sys 模块提供了许多函数和变量来处理 Python 运行时环境的不同部分.

**处理命令行参数**

在解释器启动后, argv 列表包含了传递给脚本的所有参数, 列表的第一个元素为脚本自身的名称.

**使用sys模块获得脚本的参数**

复制代码代码如下:

print "script name is", sys.argv[0]        # 使用sys.argv[0]采集脚本名称

if len(sys.argv) > 1:  
    print "there are", len(sys.argv)-1, "arguments:"  # 使用len(sys.argv)-1采集参数个数-1为减去[0]脚本名称  
    for arg in sys.argv[1:]:            #输出除了[0]外所有参数  
        print arg  
else:  
    print "there are no arguments!"

如果是从标准输入读入脚本 (比如 "python < sys-argv-example-1.py"), 脚本的名称将被设置为空串.

如果把脚本作为字符串传递给python (使用 -c 选项), 脚本名会被设置为 "-c".

**处理模块**

path 列表是一个由目录名构成的列表, Python 从中查找扩展模块( Python 源模块, 编译模块,或者二进制扩展).

启动 Python 时,这个列表从根据内建规则, PYTHONPATH 环境变量的内容, 以及注册表( Windows 系统)等进行初始化.

由于它只是一个普通的列表, 你可以在程序中对它进行操作,

**使用sys模块操作模块搜索路径**

复制代码代码如下:

print "path has", len(sys.path), "members"

sys.path.insert(0, "samples")  #将路径插入到path,[0]中  
import sample

sys.path = []  #删除path中所有路径  
import random

**使用sys模块查找内建模块**

builtin\_module\_names 列表包含 Python 解释器中所有内建模块的名称

复制代码代码如下:

def dump(module):  
    print module, "=>",  
    if module in sys.builtin\_module\_names:  #查找内建模块是否存在  
        print "<BUILTIN>"  
    else:  
        module = \_ \_import\_ \_(module)         #非内建模块输出模块路径  
        print module.\_ \_file\_ \_

dump("os")  
dump("sys")  
dump("string")  
dump("strop")  
dump("zlib")

os => C:\python\lib\os.pyc  
sys => <BUILTIN>  
string => C:\python\lib\string.pyc  
strop => <BUILTIN>  
zlib => C:\python\zlib.pyd

**使用sys模块查找已导入的模块**

modules 字典包含所有加载的模块. import 语句在从磁盘导入内容之前会先检查这个字典.

Python 在处理你的脚本之前就已经导入了很多模块.

复制代码代码如下:

print sys.modules.keys()

['os.path', 'os', 'exceptions', '\_ \_main\_ \_', 'ntpath', 'strop', 'nt',  
'sys', '\_ \_builtin\_ \_', 'site', 'signal', 'UserDict', 'string', 'stat']

**使用sys模块获得当前平台**

sys.platform  返回当前平台 出现如： "win32" "linux2" 等

**处理标准输出/输入**

标准输入和标准错误 (通常缩写为 stdout 和 stderr) 是内建在每一个 UNIX 系统中的管道。

当你 print 某些东西时，结果前往 stdout 管道；

当你的程序崩溃并打印出调试信息 (例如 Python 中的 traceback (错误跟踪)) 的时候，信息前往 stderr 管道

复制代码代码如下:

>>> for i in range(3):  
...     print'Dive in'

Dive in  
Dive in  
Dive in  
>>> import sys  
>>> for i in range(3):  
...     sys.stdout.write('Dive in')

Dive inDive inDive in  
>>> for i in range(3):  
...     sys.stderr.write('Dive in')

Dive inDive inDive in

stdout 是一个类文件对象；调用它的 write 函数可以打印出你给定的任何字符串。

实际上，这就是 print 函数真正做的事情；它在你打印的字符串后面加上一个硬回车，然后调用 sys.stdout.write 函数。

在最简单的例子中，stdout 和 stderr 把它们的输出发送到相同的地方

和 stdout 一样，stderr 并不为你添加硬回车；如果需要，要自己加上。

stdout 和 stderr 都是类文件对象，但是它们都是只写的。

它们都没有 read 方法，只有 write 方法。然而，它们仍然是类文件对象，因此你可以将其它任何 (类) 文件对象赋值给它们来重定向其输出。

**使用sys重定向输出**

复制代码代码如下:

print 'Dive in'        # 标准输出  
saveout = sys.stdout        # 终在重定向前保存stdout，这样的话之后你还可以将其设回正常  
fsock = open('out.log', 'w')      # 打开一个新文件用于写入。如果文件不存在，将会被创建。如果文件存在，将被覆盖。  
sys.stdout = fsock                 # 所有后续的输出都会被重定向到刚才打开的新文件上。

print  'This message will be logged instead of displayed'    # 这样只会将输出结果“打印”到日志文件中；屏幕上不会看到输出

sys.stdout = saveout   # 在我们将 stdout 搞乱之前，让我们把它设回原来的方式。

fsock.close()     # 关闭日志文件。

**重定向错误信息**

fsock = open('error.log', 'w')           # 打开你要存储调试信息的日志文件。   
sys.stderr = fsock                           # 将新打开的日志文件的文件对象赋值给stderr以重定向标准错误。  
raise Exception, 'this error will be logged'   # 引发一个异常,没有在屏幕上打印出任何东西,所有正常的跟踪信息已经写进error.log

还要注意你既没有显式关闭日志文件，也没有将 stderr 设回最初的值。

这样挺好，因为一旦程序崩溃 (由于引发的异常)，Python 将替我们清理并关闭文件

**打印到 stderr**

向标准错误写入错误信息是很常见的，所以有一种较快的语法可以立刻导出信息

复制代码代码如下:

>>> print 'entering function'  
entering function  
>>> import sys  
>>> print >> sys.stderr, 'entering function'

entering function

print 语句的快捷语法可以用于写入任何打开的文件 (或者是类文件对象)。

在这里，你可以将单个print语句重定向到stderr而且不用影响后面的print语句。

**使用sys模块退出程序**

复制代码代码如下:

import sys  
sys.exit(1)

注意 sys.exit 并不是立即退出. 而是引发一个 SystemExit 异常. 这意味着你可以在主程序中捕获对 sys.exit 的调用

**捕获sys.exit调用**

复制代码代码如下:

import sys  
print "hello"  
try:  
    sys.exit(1)  
except SystemExit:   # 捕获退出的异常  
    pass                    # 捕获后不做任何操作  
print "there"

hello  
there

如果准备在退出前自己清理一些东西(比如删除临时文件), 你可以配置一个 "退出处理函数"(exit handler), 它将在程序退出的时候自动被调用

**另一种捕获sys.exit调用的方法**

复制代码代码如下:

def exitfunc():  
    print "world"

sys.exitfunc = exitfunc  # 设置捕获时调用的函数

print "hello"  
sys.exit(1)     # 退出自动调用exitfunc()后，程序依然退出了  
print "there"  # 不会被 print

hello  
world