulation")

### os.listdir(dirname)

列出dirname下的目录和文件

### os.getcwd()

获得当前工作目录

### os.curdir

返回当前目录（'.')

### os.chdir(dirname)

改变工作目录到dirname

### os.path.isdir(name)

判断name是不是一个目录，name不是目录就返回false

### os.path.isfile(name)

判断name是不是一个文件，不存在name也返回false

### os.path.exists(name)

判断是否存在文件或目录name

### os.path.getsize(name)

获得文件大小，如果name是目录返回0

### os.path.abspath(name)

获得绝对路径

### os.path.normpath(path)

规范path字符串形式

### os.path.split(name)

分割文件名与目录（事实上，如果你完全使用目录，它也会将最后一个目录作为文件名而分离，同时它不会判断文件或目录是否存在）

### os.path.splitext()

分离文件名与扩展名

### os.path.join(path,name)

连接目录与文件名或目录

os.path.basename(path)

返回文件名

### os.path.dirname(path)

返回文件路径

### 遍历所有文件

os.walk(dir)

dir = *'C:\\Users\\b57252\\Desktop\\previous\_log'*

# print os.path.splitext(dir)[1]

count = 0

for root, dirs, files in os.walk(dir):

for name in files:

file\_router = os.path.join(root,name)

file\_type = os.path.splitext(file\_router)[1]

if *'.txt'* == file\_type:

count = count +1

print file\_router

print file\_type

print count

获取txt文件数量

依次返回如下，直到遍历所有的内容

当前目录， 当前目录下的文件夹，当前目录下的文件

子目录，子目录下的文件夹，子目录下的文件

os.walk()可以得到一个三元tupple(dirpath, dirnames, filenames)，其中第一个为起始路径，第二个为起始路径下的文件夹，第三个是起始路径下的文件。

其中dirpath是一个string，代表目录的路径，dirnames是一个list，包含了dirpath下所有子目录的名字。filenames是一个list，包含了非目录文件的名字。

dirnames和filenames不包含路径信息，如果需要得到全路径，需要使用os.path.join(dirpath, name)。

## configparser

新建一个ini文件，内容如下：

[config]

cpu = m4

core = hah

boot = dd

[speed]

dd\_speed = 100

ahh\_speed = 200

当前目录下创建一个py文件，脚本如下：

import ConfigParser

config = ConfigParser.SafeConfigParser()

config.read('config.ini')

print config.get('config','core')

print config.get('speed','dd\_speed')

运行结果如下：

hah

100

[Finished in 0.2s]

## Logging

Python 代码如下：

import logging

import logging.config

import os

import time

# def log\_init():

current\_time = time.strftime('%Y-%m-%d-%H-%M-%S',time.localtime(time.time()))

file\_path = os.path.abspath(os.path.dirname(\_\_file\_\_)).replace('\\','/')

main\_path = os.path.abspath(os.path.join(file\_path, '..')).replace('\\','/')

log\_file = main\_path + '/result/' + current\_time + '\_run.txt'

log\_conf = main\_path + '/config/logging.conf'

if not os.path.exists(os.path.dirname(log\_file)):

os.makedirs(os.path.dirname(log\_file))

else:

pass

# a = dict(log\_file=log\_file)

# print a

logging.config.fileConfig(log\_conf, defaults=dict(log\_file=log\_file))

logging.info(‘test’)

configure文件如下:

# logger.conf

###################################

[loggers]

keys=root

[logger\_root]

level=DEBUG

handlers=handlerScreen,handlerFile

###################################

[handlers]

keys=handlerScreen,handlerFile

[handler\_handlerScreen]

class=StreamHandler

level=DEBUG

formatter=formScreen

args=(sys.stderr,)

[handler\_handlerFile]

class=FileHandler

level=INFO

formatter=formFile

args=('%(log\_file)s', 'a')

###################################

[formatters]

keys=formScreen,formFile

[formatter\_formScreen]

format=%(asctime)s %(message)s

datefmt=%Y-%m-%d %H:%M:%S

[formatter\_formFile]

format=%(asctime)s %(filename)-21s[line:%(lineno)-3d] %(levelname)s %(message)s

datefmt=%Y-%m-%d %H:%M:%S

## Subprocess

Subprocess的出现时为了替代os.pipe跟os.system

### Popen

需要获取数据的话使用popen，否则使用call

Os的用法[pipe, system](#_pipe,_system)

Import subprocess

p = subprocess.Popen(cmd, 0, None, subprocess.PIPE, subprocess.PIPE, None, shell=True)

(sout, serr) = p.communicate()

Print sout,serr

### Call

subprocess.call(cmd)

## 正则表达式

<https://www.cnblogs.com/dyfblog/p/5880728.html>

### 函数

#### compile(pattern, flags=0)

给定一个正则表达式 pattern，指定使用的模式 flags 默认为0 即不使用任何模式,然后会返回一个 SRE\_Pattern (参见 第四小节 re 内置对象用法) 对象

regex = re.compile(".+")

print regex

# output> <\_sre.SRE\_Pattern object at 0x00000000026BB0B8>

这个对象可以调用其他函数来完成匹配，一般来说推荐使用 compile 函数预编译出一个正则模式之后再去使用，这样在后面的代码中可以很方便的复用它，当然大部分函数也可以不用 compile 直接使用，具体见 findall 函数

#### finditer(pattern, string, flags=0)

参数和作用与 findall 一样，不同之处在于 findall 返回一个列表， finditer 返回一个迭代器(参见<http://www.cnblogs.com/huxi/archive/2011/07/01/2095931.html> )， 而且迭代器每次返回的值并不是字符串，而是一个 SRE\_Match (参见 第四小节 re 内置对象用法) 对象，这个对象的具体用法见 match 函数。

[复制代码](javascript:void(0);)

s = '''first line

second line

third line'''

regex = re.compile("\w+")

print regex.finditer(s)

# output> <callable-iterator object at 0x0000000001DF3B38>

for i in regex.finditer(s):

print i

# output> <\_sre.SRE\_Match object at 0x0000000002B7A920>

# <\_sre.SRE\_Match object at 0x0000000002B7A8B8>

# <\_sre.SRE\_Match object at 0x0000000002B7A920>

# <\_sre.SRE\_Match object at 0x0000000002B7A8B8>

# <\_sre.SRE\_Match object at 0x0000000002B7A920>

# <\_sre.SRE\_Match object at 0x0000000002B7A8B8>

#### findall(pattern, string, flags=0)

参数 pattern 为正则表达式, string 为待操作字符串, flags 为所用模式，函数作用为在待操作字符串中寻找所有匹配正则表达式的字串，返回一个列表，如果没有匹配到任何子串，返回一个空列表。

s = '''first line

second line

third line'''

# compile 预编译后使用 findall

regex = re.compile("\w+")

print regex.findall(s)

# output> ['first', 'line', 'second', 'line', 'third', 'line']

# 不使用 compile 直接使用 findall

print re.findall("\w+", s)

# output> ['first', 'line', 'second', 'line', 'third', 'line']

#### search(pattern, string, flags=0)

函数类似于 match，不同之处在于不限制正则表达式的开始匹配位置

[复制代码](javascript:void(0);)

s = '''first line

second line

third line'''

# 需要从开始处匹配 所以匹配不到

print re.match('i\w+', s)

# output> None

# 没有限制起始匹配位置

print re.search('i\w+', s)

# output> <\_sre.SRE\_Match object at 0x0000000002C6A920>

print re.search('i\w+', s).group()

# output> irst

#### match(pattern, string, flags=0)

使用指定正则去待操作字符串中寻找可以匹配的子串, 返回匹配上的第一个字串，并且不再继续找，需要注意的是 match 函数是从字符串开始处开始查找的，如果开始处不匹配，则不再继续寻找，返回值为 一个 SRE\_Match (参见 第四小节 re 内置对象用法) 对象，找不到时返回 None

[复制代码](javascript:void(0);)

s = '''first line

second line

third line'''

# compile

regex = re.compile("\w+")

m = regex.match(s)

print m

# output> <\_sre.SRE\_Match object at 0x0000000002BCA8B8>

print m.group()

# output> first

# s 的开头是 "f", 但正则中限制了开始为 i 所以找不到

regex = re.compile("^i\w+")

print regex.match(s)

# output> None

### 返回所有查找内容的位置

1. **import** re
2. str=u'小明买冰棍花了5元，买糖果花了3元，买游戏花了59元，小明今天一共花了67元。'
3. word = u'元'
4. a = [m.start() **for** m **in** re.finditer(word, str)]
5. **print** a

返回[8, 16, 25, 37]

### 截取想获取的内容

Findall返回一个查找到的list，然后使用split方法来分离想要获取的内容

p = re.compile(*'.\*Response status =.\*'*, re.I|re.M)

list = p.findall(result)

print list[0].split(*' '*)[3]

print list

### 查找最后出现的字符

Result = ”””

{

"command" : "get-property",

"response" : [ 1258422784 ],

"status" : {

"description" : "0 (0x0) Success.",

"value" : 0

}

}”””

a = result[result.find('{') : result.rfind('}')]

rfind代表查找最后出现的’}’

这段代码的意思是result中截取最先出现的’{’ 到最后出现的’}’