

Py西游攻关之模块

模块3包(*****)

模块(module)的概念:

在计算机程序的开发过程中,随着程序代码越写越多,在一个文件里代码就会越来越长,越来越不容易维护。

为了编写可维护的代码,我们把很多函数分组,分别放到不同的文件里,这样,每个文件包含的代码就相对较少,很多编程语言都采用这种组织代码的方式。在Python中,一个.py文件就称之为一个模块(Module)。

使用模块有什么好处?

最大的好处是大大提高了代码的可维护性。

其次,编写代码不必从零开始。当一个模块编写完毕,就可以被其他地方引用。我们在编写程序的时候,也经常引用其他模块,包括Python内置的模块和来自第三方的模块。

所以,模块一共三种:

- python标准库
- 第三方模块
- 应用程序自定义模块

另外,使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中,因此,我们自己在编写模块时,不必考虑名字会与其他模块冲突。但是也要注意,尽量不要与内置函数名字冲突。

模块导入方法

1 import 语句

```
1 | import module1[, module2[, ... moduleN]]
```

当我们使用import语句的时候,Python解释器是怎样找到对应的文件的呢?答案就是解释器有自己的搜索路径,存在sys.path里。

```
1 | [' ', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-x86_64-linux-gnu',  
2 | '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.4/dist-packages', '/usr/lib/python3/dis'
```

因此若像我一样在当前目录下存在与要引入模块同名的文件,就会把要引入的模块屏蔽掉。

2 from...import 语句

```
1 | from modname import name1[, name2[, ... nameN]]
```

这个声明不会把整个modulename模块导入到当前的命名空间中,只会将它里面的name1或name2单个引入到执行这个声明的模块的全局符号表。

3 From...import* 语句

```
1 | from modname import *
```

这提供了一个简单的方法来导入一个模块中的所有项目。然而这种声明不该被过多地使用。大多数情况,Python程序员不使用这种方法,因为引入的其它来源的命名,很可能覆盖了已有的定义。

4 运行本质

```
1 | #1 import test  
2 | #2 from test import add
```

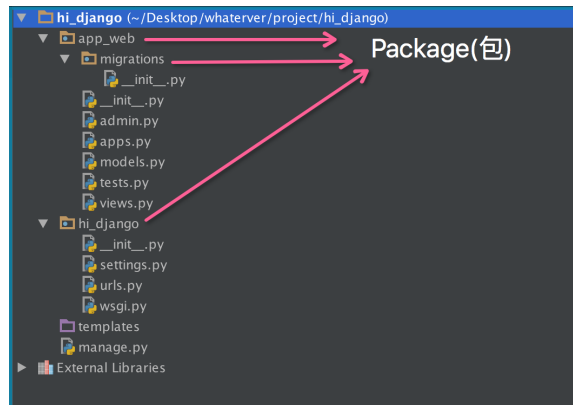
无论1还是2,首先通过sys.path找到test.py,然后执行test脚本(全部执行),区别是1会将test这个变量名加载到名字空间,而2只会将add这个变量名加载进来。

包(package)

如果不同的人编写的模块名相同怎么办？为了避免模块名冲突，Python又引入了按目录来组织模块的方法，称为包（Package）。

举个例子，一个abc.py的文件就是一个名字叫abc的模块，一个xyz.py的文件就是一个名字叫xyz的模块。

现在，假设我们的abc和xyz这两个模块名字与其他模块冲突了，于是我们可以通过包来组织模块，避免冲突。方法是选择一个顶层包名：

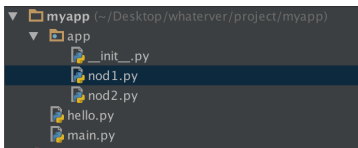


引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在，view.py模块的名字就变成了hello_django.app01.views，类似的，manage.py的模块名则是hello_django.manage。

请注意，每一个包目录下面都会有一个__init__.py的文件，这个文件是必须存在的，否则，Python就把这个目录当成普通目录(文件夹)，而不是一个包。__init__.py可以是空文件，也可以有Python代码，因为__init__.py本身就是一个模块，而它的模块名就是对应包的名字。

调用包就是执行包下的__init__.py文件

注意点 (important)



1-----

在nod1里import hello是找不到的，有同学说可以找到呀，那是因为你的pycharm为你把myapp这一层路径加入到了sys.path里面，所以可以找到，然而程序一旦在命令行运行，则报错。有同学问那怎么办？简单啊，自己把这个路径加进去不就OK啦：

```
1 import sys,os
2 BASE_DIR=os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
3 sys.path.append(BASE_DIR)
4 import hello
5 hello.hello1()
```

2 -----

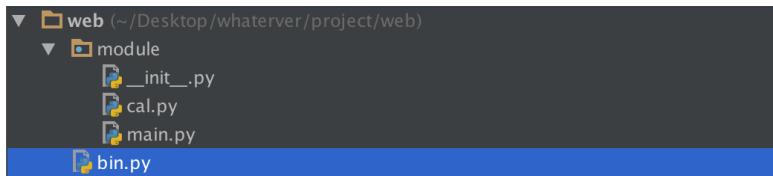
```
1 if __name__=='__main__':
2     print('ok')
```

“Make a .py both importable and executable”

如果我们直接执行某个.py文件的时候，该文件中那么“__name__ == '__main__'”是True,但是我们如果从另外一个.py文件通过import导入该文件的时候，这时__name__的值就是我们这个.py文件的名字而不是__main__。

这个功能还有一个用处：调试代码的时候，在“if __name__ == '__main__'”中加入一些我们的调试代码，我们可以让外部模块调用的时候不执行我们的调试代码，但是如果我们要想排查问题的时候，直接执行该模块文件，调试代码能够正常运行！s

3



```

1  ##-----cal.py
2  def add(x,y):
3
4      return x+y
5  ##-----main.py
6  import cal      #from module import cal
7
8  def main():
9
10     cal.add(1,2)
11
12  ##-----bin.py
13  from module import main
14
15  main.main()

```

注意

time模块(***))

三种时间表示

在Python中，通常有这几种方式来表示时间：

- 时间戳(timestamp)：通常来说，时间戳表示的是从1970年1月1日00:00:00开始按秒计算的偏移量。我们运行“type(time.time())”，返回的是float类型。
- 格式化的时间字符串
- 元组(struct_time)：struct_time元组共有9个元素共九个元素:(年，月，日，时，分，秒，一年中第几周，一年中第几天，夏令时)

```

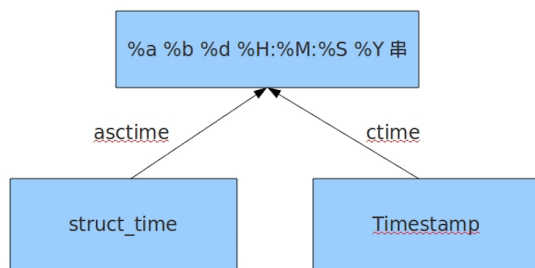
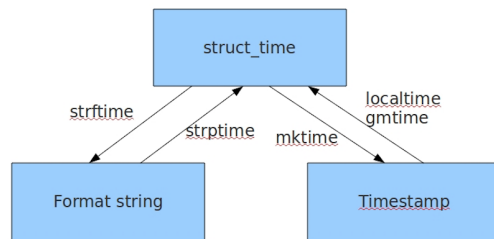
1  import time
2
3  # 1 time() :返回当前时间的时间戳
4  time.time() #1473525444.037215
5
6  #-----
7
8  # 2 localtime([secs])
9  # 将一个时间戳转换为当前时区的struct_time。secs参数未提供，则以当前时间为准。
10 time.localtime() #time.struct_time(tm_year=2016, tm_mon=9, tm_mday=11, tm_hour=0,
11 # tm_min=38, tm_sec=39, tm_wday=6, tm_yday=255, tm_isdst=0)
12 time.localtime(1473525444.037215)
13
14 #-----
15
16 # 3 gmtime([secs]) 和localtime()方法类似，gmtime()方法是将一个时间戳转换为UTC时区（0时区）的struct_tim
17
18 #-----
19
20 # 4 mktime(t)：将一个struct_time转化为时间戳。
21 print(time.mktime(time.localtime()))#1473525749.0
22
23 #-----
24
25 # 5 asctime([t])：把一个表示时间的元组或者struct_time表示为这种形式：'Sun Jun 20 23:21:05 1993'。
26 # 如果没有参数，将会将time.localtime()作为参数传入。
27 print(time.asctime())#Sun Sep 11 00:43:43 2016
28
29 #-----
30

```

```

31 # 6 ctime([secs]) : 把一个时间戳（按秒计算的浮点数）转化为time.asctime()的形式。如果参数未给或者为
32 # None的时候，将会默认time.time()为参数。它的作用相当于time.asctime(time.localtime(secs))。
33 print(time.ctime()) # Sun Sep 11 00:46:38 2016
34
35 print(time.ctime(time.time())) # Sun Sep 11 00:46:38 2016
36
37 # 7 strftime(format[, t]) : 把一个代表时间的元组或者struct_time（如由time.localtime()和
38 # time.gmtime()返回）转化为格式化的时间字符串。如果t未指定，将传入time.localtime()。如果元组中任何一个
39 # 元素越界，ValueError的错误将会被抛出。
40 print(time.strftime("%Y-%m-%d %X", time.localtime()))#2016-09-11 00:49:56
41
42 # 8 time.strptime(string[, format])
43 # 把一个格式化时间字符串转化为struct_time。实际上它和strftime()是逆操作。
44 print(time.strptime('2011-05-05 16:37:06', '%Y-%m-%d %X'))
45
46 #time.struct_time(tm_year=2011, tm_mon=5, tm_mday=5, tm_hour=16, tm_min=37, tm_sec=6,
47 # tm_wday=3, tm_yday=125, tm_isdst=-1)
48
49 #在这个函数中，format默认为：“%a %b %d %H:%M:%S %Y”。
50
51
52 # 9 sleep(secs)
53 # 线程推迟指定的时间运行，单位为秒。
54
55 # 10 clock()
56 # 这个需要注意，在不同的系统上含义不同。在UNIX系统上，它返回的是“进程时间”，它是用秒表示的浮点数（时间戳）
57 # 而在WINDOWS中，第一次调用，返回的是进程运行的实际时间。而第二次之后的调用是自第一次调用以后到现在的运行
58 # 时间，即两次时间差。

```



```

1 help(time)
2 help(time.asctime)

```

random模块**

```

1 import random
2
3 print(random.random())#(0,1)---float
4
5 print(random.randint(1,3)) #[1,3]
6
7 print(random.randrange(1,3)) #[1,3]
8
9 print(random.choice([1, '23', [4,5]]))#23
10

```

```

11 print(random.sample([1, '23', [4, 5]], 2))#[[4, 5], '23']
12
13 print(random.uniform(1, 3))#1.927109612082716
14
15
16 item=[1, 3, 5, 7, 9]
17 random.shuffle(item)
18 print(item)

```

田

验证码

os模块(****)

os模块是与操作系统交互的一个接口

田

```

os.getcwd() 获取当前工作目录，即当前python脚本工作的目录路径
os.chdir("dirname") 改变当前脚本工作目录；相当于shell下cd
os.curdir 返回当前目录：('.')
os.pardir 获取当前目录的父目录字符串名：('..')
os.makedirs('dirname1/dirname2') 可生成多层递归目录
os.removedirs('dirname1') 若目录为空，则删除，并递归到上一级目录，如若也为空，则删除，依此类推
os.mkdir('dirname') 生成单级目录；相当于shell中mkdir dirname
os.rmdir('dirname') 删除单级空目录，若目录不为空则无法删除，报错；相当于shell中rmdir dirname
os.listdir('dirname') 列出指定目录下的所有文件和子目录，包括隐藏文件，并以列表方式打印
os.remove() 删除一个文件
os.rename("oldname", "newname") 重命名文件/目录
os.stat('path/filename') 获取文件/目录信息
os.sep 输出操作系统特定的路径分隔符，win下为"\", Linux下为"/"
os.linesep 输出当前平台使用的行终止符，win下为"\t\n", Linux下为"\n"
os.pathsep 输出用于分割文件路径的字符串 win下为;, Linux下为:
os.name 输出字符串指示当前使用平台。win->'nt'; Linux->'posix'
os.system("bash command") 运行shell命令，直接显示
os.environ 获取系统环境变量
os.path.abspath(path) 返回path规范化的绝对路径
os.path.split(path) 将path分割成目录和文件名二元组返回
os.path.dirname(path) 返回path的目录。其实就是os.path.split(path)的第一个元素
os.path.basename(path) 返回path最后的文件名。如何path以/或\结尾，那么就会返回空值。即os.path.split(path)
os.path.exists(path) 如果path存在，返回True；如果path不存在，返回False
os.path.isabs(path) 如果path是绝对路径，返回True
os.path.isfile(path) 如果path是一个存在的文件，返回True。否则返回False
os.path.isdir(path) 如果path是一个存在的目录，则返回True。否则返回False
os.path.join(path1[, path2[, ...]]) 将多个路径组合后返回，第一个绝对路径之前的参数将被忽略
os.path.getatime(path) 返回path所指向的文件或者目录的最后存取时间
os.path.getmtime(path) 返回path所指向的文件或者目录的最后修改时间

```

田

sys模块(****)

1	sys.argv	命令行参数List，第一个元素是程序本身路径
2	sys.exit(n)	退出程序，正常退出时exit(0)
3	sys.version	获取Python解释程序的版本信息
4	sys.maxint	最大的Int值
5	sys.path	返回模块的搜索路径，初始化时使用PYTHONPATH环境变量的值
6	sys.platform	返回操作系统平台名称

进度条：

田

```

import sys,time
for i in range(10):
    sys.stdout.write('#')
    time.sleep(1)
    sys.stdout.flush()

```

田

json & pickle(****)

之前我们学习过用eval内置方法可以将一个字符串转成python对象，不过，eval方法是有局限性的，对于普通的数据类型，json.loads和eval都能用，但遇到特殊类型的时候，eval就不管用了，所以eval的重点还是通常用来执行

一个字符串表达式，并返回表达式的值。

```
1 import json
2 x="[null,true,false,1]"
3 print(eval(x))
4 print(json.loads(x))
```

什么是序列化?

我们把对象(变量)从内存中变成可存储或传输的过程称之为序列化，在Python中叫pickling，在其他语言中也被称之为serialization, marshalling, flattening等等，都是一个意思。

序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即unpickling。

json

如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如XML，但更好的方法是序列化为JSON，因为JSON表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。JSON不仅是标准格式，并且比XML更快，而且可以直接在Web页面中读取，非常方便。

JSON表示的对象就是标准的JavaScript语言的对象，JSON和Python内置的数据类型对应如下：

JSON类型	Python类型
{}	dict
[]	list
"string"	str
1234.56	int或float
true/false	True/False
null	None

```
1  #-----序列化
2  import json
3
4  dic={'name':'alvin','age':23,'sex':'male'}
5  print(type(dic))<class 'dict'>
6
7  j=json.dumps(dic)
8  print(type(j))<class 'str'>
9
10
11 f=open('序列化对象','w')
12 f.write(j) #-----等价于json.dump(dic,f)
13 f.close()
14 #-----反序列化<br>
15 import json
16 f=open('序列化对象')
17 data=json.loads(f.read())# 等价于data=json.load(f)
```

⊕ 注意点

pickle

```
1  ##-----序列化
2  import pickle
3
4  dic={'name':'alvin','age':23,'sex':'male'}
5
6  print(type(dic))<class 'dict'>
7
8  j=pickle.dumps(dic)
9  print(type(j))<class 'bytes'>
10
11
12 f=open('序列化对象_pickle','wb')#注意w是写入str,wb是写入bytes,j是'bytes'
13 f.write(j) #-----等价于pickle.dump(dic,f)
14
```

```

15 f.close()
16 #-----反序列化
17 import pickle
18 f=open('序列化对象_pickle','rb')
19
20 data=pickle.loads(f.read())# 等价于data=pickle.load(f)
21
22
23 print(data['age'])

```

Pickle的问题和所有其他编程语言特有的序列化问题一样，就是它只能用于Python，并且可能不同版本的Python彼此都不兼容，因此，只能用Pickle保存那些不重要的数据，不能成功地反序列化也没关系。

shelve模块(***)

shelve模块比pickle模块简单，只有一个open函数，返回类似字典的对象，可读可写;key必须为字符串，而值可以是python所支持的数据类型

```

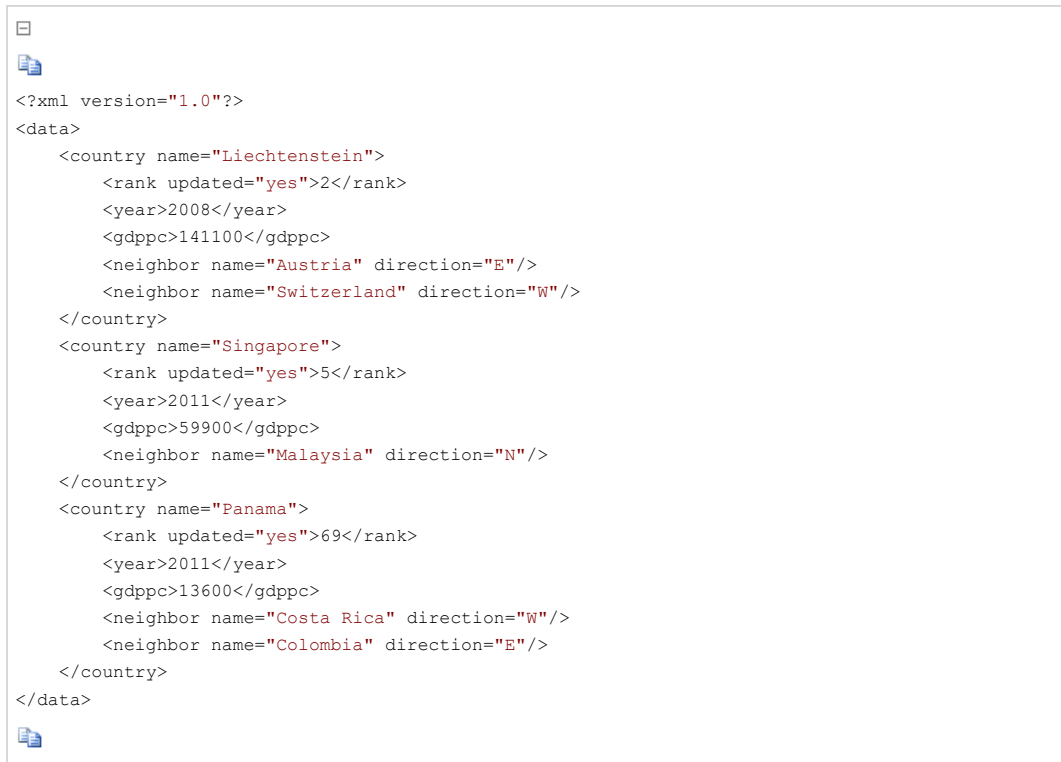
1 import shelve
2
3 f = shelve.open(r'shelve.txt')
4
5 # f['stu1_info']={'name':'alex','age':'18'}
6 # f['stu2_info']={'name':'alvin','age':'20'}
7 # f['school_info']={'website':'oldboyedu.com','city':'beijing'}
8 #
9 #
10 # f.close()
11
12 print(f.get('stu_info')['age'])

```

xml模块(***)

xml是实现不同语言或程序之间进行数据交换的协议，跟json差不多，但json使用起来更简单，不过，古时候，在json还没诞生的黑暗年代，大家只能选择用xml呀，至今很多传统公司如金融行业的很多系统的接口还主要是xml。

xml的格式如下，就是通过<>节点来区别数据结构的:



```

<?xml version="1.0"?>
<data>
  <country name="Liechtenstein">
    <rank updated="yes">2</rank>
    <year>2008</year>
    <gdppc>141100</gdppc>
    <neighbor name="Austria" direction="E"/>
    <neighbor name="Switzerland" direction="W"/>
  </country>
  <country name="Singapore">
    <rank updated="yes">5</rank>
    <year>2011</year>
    <gdppc>59900</gdppc>
    <neighbor name="Malaysia" direction="N"/>
  </country>
  <country name="Panama">
    <rank updated="yes">69</rank>
    <year>2011</year>
    <gdppc>13600</gdppc>
    <neighbor name="Costa Rica" direction="W"/>
    <neighbor name="Colombia" direction="E"/>
  </country>
</data>

```

xml协议在各个语言里的都是支持的，在python中可以用以下模块操作xml:

[View Code](#)

自己创建xml文档:

+

创建xml文档

configparser模块(🌟🌟)

来看一个好多软件的常见文档格式如下:

```

1  [DEFAULT]
2  ServerAliveInterval = 45
3  Compression = yes
4  CompressionLevel = 9
5  ForwardX11 = yes
6
7  [bitbucket.org]
8  User = hg
9
10 [topsecret.server.com]
11 Port = 50022
12 ForwardX11 = no

```

如果想用python生成一个这样的文档怎么做呢?

```

1  import configparser
2
3  config = configparser.ConfigParser()
4  config["DEFAULT"] = {'ServerAliveInterval': '45',
5                      'Compression': 'yes',
6                      'CompressionLevel': '9'}
7
8  config['bitbucket.org'] = {}
9  config['bitbucket.org']['User'] = 'hg'
10 config['topsecret.server.com'] = {}
11 topsecret = config['topsecret.server.com']
12 topsecret['Host Port'] = '50022' # mutates the parser
13 topsecret['ForwardX11'] = 'no' # same here
14 config['DEFAULT']['ForwardX11'] = 'yes'<br>
15 with open('example.ini', 'w') as configfile:
16     config.write(configfile)

```

+

增删改查

hashlib模块(🌟🌟)

用于加密相关的操作, 3.x里代替了md5模块和sha模块, 主要提供 SHA1, SHA224, SHA256, SHA384, SHA512, MD5 算法

```

1  import hashlib
2
3  m=hashlib.md5()# m=hashlib.sha256()
4
5  m.update('hello'.encode('utf8'))
6  print(m.hexdigest()) #5d41402abc4b2a76b9719d911017c592
7
8  m.update('alvin'.encode('utf8'))
9
10 print(m.hexdigest()) #92a7e713c30abb0319fa07da2a5c4af
11
12 m2=hashlib.md5()
13 m2.update('helloalvin'.encode('utf8'))
14 print(m2.hexdigest()) #92a7e713c30abb0319fa07da2a5c4af

```

以上加密算法虽然依然非常厉害, 但时候存在缺陷, 即: 通过撞库可以反解。所以, 有必要对加密算法中添加自定义key再来做加密。

```

1  import hashlib
2
3  # ##### 256 #####
4
5  hash = hashlib.sha256('8980aFs09f'.encode('utf8'))

```



```

6 | hash.update('alvin'.encode('utf8'))
7 | print (hash.hexdigest())#e79e68f070cdedcfe63eaf1a2e92c83b4cfb1b5c6bc452d214c1b7e77cdfd1c7

```

python 还有一个 hmac 模块，它内部对我们创建 key 和 内容 再进行处理然后再加密:

```

1 | import hmac
2 | h = hmac.new('alvin'.encode('utf8'))
3 | h.update('hello'.encode('utf8'))
4 | print (h.hexdigest())#320df9832eab4c038b6c1d7ed73a5940

```

subprocess模块(***)

当我们需要调用系统的命令的时候，最先考虑的os模块。用os.system()和os.popen()来进行操作。但是这两个命令过于简单，不能完成一些复杂的操作，如给运行的命令提供输入或者读取命令的输出，判断该命令的运行状态，管理多个命令的并行等等。这时subprocess中的Popen命令就能有效的完成我们需要的操作。

subprocess模块允许一个进程创建一个新的子进程，通过管道连接到子进程的stdin/stdout/stderr，获取子进程的返回值等操作。

The subprocess module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.

This module intends to replace several other, older modules and functions, such as: os.system、os.spawn*、os.popen*、popen2.*、commands.*

这个模块一个类：Popen。

```

1 | #Popen它的构造函数如下:
2 |
3 | subprocess.Popen(args, bufsize=0, executable=None, stdin=None, stdout=None, stderr=None, preexec_fi

```



简单命令:

```

1 | import subprocess
2 |
3 | a=subprocess.Popen('ls')# 创建一个新的进程,与主进程不同步
4 |
5 | print('>>>>>>',a)#a是Popen的一个实例对象
6 |
7 | '''
8 | >>>>>> <subprocess.Popen object at 0x10185f860>
9 | __init__.py
10 | __pycache__
11 | log.py
12 | main.py
13 |
14 | '''
15 |
16 | # subprocess.Popen('ls -l',shell=True)
17 |
18 | # subprocess.Popen(['ls', '-l'])

```

subprocess.PIPE

在创建Popen对象时，subprocess.PIPE可以初始化stdin, stdout或stderr参数。表示与子进程通信的标准流。

```

1 | import subprocess
2 |
3 | # subprocess.Popen('ls')
4 | p=subprocess.Popen('ls',stdout=subprocess.PIPE)#结果跑哪去啦?
5 |
6 | print(p.stdout.read())#这这呢:b'__pycache__\nhello.py\nok.py\nweb\n'

```

这是因为subprocess创建了子进程，结果本在子进程中，if 想要执行结果转到主进程中，就得需要一个管道，即：
stdout=subprocess.PIPE

subprocess.STDOUT

创建Popen对象时，用于初始化stderr参数，表示将错误通过标准输出流输出。

Popen的方法

 [View Code](#)

subprocess模块的工具函数

```

1 subprocess模块提供了一些函数，方便我们用于创建进程来实现一些简单的功能。
2
3 subprocess.call(*popenargs, **kwargs)
4 运行命令。该函数将一直等待到子进程运行结束，并返回进程的returncode。如果子进程不需要进行交互,就可以使用该
5
6 subprocess.check_call(*popenargs, **kwargs)
7 与subprocess.call(*popenargs, **kwargs)功能一样，只是如果子进程返回的returncode不为0的话，将触发CalledProcessError
8
9 check_output(*popenargs, **kwargs)
10 与call()方法类似，以byte string的方式返回子进程的输出，如果子进程的返回值不是0，它抛出CalledProcessError
11
12 getstatusoutput(cmd)/getoutput(cmd)
13 这两个函数仅仅在Unix下可用，它们在shell中执行指定的命令cmd，前者返回(status, output)，后者返回output。其

```

 [演示](#)

交互命令：

终端输入的命令分为两种：

- 输入即可得到输出，如：ifconfig
- 输入进行某环境，依赖再输入，如：python

需要交互的命令示例

待续

logging模块(****)

一 (简单应用)

```

import logging
logging.debug('debug message')
logging.info('info message')
logging.warning('warning message')
logging.error('error message')
logging.critical('critical message')

```

输出：

```


WARNING:root:warning message
ERROR:root:error message
CRITICAL:root:critical message

```

可见，默认情况下Python的logging模块将日志打印到了标准输出中，且只显示了大于等于WARNING级别的日志，这说明默认的日志级别设置为WARNING（日志级别等级CRITICAL > ERROR > WARNING > INFO > DEBUG > NOTSET），默认的日志格式为日志级别：Logger名称：用户输出消息。

二 灵活配置日志级别，日志格式，输出位置

```


import logging
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(filename)s[line:%(lineno)d] %(levelname)s %(message)s',
                    datefmt='%a, %d %b %Y %H:%M:%S',
                    filename='/tmp/test.log',
                    filemode='w')

logging.debug('debug message')
logging.info('info message')
logging.warning('warning message')

```

```
logging.error('error message')
logging.critical('critical message')
```



查看输出：

```
cat /tmp/test.log
```

```
Mon, 05 May 2014 16:29:53 test_logging.py[line:9] DEBUG debug message
Mon, 05 May 2014 16:29:53 test_logging.py[line:10] INFO info message
Mon, 05 May 2014 16:29:53 test_logging.py[line:11] WARNING warning message
Mon, 05 May 2014 16:29:53 test_logging.py[line:12] ERROR error message
Mon, 05 May 2014 16:29:53 test_logging.py[line:13] CRITICAL critical message
```

可见在logging.basicConfig()函数中可通过具体参数来更改logging模块默认行为，可用参数有
filename：用指定的文件名创建FileHandler（后边会具体讲解handler的概念），这样日志会被存储在指定的文件中。

filemode：文件打开方式，在指定了filename时使用这个参数，默认值为“a”还可指定为“w”。

format：指定handler使用的日志显示格式。

datefmt：指定日期时间格式。

level：设置rootlogger（后边会讲解具体概念）的日志级别

stream：用指定的stream创建StreamHandler。可以指定输出到sys.stderr,sys.stdout或者文件（f=open('test.log','w')），默认为sys.stderr。若同时列出了filename和stream两个参数，则stream参数会被忽略。

format参数中可能用到的格式化串：

```
%(name)s Logger的名字
%(levelno)s 数字形式的日志级别
%(levelname)s 文本形式的日志级别
%(pathname)s 调用日志输出函数的模块的完整路径名，可能没有
%(filename)s 调用日志输出函数的模块的文件名
%(module)s 调用日志输出函数的模块名
%(funcName)s 调用日志输出函数的函数名
%(lineno)d 调用日志输出函数的语句所在的代码行
%(created)f 当前时间，用UNIX标准的表示时间的浮点数表示
%(relativeCreated)d 输出日志信息时的，自Logger创建以来的毫秒数
%(asctime)s 字符串形式的当前时间。默认格式是“2003-07-08 16:49:45,896”。逗号后面的是毫秒
%(thread)d 线程ID。可能没有
%(threadName)s 线程名。可能没有
%(process)d 进程ID。可能没有
%(message)s 用户输出的消息
```

三 logger对象

上述几个例子中我们了解到了logging.debug()、logging.info()、logging.warning()、logging.error()、logging.critical()（分别用以记录不同级别的日志信息），logging.basicConfig()（用默认日志格式（Formatter）为日志系统建立一个默认的流处理器（StreamHandler），设置基础配置（如日志级别等）并加到root logger（根Logger）中）这几个logging模块级别的函数，另外还有一个模块级别的函数是logging.getLogger([name])（返回一个logger对象，如果没有指定名字将返回root logger）

先看一个最简单的过程：



```
import logging

logger = logging.getLogger()
# 创建一个handler，用于写入日志文件
fh = logging.FileHandler('test.log')

# 再创建一个handler，用于输出到控制台
ch = logging.StreamHandler()

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

fh.setFormatter(formatter)
ch.setFormatter(formatter)
```

```

logger.addHandler(fh) #logger对象可以添加多个fh和ch对象
logger.addHandler(ch)

logger.debug('logger debug message')
logger.info('logger info message')
logger.warning('logger warning message')
logger.error('logger error message')
logger.critical('logger critical message')

```



先简单介绍一下，logging库提供了多个组件：Logger、Handler、Filter、Formatter。Logger对象提供应用程序可直接使用的接口，Handler发送日志到适当的目的地，Filter提供了过滤日志信息的方法，Formatter指定日志显示格式。

(1)

Logger是一个树形层级结构，输出信息之前都要获得一个Logger（如果没有显示的获取则自动创建并使用root Logger，如第一个例子所示）。

logger = logging.getLogger()返回一个默认的Logger也即root Logger，并应用默认的日志级别、Handler和Formatter设置。

当然也可以通过Logger.setLevel(lvl)指定最低的日志级别，可用的日志级别有logging.DEBUG、logging.INFO、logging.WARNING、logging.ERROR、logging.CRITICAL。

Logger.debug()、Logger.info()、Logger.warning()、Logger.error()、Logger.critical()输出不同级别的日志，只有日志等级大于或等于设置的日志级别的日志才会被输出。

```

logger.debug('logger debug message')
logger.info('logger info message')
logger.warning('logger warning message')
logger.error('logger error message')
logger.critical('logger critical message')

```

只输出了

2014-05-06 12:54:43,222 - root - WARNING - logger warning message

2014-05-06 12:54:43,223 - root - ERROR - logger error message

2014-05-06 12:54:43,224 - root - CRITICAL - logger critical message

从这个输出可以看出 logger = logging.getLogger() 返回的 Logger 名为 root。这里没有用 `logger.setLevel(logging.Debug)` 显示的为 logger 设置日志级别，所以使用默认的日志级别 WARNING，故结果只输出了大于等于 WARNING 级别的信息。

(2) 如果我们再创建两个 logger 对象：

```

#####
logger1 = logging.getLogger('mylogger')
logger1.setLevel(logging.DEBUG)

logger2 = logging.getLogger('mylogger')
logger2.setLevel(logging.INFO)

logger1.addHandler(fh)
logger1.addHandler(ch)

logger2.addHandler(fh)
logger2.addHandler(ch)

logger1.debug('logger1 debug message')
logger1.info('logger1 info message')
logger1.warning('logger1 warning message')
logger1.error('logger1 error message')
logger1.critical('logger1 critical message')

logger2.debug('logger2 debug message')
logger2.info('logger2 info message')
logger2.warning('logger2 warning message')
logger2.error('logger2 error message')
logger2.critical('logger2 critical message')

```



结果：

```

/Library/Frameworks/Python.framework/Versions/3.4/bin/python3.4 /Users/yuanf
2016-08-03 13:06:15,816 - root - DEBUG - logger debug message
2016-08-03 13:06:15,818 - root - INFO - logger info message
2016-08-03 13:06:15,819 - root - WARNING - logger warning message
2016-08-03 13:06:15,819 - root - ERROR - logger error message
2016-08-03 13:06:15,819 - root - CRITICAL - logger critical message
2016-08-03 13:06:15,820 - mylogger - INFO - logger1 info message
2016-08-03 13:06:15,820 - mylogger - INFO - logger1 info message
2016-08-03 13:06:15,820 - mylogger - WARNING - logger1 warning message
2016-08-03 13:06:15,820 - mylogger - WARNING - logger1 warning message
2016-08-03 13:06:15,821 - mylogger - ERROR - logger1 error message
2016-08-03 13:06:15,821 - mylogger - ERROR - logger1 error message
2016-08-03 13:06:15,821 - mylogger - CRITICAL - logger1 critical message
2016-08-03 13:06:15,821 - mylogger - CRITICAL - logger1 critical message
2016-08-03 13:06:15,822 - mylogger - INFO - logger2 info message
2016-08-03 13:06:15,822 - mylogger - INFO - logger2 info message
2016-08-03 13:06:15,822 - mylogger - WARNING - logger2 warning message
2016-08-03 13:06:15,822 - mylogger - WARNING - logger2 warning message
2016-08-03 13:06:15,822 - mylogger - ERROR - logger2 error message
2016-08-03 13:06:15,822 - mylogger - ERROR - logger2 error message
2016-08-03 13:06:15,823 - mylogger - CRITICAL - logger2 critical message
2016-08-03 13:06:15,823 - mylogger - CRITICAL - logger2 critical message

```

这里有两个问题：

<1>我们明明通过`logger1.setLevel(logging.DEBUG)`将`logger1`的日志级别设置为了`DEBUG`，为何显示的时候没有显示出`DEBUG`级别的日志信息，而是从`INFO`级别的日志开始显示呢？

原来`logger1`和`logger2`对应的是同一个`Logger`实例，只要`logging.getLogger(name)`中名称参数`name`相同则返回的`Logger`实例就是同一个，且仅有一个，也即`name`与`Logger`实例一一对应。在`logger2`实例中通过`logger2.setLevel(logging.INFO)`设置`mylogger`的日志级别为`logging.INFO`，所以最后`logger1`的输出遵从了后来设置的日志级别。

<2>为什么`logger1`、`logger2`对应的每个输出分别显示两次？

这是因为我们通过`logger = logging.getLogger()`显示的创建了`root Logger`，而`logger1 = logging.getLogger('mylogger')`创建了`root Logger`的孩子(`root.mylogger`),`logger2`同样。而孩子,孙子，重孙.....既会将消息分发给他的`handler`进行处理也会传递给所有的祖先`Logger`处理。

ok,那么现在我们把

```
# logger.addHandler(fh)
```

`# logger.addHandler(ch)` 注释掉，我们再来看效果：

```

/Library/Frameworks/Python.framework/Versions/3.4/bin/python3.4 /Users/yuanhao/Desk
logger warning message
logger error message
logger critical message
2016-08-03 13:15:17,457 - mylogger - INFO - logger1 info message
2016-08-03 13:15:17,458 - mylogger - WARNING - logger1 warning message
2016-08-03 13:15:17,458 - mylogger - ERROR - logger1 error message
2016-08-03 13:15:17,458 - mylogger - CRITICAL - logger1 critical message
2016-08-03 13:15:17,458 - mylogger - INFO - logger2 info message
2016-08-03 13:15:17,458 - mylogger - WARNING - logger2 warning message
2016-08-03 13:15:17,458 - mylogger - ERROR - logger2 error message
2016-08-03 13:15:17,458 - mylogger - CRITICAL - logger2 critical message

Process finished with exit code 0

```

因为我们注释了`logger`对象显示的位置，所以才用了默认方式，即标准输出方式。因为它的父级没有设置文件显示方式，所以在这里只打印了一次。

孩子,孙子，重孙.....可逐层继承来自祖先的日志级别、`Handler`、`Filter`设置，也可以通过`Logger.setLevel(lvl)`、`Logger.addHandler(hdlr)`、`Logger.removeHandler(hdlr)`、`Logger.addFilter(filt)`、`Logger.removeFilter(filt)`。设置自己特别的日志级别、`Handler`、`Filter`。若不设置则使用继承来的值。

<3> Filter

限制只有满足过滤规则的日志才会输出。

比如我们定义了`filter = logging.Filter('a.b.c')`并将这个`Filter`添加到了一个`Handler`上，则使用该`Handler`的`Logger`中只有名字带 `a.b.c`前缀的`Logger`才能输出其日志。

```
filter = logging.Filter('mylogger')
```

```
logger.addFilter(filter)
```

这是只对`logger`这个对象进行筛选

如果想对所有的对象进行筛选，则：

```
filter = logging.Filter('mylogger')
```

```
fh.addFilter(filter)
```

```
ch.addFilter(filter)
```

这样，所有添加fh或者ch的logger对象都会进行筛选。

完整代码1:

```
View Code
```

完整代码2:

```
View Code
```

应用:

```
View Code
```

re模块(****)

就其本质而言，正则表达式（或 RE）是一种小型的、高度专业化的编程语言，（在Python中）它内嵌在Python中，并通过 re 模块实现。正则表达式模式被编译成一系列的字节码，然后由用 C 编写的匹配引擎执行。

字符匹配（普通字符，元字符）：

1 普通字符：大多数字符和字母都会和自身匹配

```
>>> re.findall('alvin','yuanaleSxalewxupeiqi')
['alvin']
```

2 元字符：. ^ \$ * + ? { } [] | () \

元字符之. ^ \$ * + ? { }

```
1 import re
2
3 ret=re.findall('a..in','helloalvin')
4 print(ret)#[ 'alvin' ]
5
6
7 ret=re.findall('^a...n','alvinhelloawwn')
8 print(ret)#[ 'alvin' ]
9
10
11 ret=re.findall('a...n$', 'alvinhelloawwn')
12 print(ret)#[ 'awwn' ]
13
14
15 ret=re.findall('a...n$', 'alvinhelloawwn')
16 print(ret)#[ 'awwn' ]
17
18
19 ret=re.findall('abc*', 'abcccc')#贪婪匹配[0,+oo]
20 print(ret)#[ 'abcccc' ]
21
22 ret=re.findall('abc+', 'abccc')#[1,+oo]
23 print(ret)#[ 'abccc' ]
24
25 ret=re.findall('abc?', 'abccc')#[0,1]
26 print(ret)#[ 'abc' ]
27
28
29 ret=re.findall('abc{1,4}', 'abccc')
30 print(ret)#[ 'abccc' ] 贪婪匹配
```

注意：前面的*,+,?等都是贪婪匹配，也就是尽可能匹配，后面加?号使其变成惰性匹配

```
1 ret=re.findall('abc*?', 'abcccccc')
2 print(ret)#[ 'ab' ]
```

元字符之字符集 []：

```
1 #------字符集[]
2 ret=re.findall('a[bc]d', 'acd')
3 print(ret)#[ 'acd' ]
```

```

4
5 ret=re.findall('[a-z]','acd')
6 print(ret)#['a', 'c', 'd']
7
8 ret=re.findall('[. *+]', 'a.cd+')
9 print(ret)#['.', '+']
10
11 #在字符集里有功能的符号: - ^ \
12
13 ret=re.findall('[1-9]','45dha3')
14 print(ret)#['4', '5', '3']
15
16 ret=re.findall('[^ab]','45bdha3')
17 print(ret)#['4', '5', 'd', 'h', '3']
18
19 ret=re.findall('[\d]','45bdha3')
20 print(ret)#['4', '5', '3']

```

元字符之转义符\

反斜杠后边跟元字符去除特殊功能,比如\.

反斜杠后边跟普通字符实现特殊功能,比如\d

\d 匹配任何十进制数; 它相当于类 [0-9]。
 \D 匹配任何非数字字符; 它相当于类 [^0-9]。
 \s 匹配任何空白字符; 它相当于类 [\t\n\r\f\v]。
 \S 匹配任何非空白字符; 它相当于类 [^\t\n\r\f\v]。
 \w 匹配任何字母数字字符; 它相当于类 [a-zA-Z0-9_]。
 \W 匹配任何非字母数字字符; 它相当于类 [^a-zA-Z0-9_]。
 \b 匹配一个特殊字符边界, 比如空格, &, # 等

```

1 ret=re.findall('I\b','I am LIST')
2 print(ret)#[]
3 ret=re.findall(r'I\b','I am LIST')
4 print(ret)#['I']

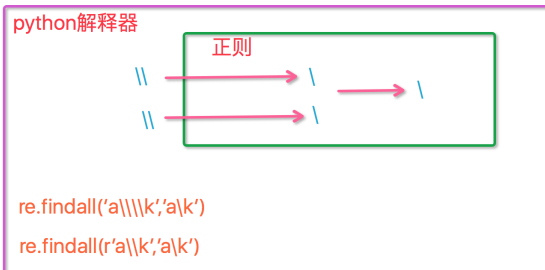
```

现在我们聊一聊,先看下面两个匹配:

```

1 #-----eg1:
2 import re
3 ret=re.findall('c\l','abc\le')
4 print(ret)#[]
5 ret=re.findall('c\\l','abc\le')
6 print(ret)#[]
7 ret=re.findall('c\\\l','abc\le')
8 print(ret)#['c\\l']
9 ret=re.findall(r'c\\l','abc\le')
10 print(ret)#['c\\l']
11
12 #-----eg2:
13 #之所以选择\b是因为\b在ASCII表中是有意义的
14 m = re.findall('\bblow', 'blow')
15 print(m)
16 m = re.findall(r'\bblow', 'blow')
17 print(m)

```



元字符之分组()

```

1 m = re.findall(r'(ad)+', 'add')

```

```

2 | print(m)
3 |
4 | ret=re.search('(P<id>\d{2})/(P<name>\w{3})', '23/com')
5 | print(ret.group())#23/com
6 | print(ret.group('id'))#23

```

元字符之 |

```

1 | ret=re.search('(ab)|\d', 'rabhdg8sd')
2 | print(ret.group())#ab

```

re模块下的常用方法

```

1 | import re
2 | #1
3 | re.findall('a', 'alvin yuan')    #返回所有满足匹配条件的结果,放在列表里
4 | #2
5 | re.search('a', 'alvin yuan').group() #函数会在字符串内查找模式匹配,只到找到第一个匹配然后返回一个包含匹
6 |                                     # 通过调用group()方法得到匹配的字符串,如果字符串没有匹配, 则返回N
7 |
8 | #3
9 | re.match('a', 'abc').group()    #同search,不过尽在字符串开始处进行匹配
10 |
11 | #4
12 | ret=re.split('[ab]', 'abcd')    #先按'a'分割得到''和'bcd',在对''和'bcd'分别按'b'分割
13 | print(ret)#['', '', 'cd']
14 |
15 | #5
16 | ret=re.sub('\d', 'abc', 'alvin5yuan6', 1)
17 | print(ret)#alvinabcyuan6
18 | ret=re.subn('\d', 'abc', 'alvin5yuan6')
19 | print(ret)#('alvinabcyuanabc', 2)
20 |
21 | #6
22 | obj=re.compile('\d{3}')
23 | ret=obj.search('abc123eeee')
24 | print(ret.group())#123

```

```

1 | import re
2 | ret=re.finditer('\d', 'ds3sy4784a')
3 | print(ret)          #<callable_iterator object at 0x10195f940>
4 |
5 | print(next(ret).group())
6 | print(next(ret).group())

```

注意:

```

1 | import re
2 |
3 | ret=re.findall('www.(baidu|oldboy).com', 'www.oldboy.com')
4 | print(ret)#['oldboy']    这是因为findall会优先把匹配结果组里内容返回,如果想要匹配结果,取消权限即可
5 |
6 | ret=re.findall('www.(?:baidu|oldboy).com', 'www.oldboy.com')
7 | print(ret)#['www.oldboy.com']

```

补充:

[View Code](#)

补充2

[View Code](#)


好文要顶

已关注

收藏该文







Yuan先生

关注 - 1

粉丝 - 3936

我在关注他 取消关注

280

posted @ 2016-08-03 13:39 Yuan先生 阅读(24470) 评论(1) 编辑 收藏

Post Comment

#1楼 2017-08-25 15:05 | microhard

回复 引用

好文章，必须要顶

支持(7) 反对(0)

刷新评论 刷新页面 返回顶部

- 【推荐】超50万行VC++源码：大型组态工控、电力仿真CAD与GIS源码库
- 【推荐】了不起的开发者，挡不住的华为，园子里的品牌专区
- 【推荐】有道智云周年庆，API服务大放送，注册即送100元体验金！
- 【推荐】开放下载！《15分钟打造你自己的小程序》（内附详细代码）



相关博文：

- [Py西游攻关之Socket网络编程](#)
 - [Py西游攻关之Socket网络编程](#)
 - [Py西游攻关之多进程\(multiprocessing模块\)](#)
 - [Py徐少攻关之基础\(3\)](#)
 - [袁老师Py西游攻关之基础数据类型](#)
- » 更多推荐...

最新 IT 新闻：

- [买Mac送AirPods，苹果想通做慈善了吗？](#)
 - [蚂蚁重塑蚂蚁](#)
 - [快手签下周杰伦，值了](#)
 - [B站推出视频剪辑工具“必剪”App](#)
 - [华为再申请区块链专利“区块链账本的存储方法及装置”](#)
- » 更多新闻...