中软国际公司内部技术规范

Python 语言安全编程规范

中软国际科技服务有限公司

修订声明

参考:华为 Python 语言安全编码规范。

0.

日期	修订版本	修订描述	作者
2017-07-28	1.0	整理创建初稿	陈丽佳
2017-08-02	1.1	统一整理格式	陈丽佳

目录

前言

背景

《Python 语言安全编程规范》针对 Python 语言编程中的数据校验、异常行为、IO 操作、序列化和反序列化、运行环境等方面，描述可能导致安全漏洞或风险的常见编码错误。该规范旨在减少 SQL 注入、敏感信息泄露、命令注入攻击等安全问题的发生。

使用对象

本规范的读者及使用对象主要为使用 Python 语言的研发人员和测试人员。

适用范围

该规范适用于基于 Python 语言的产品开发。除非有特别说明，所有的代码示例都是基于 Python 2.7 版本。

术语定义

规则：编程时必须遵守的约定
建议：编程时必须加以考虑的约定
说明：某个规则的具体解释
错误示例：违背某条规则的例子
正确示例：遵循某条规则的例子
例外情况：相应的规则不适用的场景
信任边界：位于信任边界之内的所有组件都是被系统本身直接控制的。所有来自不受控的外部系统的连接与数据，包括客户端与第三方系统，都应该被认为是不可信的，要先在边界处对其校验，才能允许它们进一步与本系统交互。

1. 数据校验

规则 1.1 校验跨信任边界传递的不可信数据

说明：程序可能会接收来自用户、网络连接或其他来源的不可信数据，并将这些数据跨信任边界传递到目标系统（如浏览器、数据库等）。由于目标系统可能无法区分处理畸形的不可信数据，未经校验的不可信数据可能会引起某种注入攻击，对系统造成严重影响，因此，必须对不可信数据进行校验，且数据校验必须在信任边界以内进行（如对于 Web 应用，需要在服务端做校验）。数据校验有输入校验和输出校验，对从信任边界外传入的数据进行校验的叫输入校验，对传出到信任边界外的数据进行校验的叫输出校验。

如下描述了四种数据校验策略(任何时候,尽可能使用接收已知合法数据的“白名单”策略)。

接受已知好的数据

这种策略被称为“白名单”或者“正向”校验。该策略检查数据是否属于一个严格约束的、已知的、可接受的合法数据集合。例如，下面的示例代码确保 name 参数只包含字母、数字以及下划线

```
if not re.match("^[0-9A-Za-z_]+$", name):
    raise IllegalArgumentException
```

拒绝已知坏的数据

这种策略被称为“黑名单”或者“负向”校验，相对于正向校验，这是一种较弱的校验方式。由于潜在的不合法数据可能是一个不受约束的无限集合，这就意味着你必须一直维护一个已知不合法字符或者模式的列表。如果不定期研究新的攻击方式并对校验的表达式进行日常更新，该校验方式就会很快过时。

```
def removeJavascript(input):
    p = re.compile("javascript", re.IGNORECASE)
    m = p.match(input)
    return not m and input or ""
```

“白名单”方式净化

对任何不属于已验证合法字符数据中的字符进行净化，然后再使用净化后的数据，净化的方式包括删除、编码、替换。比如，如果你期望接收一个电话号码，那么你可以删除掉输入中所有的非数字字符，“(555)123-1234”，“555.123.1234”，与“555\";DROP TABLE USER;--123.1234”全部会被转换为“5551231234”，然后再对转换的结果进行校验。又比如，对于用户评论栏的文本输入，由于几乎所有的字符都可能被用到，确定一个合法的数据集合是非常困难的，一种解决方案是：将所有非字母数字替换成其编码后的版本，那么“I like your web page!”被净化后将输出为“I%20like%20your%20web%20page%21”，这里使用了 URL 编码，利用如下代码，可以方便的进行转换。

```
import urllib
print urllib.quote("I like your web page!")
```

“黑名单”方式净化

为了确保输入数据是“安全”的，可以剔除或者转换某些字符（例如，删除引号、转换成 HTML 实体）。跟“黑名单”校验类似，随着时间推移不合法字符的范围很可能不一样，需要对不合法字符进行日常维护。因此，执行一个单纯针对正确输入的“正向”校验更加简单、高效、安全。

```
def quoteApostrophe(input):
```

```
    return isinstance(input, basestring) and input.replace("'",
"&rsquo;") or None
```

规则 1.2 禁止直接使用不可信数据来拼接 SQL 语句

说明：SQL 注入是指原始 SQL 查询被动态更改成一个与程序预期完全不同的查询。执行这样一个更改后的查询可能导致信息泄露或者数据被篡改。防止 SQL 注入的方式主要可以分为两类：

- 使用参数化查询
- 对不可信数据进行校验

参数化查询是一种简单有效的防止 SQL 注入的方式，应该被优先考虑使用。另外，参数化查询还能改进数据库访问的性能，例如，SQL Server 与 Oracle 数据库会为其缓存一个查询计划，以便在多次重复执行相同的查询语句时重复使用。

使用支持参数化查询功能的数据库操作接口就能便捷有效地防止 SQL 注入攻击。Python 的 DB-API 是一个规范，它定义了一系列必须的对象和数据库存取方式，以便为各种各样的底层数据库系统和多种多样的数据库接口程序提供一致的访问接口。DB-API 规范目的在于鼓励促进所有用于访问数据库的 Python 模块相互之间的一致性。目前 Python 支持几乎所有的平台，它支持大量主要的数据库访问，当中有很多都是基于 DB-API 规范定义的，其中的 execute()函数可以支持参数传入，从库函数的层面上解决了 SQL 注入的问题。比如操作 PostgreSQL 的 psycopg2，操作 SQLite 的 sqlite3，操作 Sybase 的 Sybase module for Python，处理 ODBC 连接的 pyodbc，处理 JDBC 连接的 zxJDBC 库等，都是支持 DB-API2.0 的 Python 库。详情可以查询 Python DB-API 的官方介绍：https://www.python.org/dev/peps/pep-0249/#id41，对应的中文翻译的介绍可以参照：http://blog.csdn.net/dajianshi/article/details/7482201

错误示例：直接拼接成 Sql 语句

```
sql = "select id,type,name from xl_bugs where id = %s and type = %s" %
(id, type)
cur.execute(sql)
```

正确示例：

```
args = (id, type)
cur.execute('select id, type ,name from xl_bugs where id = %s and type
= %s', args )
```

对不可信数据进行校验的有如下两个例子：

1、对于字符型的字段进行处理时，要对输入做转义，将单引号替换为两个单引号。

错误示例：

```
def gen_sql_str(name_str):
    sql_str = "select * from stu where name ='%s'" % name_str
    print sql_str
```

如果 name_str 获取到字符串为“' or 1=1 --”，最终拼接的 SQL 语句为：

“select * from stu where name ='' or 1=1 --'”

该 SQL 语句被注入后，会导致所有的 stu 表的数据被放到结果集中

正确示例：

```
def gen_sql_str(name_str):
    #将单引号替换为两个单引号
    name_str=name_str.replace("'","''")
```

```
    sql_str = "select * from stu where name ='%s'" % name_str
    print sql_str
```

2、对于整型字段进行处理时，需要对输入进行类型检测，拒绝传入的不是合法数字。

错误示例：

```
def gen_sql_str(id_str):
    sql_str = "select * from stu where id =%s" % id_str
    print sql_str
```

如果 id_str 获取到字符串为“1 or 1=1”，最终拼接的 SQL 语句为：

“select * from stu where id =1 or 1=1”

该 SQL 语句被注入后，会导致所有的 stu 表的数据被放到结果集中。

正确示例：

```
def gen_sql_str(id_str):
    try:
        id_int = int(id_str)
        sql_str = "select * from stu where id =%d" % id_int
        print sql_str
    except ValueError, ex:
        print ex
```

对于字符型的字段，如果作为 like 的一部分。需要参考附录 A 做转义。

规则 1.3 验证路径之前应该先将其标准化

说明：绝对路径或者相对路径中可能会包含如符号（软）链接（symbolic [soft] links）、硬链接（hard links）、快捷方式（shortcuts）、影子文件（shadows）、别名（aliases）和连接文件（junctions）等形式，在进行文件验证操作之前必须完整解析这些文件链接。路径中也可能会包含如下所示的文件名，使得验证变得困难：

1. “.”指目录本身。
2. 在一个目录内，“..”指该目录的上一级目录。

除此之外，还有与特定操作系统和特定文件系统相关的命名约定，也会使验证变得困难。同一个目录或者文件，可以通过多种路径名来引用它，对其进行标准化，可以使文件路径验证较为容易。

当试图限制用户只能访问某个特定目录中的文件时，或者当基于文件名或者路径名来做安全决策时，验证是必须的。攻击者可能会利用目录遍历（directory traversal）或者等价路径（path equivalence）漏洞的方式来绕过这些限制。目录遍历漏洞使得攻击者能够转移到一个特定目录进行 I/O 操作，等价路径漏洞使得攻击者可以使用与某个资源名不同但是等价的名称来绕过安全检查。

在程序获取一个文件标准路径与打开这个文件之间，会有一个固有的时间竞争窗口。在对标准化的文件路径进行验证时，文件系统可能被修改，标准化的路径名可能不再指向原始的有效文件。值得庆幸的是，可以使用标准化的路径名来判断引用的文件名是否在安全目录中，来消除条件竞争。如果引用的文件是在一个安全目录之中，很明显攻击者无法篡改文件，也无法利用条件竞争。

错误示例：

```
try:
    print os.path.abspath("pwck")
```

```
except Exception, ex:
    print ex
```

os.path.abspath 返回文件的绝对路径，但是它不会解析文件链接。

正确示例：

```
try:
    print os.path.realpath("pwck")
except Exception, ex:
    print ex
```

这个正确示例使用了 os.path.realpath 方法，它能在所有的平台上对所有别名、快捷方式以及符号链接进行一致地解析。特殊的文件名，比如“..”会被移除，这样输入在验证之前会被简化成其标准形式。当使用标准形式的文件路径来做验证时，攻击者将无法使用../序列来跳出特定目录。

规则 1.4 禁止使用 eval 和 exec 执行不可信代码

说明：

eval 函数的声明为：eval(expression[, globals[, locals]])。它将字符串 expression 当成有效 Python 表达式来求值，并返回计算结果。

exec 语句将字符串当成有效 Python 代码来执行。提供给 exec 的代码名称空间和 exec 语句的名称空间相同。

两个函数都可以把一个字符串当做有效 Python 代码来执行。如果引入外部输入的字符串，要严格校验输入的 str 表达式的合法性。如果校验缺失，可以导致用户访问代码名称空间的变量，引起信息泄露。另外，eval 函数还可以使用__import__导入任意模块。

因此，在使用 eval 或者 exec 时，如果不允许使用名称空间中的变量，建议对代码名称空间中的变量做黑名单校验。

错误示例：

```
# example1.py
from random import randint
num = randint(1,100)
print "Guess the number! 1 to 100."
while True:
    guess = eval(raw_input('Your guess> '))
    if guess != num:
        print "Nope,", guess, "is wrong."
    else:
        print "You win!"
        exit(0)
```

下面是一些有效的 Python 表达式：

```
$ python example1.py
Guess the number! 1 to 100.
Your guess> 3*4
Nope, 12 is wrong.
Your guess> "hello"
Nope, hello is wrong.
```

下面表达式可以**绕过**程序。

```
$ python example1.py
Guess the number! 1 to 100.
Your guess> num
You win!
```

正确示例：

没有使用 eval 函数对外部输入进行表达式求值。

```
from random import randint
num = randint(1,100)
print "Guess the number! 1 to 100."
while True:
    guess = raw_input('Your guess> ')
    if guess != str(num):
        print "Nope,", guess, "is wrong."
    else:
        print "You win!"
        exit(0)
```

规则 1.5 禁止使用 input 函数

说明：input 能接收字符串，它希望能够读取一个合法的 python 表达式，并去执行它。它可以导致用户非法访问程序名称空间变量。但 raw_input()直接读取控制台的输入，并将其当做字符串保存起来。在 python 的官方手册上说明 input(x)等同于 eval(raw_input(x))，需要用户输入时，可以使用 raw_input 函数。因此不允许使用 input 函数。

错误示例：

```
Secret = "50"
value = input("Answer to everything is ? ")
print "The answer to everything is %s" % (value,)
```

在以上的代码中 input()会接受原始输入，如何这里用户传入一个 dir()再结合 print，就会执行 dir()的功能返回一个对象的**大部分**属性：

```
Answer to everything is ? dir()
The answer to everything is
[‘Secret’, '__builtins__', '__doc__', '__file__', '__name__',
'__package__']
```

我在这里看到了有一个 **Secret** 对象，然后借助原来程序的功能就可以得到该值：

```
Answer to everything is ? Secret
The answer to everything is 50
```

正确示例：

```
Secret = "50"
value = raw_input("Answer to everything is ? ")
print "The answer to everything is %s" % (value,)
```

规则 1.6 禁止调用 OS 命令解析器执行命令或运行程序， 防止命令注入

说明：使用未经校验的不可信输入作为系统命令的参数或命令的一部分，可能导致命令注入漏洞。对于命令注入漏洞，命令将会以与 Python 应用程序相同的特权级别执行，它向攻击者提供了类似系统 shell 的功能。在 Python 中，os.system 或 os.popen 经常被用来调用一个新的进程，如果被执行的命令来自于外部输入，则可能会产生命令和参数注入。命令注入相关的特殊符号的详细信息，请参考附录 B。

执行命令的时候，请注意以下几点：

1、命令执行的字符串不要去拼接输入的参数，如果必须拼接时，要对输入参数进行白名单过滤

2、对传入的参数要做类型校验，例如：整数数据，可以对数据进行整数强制转换

3、保证格式化字符串的正确性，例如：int 类型参数的拼接，对于参数要用%d，不能用%s

错误示例 1：

```
home = os.getenv('APPHOME')
cmd = home.join(INITCMD)
os.system(cmd);
```

攻击者可以通过修改环境变量 APPHOME，并且在相应目录下放置 INITCMD

错误示例 2：

```
btype = req.field('backuptype')
cmd = "cmd.exe /K \"c:\\util\\rmanDB.bat " + btype + "&&c:\\util\\cleanup.bat\""
os.system(cmd);
```

没有校验 backuptype，这个是用户输入的，攻击者可能进行攻击，例如：用户输入的是："&& del c:\\dbms*.*"。

错误示例 3：

```
try:
    import os
    print os.system("ls " + sys.argv[1])
except Exception, ex:
print ex
```

攻击者可以通过以下命令来利用这个漏洞程序：

```
python test.py ". && echo bad"
```

实际将会执行两个命令：

```
ls .
echo bad
```

正确示例 1（避免使用 os.system ()）：

```
try:
    import os
    print os.listdir(sys.argv[1])
except Exception, ex:
    print ex
```

可以使用标准的 API 替代运行系统命令来完成任务。这个正确示例使用方法 os.listdir 来列举目录下的内容，而不是调用 os.system 来运行一个外部进程，从而消除了发生命令注入与参数注入的可能。

正确示例 2（数据校验）：

```
import sys
try:
    import os
    import re
    if len(sys.argv) > 20:
        print 'Parameter length error.'
        sys.exit()
    if re.match("[\.0-9A-Za-z@]+/Z", sys.argv[1]):
        print os.system("ls " + sys.argv[1])
except Exception, ex:
    ……
```

如果无法避免使用 os.system，则必须要对输入数据进行检查和净化，要过滤的字符参考附录 B。

规 则 1.7 禁 止 使 能 SimpleXMLRPCServer 模 块 中 的 allow_dotted_names 选项

说 明： 在 版 本 2.4.1 以 前， 使 能 allow_dotted_names 选 项 是 不 安 全 的。 如 果 使 能 allow_dotted_names 选项，会允许调用者访问实例所在模块的所有变量，甚至执行任意代码。

错误示例：

```
import SimpleXMLRPCServer

class StringFunctions:
    def __init__(self):
        # Make all of the Python string functions available through
        # python_string.func_name
        import string
        self.python_string = string

    def _privateFunction(self):
        # This function cannot be called through XML-RPC because it
        # starts with an '_'
        pass

    def chop_in_half(self, astr):
        return astr[:len(astr)/2]

    def repeat(self, astr, times):
        return astr * times
```

```
server = SimpleXMLRPCServer.SimpleXMLRPCServer(("localhost", 8000))
server.register_instance(StringFunctions(), allow_dotted_names = True)
server.register_function(lambda astr: '_' + astr, '_string')
server.serve_forever()
```

以上面的代码为例：

```
>>> StringFunctions.chop_in_half.im_func.func_globals
```

运行上面的代码将会输出 StringFunctions 所在模块的所有全局变量，导致敏感信息泄露。如果你在模块导入了 os 、command 、subprocess 等模块，那么攻击者就可以调用任意命令。如果非要使能 allow_dotted_names 选项，需要在实例中定义或重载`_dispatch`方法，实现自己的 RPC 方法进行分发。

如果使用一个实例来注册 RPC 方法，需求区分两种情况：

正确示例 1：

该实例不从 SimpleXMLRPCServer 继承，需要自己实现_dispatch 方法；

```
class Math:
    def _listMethods(self):
        return ['add', 'pow']
    def _methodHelp(self, method):
        if method == 'add':
            return "add(2,3) => 5"
        elif method == 'pow':
            return "pow(x, y[, z]) => number"
        else:
            return ""
    def _dispatch(self, method, params):
        if method == 'pow':
            return pow(*params)
        elif method == 'add':
            return params[0] + params[1]
        else:
            raise 'bad method'

server = SimpleXMLRPCServer(("localhost", 8000))
server.register_introspection_functions()
server.register_instance(Math(), allow_dotted_names = True)
server.serve_forever()
```

正确示例 2：

该实例继承 SimpleXMLRPCServer 来实现自己的 XML-RPC Server，需要重载_dispatch 方法来实现自己 RPC 方法分发机制。

```
class MathServer(SimpleXMLRPCServer):
    def _dispatch(self, method, params):
        try:
            # We are forcing the 'export_' prefix on methods that are
            # callable through XML-RPC to prevent potential security
```

```
            # problems
            func = getattr(self, 'export_' + method)
        except AttributeError:
            raise Exception('method "%s" is not supported' % method)
        else:
            return func(*params)

    def export_add(self, x, y):
        return x + y

server = MathServer(("localhost", 8000))
server.serve_forever()
```

2. 异常行为

规则 2.1 不要抑制或者忽略已检查异常

说明：编码人员常常会通过一个空的或者无意义的 catch 块来抑制捕获的已检查异常。每一个 catch 块都应该确保程序只会在继续有效的情况下才会继续运行下去。因此，catch 块必须要么从异常情况中恢复，要么重新抛出适合当前 catch 块上下文的另一个异常以允许最邻近的外层 try-catch 语句块来进行恢复工作。异常会打断应用原本预期的控制流程。例如，try 块中位于异常发生点之后的任何表达式和语句都不会被执行。因此，异常必须被妥当处理。许多抑制异常的理由都是不合理的。例如，当对客户端从潜在问题恢复过来不抱期望时，一种好的做法是让异常被广播出来，而不是去捕获和抑制这个异常。

错误示例：

```
try:
    # to_do
    pass
except Exception:
    import traceback
    traceback.print_exc()
```

错误示例中，except 块只是简单地将异常堆栈轨迹打印出来。虽然打印异常的堆栈轨迹对于定位问题是有帮助的，但是最终的程序运行逻辑等同于抑制异常。注意，即使这个错误示例在发生异常时会打印一个堆栈轨迹，但是程序会继续运行，如同异常从未被抛出过。换句话说，除了 try 块中位于异常发生点之后的表达式和语句不会被执行之外，发生的异常不会影响程序的其他行为。

正确示例 1:

这个正确示例通过要求用户指定另外一个文件名来处理 FileNotFoundException 异常。

```
validFlag = False
while not validFlag:
    try:
        # If requested file does not exist, throws FileNotFoundException
        # If requested file exists, sets validFlag to true
        validFlag = True
        pass
    except FileNotFoundException:
        import traceback
        traceback.print_exc()
```

正确示例 2:

继承 Exception Reporter 并过滤敏感异常。有时候异常必须对用户隐藏。在这种情况下，一种可行的方式是继承 Exception 类，并且在重写默认的__str__方法的基础上，增加一个 filter() 方法。

```
class MyExceptionReporter(Exception):
    def __init__(self, msg):
        self.msg = msg
```

```
    def __str__(self):
        return filter(self.msg)

    def filter(self):
        # Sanitize sensitive data or replace sensitive exceptions with
non-sensitive exceptions (whitelist)
        # Return non-sensitive exception
        pass
```

例外情况：

1) 在资源释放失败不会影响程序后续行为的情况下，释放资源时发生的异常可以被抑制。释放资源的例子包括关闭文件、网络套接字、线程等等。这些资源通常是在except或者fiannly块中被释放，并且在后续的程序运行中都不会再被使用。因此，除非资源被耗尽，否则不会有其他途径使得这些异常会影响程序后续的行为。在充分处理了资源耗尽问题的情况下，只需对异常进行净化和记录日志（以备日后改进）就足够了；在这种情况下没必要做其他额外的错误处理。
2) 如果在特定的抽象层次上不可能从异常情况中恢复过来，则在那个层级的代码就不用处理这个异常，而是应该抛出一个合适的异常，让更高层次的代码去捕获处理，并尝试恢复。对于这种情况，最通常的实现方法是省略掉catch语句块，允许异常被广播出去。

规则 2.2 禁止在异常中泄露敏感信息

说明：敏感数据的范围应该基于应用场景以及产品威胁分析的结果来确定。典型的敏感数据包括口令、银行账号、个人信息、通讯记录、密钥等。如果在传递异常的时候未对其中的敏感信息进行过滤常常会导致信息泄露，而这可能帮助攻击者尝试发起进一步的攻击。攻击者可以通过构造恶意的输入参数来发掘应用的内部结构和机制。不管是异常中的文本消息，还是异常本身的类型都可能泄露敏感信息。例如，对于 IOError 异常，其中的异常消息会透露文件系统的结构信息，而通过异常本身的类型，可以得知所请求的文件不存在。因此，当异常会被传递到信任边界以外时，必须同时对敏感的异常消息和敏感的异常类型进行过滤。

错误示例:

```
import sys

if 2 != len(sys.argv):
    print ("Invalid file")
else:
    try:
        open(sys.argv[1], "r")
    except MyError as e:
        # Log the exception
```

如果传入一个文件名后，程序返回一个异常，则暗示该文件不存在，而如果不抛出异常则说明该文件是存在的。使得系统面临暴力攻击的风险，攻击者可以多次传入所有可能的文件名进行查询来发现有效文件。

正确示例:

```
import sys
```

```
if 2 != len(sys.argv):
    print ("Invalid file")
else:
    #先到路径进行标准化
    try:
        turepath = os.path.realpath(sys.argv[1])
    except Exception, ex:
        print ("Invalid file")
    #校验路径是否以/home/python/开头
    if 0 == turepath.startswith('/home/python/'):
        try:
            open(turepath)
        except IOError:
            print ("Invalid file")
    else:
        print ("Invalid file")
```

在这个正确示例中，规定用户只能打开/home/python/目录下的文件，用户不可能发现这个目录以外的任何信息。在这个方案中，如果无法打开文件，或者文件不在合法的目录下，则会产生一条简洁的错误消息。

规则 2.3 方法发生异常时要恢复到之前的对象状态

说明：当发生异常的时候，对象一般需要（关键的安全对象则必须）维持其状态的一致性。常用的可用来维持对象状态一致性的手段包括：

- 输入校验（如校验方法的调用参数）
- 调整逻辑顺序，使可能发生异常的代码在对象被修改之前执行
- 当业务操作失败时，进行回滚
- 对一个临时的副本对象进行所需的操作，直到成功完成这些操作后，才把更新提交到原始的对象
- 避免去修改对象状态

错误示例：

```
PADDING = 2
MAX_DIMENSION = 10

class Dimensions:
    def __init__(self, length, width, height):
        self.length = length
        self.width = width
        self.height = height

    def getVolumePackage(self, weight):
        self.length += PADDING
        self.width += PADDING
```

```
        self.height += PADDING
        try:
            self.validate(weight)
            volume = self.length * self.width * self.height
            self.length -= PADDING
            self.width -= PADDING
            self.height -= PADDING
            return volume
        except (Exception , ex) as e:
            return -1

    def validate(self, weight) :
        # do some validation and may throw a exception
        if weight>20:
            raise Exception
        pass
if __name__ == '__main__'
    d = Dimensions(10, 10, 10)
    print d.getVolumePackage(21)  # Prints -1 (error)
    print d.getVolumePackage(19)  # Prints 2744 instead of 1728
```

在这个错误示例中，未有异常发生时，代码逻辑会恢复对象的原始状态。但是如果出现异常事件，则回滚代码不会被执行，从而导致后续的 getVolumePackage()调用不会返回正确的结果。

正确示例 1:回滚

```
except (Exception , ex) as e:
    self.length -= PADDING
    self.width -= PADDING
    self.height -= PADDING
    return -1
```

这个正确示例在 getVolumePackage()方法的 except 块中加入了发生异常时恢复对象状态的代码。

正确示例 2:finally 子句

```
def getVolumePackage(self, weight):
    self.length += PADDING
    self.width += PADDING
    self.height += PADDING
    try:
        self.validate(weight)
        volume = self.length * self.width * self.height
        return volume
    except (Exception , ex) as e:
        return -1
    finally:
        self.length -= PADDING
```

```
        self.width -= PADDING
        self.height -= PADDING
```

这个正确示例使用一个 finally 子句来执行回滚操作，以保证不管是否发生异常，都会进行回滚。

正确示例 3:输入校验

```
def getVolumePackage(self, weight):
    try:
        self.validate(weight)
    except (Exception , ex) as e:
        return -1

    self.length += PADDING
    self.width += PADDING
    self.height += PADDING
    volume = self.length * self.width * self.height

    self.length -= PADDING
    self.width -= PADDING
    self.height -= PADDING
    return volume
```

这个正确示例在修改对象状态之前执行输入校验。注意，try 代码块中只包含可能会抛出异常的代码，而其他代码都被移到 try 块之外。

正确示例 4:未修改的对象

```
def getVolumePackage(self, weight):
    try:
        self.validate(weight)
    except (Exception , ex) as e:
        return -1

    volume = (self.length + PADDING) * (self.width + PADDING) *
(self.height + PADDING)
    return volume
```

这个正确示例避免了需要修改对象，使得对象状态不可能不一致，也因此没有必要进行回滚操作。相比之前的解决方案，更推荐使用这种方式。但是对于一些复杂的代码，这种方式可能无法实行。

3. I/O 操作

规则 3.1 禁止使用 mktemp 创建临时文件

mktemp 函数返回的临时文件名中含有进程 ID 号，这导致临时文件名很容易被猜测。攻击者可利用此漏洞执行符号链接攻击，使本地攻击者可以覆盖任意文件，导致拒绝服务。Python2.3 版本之后该函数被废除。可以使用替代函数 namedTemporaryFile()、mkstemp()和 mkdtemp()。使用新函数创建的临时文件名不再包含进程 ID，而是替代为由六个随机字符组成的一个字符串。

错误示例:

```
import tempfile
import os

tempfile = tempfile.mktemp()
```

正确示例:

```
import tempfile
import os

tempfile = tempfile.mkstemp()
```

规则 3.2 临时文件使用完毕应及时删除

说明：在全局可写的目录中创建临时文件，例如，POSIX 系统下的/tmp 与/var/tmp 目录，Windows 系统下的 C:\TEMP 目录。这类目录中的文件可能会被定期清理，例如，每天晚上或者重启时。然而，如果文件未被安全地创建或者用完后还是可访问的，具备本地文件系统访问权限的攻击者便可以利用共享目录中的文件进行恶意操作。删除已经不再需要的临时文件有助于对文件名和其他资源（如二级存储）进行回收利用。每一个程序在正常运行过程中都有责任确保删除已使用完毕的临时文件。

注意：下面的示例代码已假设文件在创建时指定了合适的访问权限，以遵循规则 3.3 在多用户系统中创建文件时指定合适的访问权限。

错误示例：

```
import os

f = open("tempnam.tmp")
if os.path.isfile(f):
    print "This file already exists"
    return

try:
    strs = "Data"
    f.write(strs)
finally:
```

```
    if f:
        try:
            f.close()
        except :
            # handle error
            pass
```

这个错误示例代码在运行结束时未将临时文件删除。

正确示例 1：

```
import tempfile
try:
    a = tempfile.NamedTemporaryFile(delete=True)
    print a.name
    a.write("abc")
finally:
    a.close()
```

这个正确示例创建临时文件时用到了 NamedTemporaryFile ()方法，这个方法会新建一个随机的文件名。文件使用 try-finally 构造块来打开，这种方式将会自动关闭文件，而不管是否有异常发生，并且在打开文件时用到了 delete 选项，使得文件在关闭时会被自动删除。

正确示例 2:手动删除临时文件

```
import os

f = os.open("tempnam.tmp", os.O_WRONLY, int("0600", 8))

try:
    strs = "Data"
    f.write(strs)
finally:
    if f:
        try:
            f.close()
        except :
            # handle error
            pass
    if os.path.isfile("tempnam.tmp"):
        os.remove("tempnam.tmp")
```

临时文件应该在使用完毕之后、系统终止之前被手动删除。

规则 3.3 在多用户系统中创建文件时指定合适的访问许可

说明：多用户系统中的文件通常归属于一个特定的用户。文件的主人能够指定系统中哪些其他用户能够访问该文件的内容。这些文件系统使用权限和许可模型来保护文件访问。当一个文件被创建时，文件访问许可规定了哪些用户可以访问或者操作这个文件。当一个程序在创建文件时没有对文件的访问许可做足够的限制，攻击者可能在程序修改此文件的访问权限之前对其进行读取或者修改。因此，一定要在创建文件时就为其指定访问许可，以防止未授权

的文件访问。

错误示例：

```
f = open("tempnam.tmp")
```

python 内置的 open 方法无法让程序员显式的指定文件的访问权限。在这个错误示例中，所创建文件的访问许可取决于具体的实现机制，可能无法防止未授权的访问。

正确示例 1：

```
import os
#使用 os 的 open 函数指定访问许可
fd = os.open("tempnam.tmp", os.O_WRONLY, int("0600", 8))
myFileObject = os.fdopen(fd)
myFileObject.write("...")
myFileObject.close()
```

正确示例 2：

```
import os
name = "tempnam.tmp"
with open(name, "wt") as myfile:
    #使用 os 的 chmod 函数设定访问许可
    os.chmod(name, 0600)
    myfile.write("eeek")
```

例外情况：

如果文件是创建在一个安全目录中，而且对于非受信用户是不可读的，那么可以允许以默认许可创建文件。例如，如果整个文件系统是可信的或者只有可信用户可以访问，就属于这种情况。

4. 序列化和反序列化

规则 4.1 禁止对不可信来源的数据进行 unpickle 和 load shelf

说明：pickle 模块实现了序列化和反序列化一个 Python 结构体对象。"Pickling"过程是把 Python 分层的对象结构转换成字节流， "unpickling"过程是把字节流转换回分层的对象结构。pickle 存在安全性问题。Python 的文档清晰地表明它不提供安全性保证，因此从一个不可信的数据源接收到的数据禁止进行反序列化。
因为 shelf 依赖于 pickle，所以从不可信来源的数据 load shelf 也是不安全的。

规则 4.2 将敏感对象发送出信任区域前进行签名并加密

说明：敏感数据传输过程中要防止窃取和恶意篡改。使用安全的加密算法加密传输对象可以保护数据，防止对象被非法篡改，保持其完整性。在以下场景中，需要对对象密封和数字签名来保证数据安全：
1)序列化或传输敏感数据
2)没有使用类似于 SSL 传输通道
3) 敏感数据需要长久保存（比如在硬盘驱动器上）

例外情况：

1) 为已加密对象签名在特定场景下是合理的，比如验证从其他地方接收的加密对象的真实性。这是对于被机密对象本身而非其内容的保证。
2) 签名和加密仅仅对于必须跨过信任边界的对象是必需的。始终位于信任边界内的对象不需要签名或加密。例如，如果某网络全部位于信任边界内，始终处于该网络上的对象无需签名或加密。

规则 4.3 禁止序列化未加密的敏感数据

说明：虽然序列化可以将对象的状态保存为一个字节序列，之后通过反序列化该字节序列又能重新构造出原来的对象，但是它并没有提供一种机制来保证序列化数据的安全性。可访问序列化数据的攻击者可以借此获取敏感信息并确定对象的实现细节。攻击者也可恶意修改其中的数据，试图在其被反序列化之后对系统造成危害。敏感数据序列化之后是潜在对外暴露着的，因此序列化信息中不应该包括：密钥、数字证书、以及那些在序列化时引用敏感数据的类。此条规则的意义在于防止敏感数据被无意识的序列化导致敏感信息泄露。

错误示例：

```
class GPSLocation:
    def __init__(self, x, y, id):
        self.x = x
        self.y = y
        self.id = id
```

```
    def __getstate__(self):
        state = dict(self.__dict__)
        return state

import pickle
a = pickle.dumps(GPSLocation(1, 2, 3))
b = pickle.loads(a)
print b.y
```

在这段示例代码中，假定坐标信息是敏感的，那么将其序列化到数据流中使之面临敏感信息泄露与被恶意篡改的风险。

正确示例：

```
class GPSLocation:
    def __init__(self, x, y, id):
        self.x = x
        self.y = y
        self.id = id

    def __getstate__(self):
        state = dict(self.__dict__)
        del state['x']
        del state['y']
        return state

import pickle
a = pickle.dumps(GPSLocation(1, 2, 3))
b = pickle.loads(a)
print b.y
```

在将某个包含敏感数据的类序列化时，程序必须确保敏感数据不被序列化。这包括阻止包含敏感信息的数据成员被序列化，以及不可序列化或者敏感对象的引用被序列化。该示例使用__getstate__方法，从而使它们不包括在依照默认的序列化机制应该被序列化的字段列表中。这样既避免了错误的序列化，又防止了敏感数据被意外序列化。

例外情况：

可以序列化已正确加密的敏感数据。

5. 运行环境

规则 5.1 生产代码不能包含任何调试入口点

说明：由于调试或者测试目的，开发者经常在代码中添加特定的调测代码，这些代码并没有打算与应用一起交付或者部署。当这类的调测代码不小心被留在了应用中，这个应用对某些特殊交互就是开放的。这些后门入口点可以导致安全风险。

6. 其他

规则 6.1 禁止在日志中保存口令、密钥等敏感信息

说明：在日志中不能输出口令、密钥和其他敏感信息，口令包括明文口令和密文口令。对于敏感信息建议采取以下方法：

- 不在日志中打印敏感信息。
- 若因为特殊原因必须要打印日志，则用固定长度的星号（*）代替输出的敏感信息。

规则 6.2 禁止使用私有或者弱加密算法

说明：禁止使用私有算法或者弱加密算法（比如 DES，MD5 等）。应该使用经过验证的、安全的、公开的加密算法。

规则 6.3 基于哈希算法的口令安全存储必须加入盐值（salt）

说明：单向哈希是在一个方向上工作的哈希函数，从预映射的值很容易计算其哈希值，但要根据特定哈希值产生一个预映射的值却是非常困难的。单向哈希主要应用于加密、消息完整性校验、冗余校验等。假如没有加入盐值，则加密原理是：

密文= 哈希算法（明文）

此时，若攻击者获取到密文，同时知道哈希算法，则就可以通过字典攻击来探测和获取口令。

加入盐值之后：

密文= 哈希算法（明文+盐值）

其中盐值可以随机设置，这样即使相同的口令，但盐值不同，密文也不同，从而增加了口令的破解难度、增强安全性。

正确示例（PBKDF2）：

```
import hashlib, binascii
#  b'salt'为安全随机数
dk = hashlib.pbkdf2_hmac('sha256', b'password', b'salt', 10000)
print binascii.hexlify(dk)
```

口令单向 Hash 场景下可以使用 PBKDF2 算法，PBKDF2 是一个密钥导出算法，既可用于导出密钥，也可用于口令保存，并且已在 RFC 2898 标准中定义。它使用最为广泛，能被大多数算法库所支持。

规则 6.4 禁止将敏感信息硬编码在程序中

说明：如果将敏感信息（包括口令和加密密钥）硬编码在程序中，可能会将敏感信息暴露给攻击者。无需反编译 pyc 文件，任何能够访问到 pyc 文件的人都可以获取这些敏感信息。因此，不能将敏感信息硬编码在程序中。

错误示例：

```
class Defpassword:
   defPassword = "123456"
```

恶意用户可以直接打开 pyc 文件发现其中硬编码的默认密码是：123456。

规则 6.5 使用安全随机数

说明: Python 产生随机数的功能在 random 模块中实现，实现了各种分布的伪随机数生成器。产生的随机数可以是均匀分布，高斯分布，对数正态分布，负指数分布以及 alpha，beta 分布，但是这些随机数都是伪随机数，不能应用于安全加密目的的应用中。如果你需要一个真正的密码安全随机数，在 Linux 和类 Unix 下用，请使用/dev/random 生成安全随机数，在 windows 下，使用 random 模块中的 SystemRandom 类来实现。

错误示例：

```
import random

sr = random.randint(0, 100)
```

正确示例：

```
file = open("/dev/random", 'rb')
sr = file.read(10)
file.close()
```

规则 6.6 使用 SSLSocket 代替 Socket 来进行安全数据交互

说明：当在不安全的传输通道中传输敏感数据时，程序必须使用 SSLSocket 类，而不能是 socket 类。SSLSocket 类提供了诸如 SSL/TLS 等安全协议来保证通道不受监听和恶意篡改的影响。SSLSocket 类提供的主要保护包括：

- 完整性保护：SSL防止消息被主动窃取者篡改。
- 认证：在大多数模式下，SSL都对对端进行认证。服务器通常都被认证，如果服务器要求，客户端也可以被认证。
- 保密性（隐私保护）：在大多数模式下，SSL对客户端和服务器之间传输的数据进行加密。这样保护了数据的保密性，被动窃听器不能监听诸如财务或者个人信息之类的敏感信息。

错误示例：未使用 SSLSocket 保护的示例(摘自 python 手册)：

```
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
```

```
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
conn.close()

# Echo client program
import socket

HOST = ''    # The remote host
PORT = 50007              # The same port as used by the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.sendall('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)
```

正确示例：

使用 SSLSocket 保护的示例(代码原型来自 python2.7.3 手册)：

服务端

```
#!/usr/bin/env python
#coding:utf-8
import socket, ssl

bindsocket = socket.socket()
bindsocket.bind(('127.0.0.1', 10023))
bindsocket.listen(5)

# ssl_version 建议使用 TLSv1.2
while True:
    newsocket, fromaddr = bindsocket.accept()
    connstream = ssl.wrap_socket(newsocket,
                                 server_side=True,
                                 certfile="mycertfile",
                                 keyfile="mykeyfile",
                                 ssl_version=ssl.PROTOCOL_TLSv1_2)
    try:
        deal_with_client(connstream)
    finally:
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()

def deal_with_client(connstream):
    data = connstream.read()
```

```
    # null data means the client is finished with us
    while data:
        if not do_something(connstream, data):
            # we'll assume do_something returns False
            # when we're finished with client
            break
        data = connstream.read()
    # finished with client
```

客户端:

```
#!/usr/bin/env python
#coding:utf-8
import socket, ssl, pprint

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# require a certificate from the server
ssl_sock = ssl.wrap_socket(s,
                           ca_certs="/etc/ca_certs_file",
                           cert_reqs=ssl.CERT_REQUIRED)

ssl_sock.connect(('127.0.0.1', 10023))

print repr(ssl_sock.getpeername())
print ssl_sock.cipher()
print pprint.pformat(ssl_sock.getpeercert())

# Set a simple HTTP request -- use httplib in actual code.
ssl_sock.write("""hello, world""")

# Read a chunk of data.  Will not necessarily
# read all the data returned by the server.
data = ssl_sock.read()

# note that closing the SSLSocket will also close the underlying socket
ssl_sock.close()
```

在 python2.7.10 及以上版本 ssl 库有变化,可以参照所使用的 python 版本的手册

该正确代码示例应用 ssl 库使用 SSL/TLS 安全协议保护传输的报文。使用 ssl 的程序如果尝试连接不使用 SSL 的端口，那么这个程序将无限期被阻塞。同理，一个不使用 ssl 的程序如果要同一个使用 SSL 的端口建立连接也将会被阻塞。

例外情况：

因为 SSLSocket 提供的报文安全传输机制性，将造成巨大的性能开销。在以下情况下，普通的套接字就可以满足需求：

- 套接字上传输的数据不敏感。
- 数据虽然敏感，但是已经过恰当加密（参考规则 4.2将敏感对象发送出信任区域前进行

签名并加密）。

- 套接字的网络路径没有越出信任边界。这种情况只有在特定的情况下才能发生。例如，套接字的两端都在同一个本例网络，而且整个网络都是可信的情况时。

规则 6.7 对子类继承的变量要做显式定义和赋初值

说明：在 Python 中，类变量都是作为字典进行内部处理的，并且遵循方法解析顺序（MRO）。子类没有定义的属性会引用基类的属性值，如果基类的属性值发生变化，对应的子类引用的基类的属性的值也相应发生了变化。

错误示例：

```
class A(object):
   x = 1

class B(A):
   pass

class C(A):
   pass

>>> B.x = 2
>>> print A.x, B.x, C.x
1 2 1

>>> A.x = 3
>>> print A.x, B.x, C.x
3 2 3
```

这里虽然没有给 C.x 赋值，但是由于基类的值 A.x 发生改变，在获取 C.x 的值得时候发现它引用的数据发生了变化。在上面这段代码中，因为属性 x 没有在类 C 中发现，它会查找它的基类（在上面例子中只有 A，尽管 Python 支持多继承）。换句话说，就是 C 自己没有 x 属性，独立于 A，因此，引用 C.x 其实就是引用 A.x。

正确示例：如果希望类 C 中的 x 不引用自 A 类，可以在 C 类中重新定义属性 X，这样 C 类的就不会引用 A 类的属性 x 了，值的变化就不会相互影响。

```
class B(A):
   x = 2
class C(A):
   x = -1
>>> print A.x, B.x, C.x
1 2 -1

>>> A.x = 3
>>> print A.x, B.x, C.x
3 2 -1
```

规则 6.8 禁止通过注释的方式删除无用的功能代码

说明：python 的注释包含：单行注释、多行注释、代码间注释、doc string 等。除了 doc string 是使用"""""""括起来的多行注释，常用来描述类或者函数的用法、功能、参数、返回等信息外，其余形式注释都是使用#符号开头用来注释掉#后面的内容。基于 python 语言运行时编译的特殊性，如果在提供代码的时候提供的是 py 文件，即便是某些函数和方法在代码中进行了注释，别有用心的人依然可以通过修改注释来使某些功能启用；尤其是某些接口函数，如果不在代码中进行彻底删除，可能在不知情的情况下就被启用了某些本应被屏蔽的功能。因此根据红线要求，在 python 中不使用的功能、模块、函数、变量等一定要在代码中彻底删除，不给安全留下隐患。即便是不提供源码 py 文件，提供编译过的 pyc、pyo 文件，别有用心的人可以通过反编译来获取源代码，同样注释掉的代码依然有被意外执行的风险，可能会造成不可预测的结果。

错误示例：在 main.py 中有两个接口被注释掉了，但是没有被删除。

```
if __name__ == "__main__":
    if sys.argv[1].startswith('--'):
        option = sys.argv[1][2:]
        if option == "load":
            #安装应用
            LoadCmd(option, sys.argv[2:3][0])
        elif option == 'unload':
            #卸载应用
            UnloadCmd(sys.argv[2:3][0])
        elif option == 'unloadproc':
            #卸载流程
            UnloadProcessCmd(sys.argv[2:3][0])
#       elif option == 'active':
#           ActiveCmd(sys.argv[2:3][0])
#       elif option == 'inactive':
#           InActiveCmd(sys.argv[2:3][0])
        else:
            Loginfo("Command %s is unknown"%(sys.argv[1]))
```

在上例中很容易让其他人看到我们程序中的两个屏蔽的接口，容易造成不安全的因素，注释的代码应该删除。

正确实例：

```
if __name__ == "__main__":
    if sys.argv[1].startswith('--'):
        option = sys.argv[1][2:]
        if option == "load":
            #安装应用
            LoadCmd(option, sys.argv[2:3][0])
        elif option == 'unload':
            #卸载应用
            UnloadCmd(sys.argv[2:3][0])
        elif option == 'unloadproc':
```

```
            #卸载流程
            UnloadProcessCmd(sys.argv[2:3][0])
        else:
            Loginfo("Command %s is unknown"%(sys.argv[1]))
```

建议 6.9 慎用__del__方法

说明：当一个程序正常运行到最后一条语句，或者出现一个未捕获的 SystemExit 异常(由 sys.exit()产生)，或者解释器收到一个 SIGTERM 或者 SIGHUP(在 UNIX 下)信号时，程序就会终止。解释器会将所有已知名称空间下的所有对象、对象的引用记数清为 0(并同时删除所有的名称空间)。当一个对象的引用记数变为零，就会自动调用它的__del__()来销毁该对象。当两个对象存在互相引用时，在程序结束时，这两个对象就无法被销毁(这会造成内存泄漏)。尽管 Python 的垃圾回收机制能在运行时删除这些对象，在程序结束时该机制不会被自动调用。

Python 的垃圾回收过程与常用语言的不一样，Python 按照 locals 中的字典顺序进行垃圾回收，而不是按照创建顺序进行。如果有些对象的__del__方法会访问全局数据或者其他模块中的方法定义，那么由于这些对象有可能已经被删除，__del__方法就有可能引发 NameError 异常。这说明某个对象的 __del__方法执行失败，通常意味着有一项重要操作没有完成(例如关闭一个服务器连接)。因此，最好在代码中显式的执行清理操作，而不是依赖解释器来自动做这件事。

错误示例：

```
# encoding:utf8

class NewClass(object):
    num_count = 0 # 所有的实例都共享此变量，即不单独为每个实例分配
    def __init__(self,name):
        self.name = name
        NewClass.num_count += 1
        print name,NewClass.num_count
    def __del__(self):
        print "Deleting",self.name, "..."
        NewClass.num_count -= 1
        print self.name,"is deleted, num_count=",NewClass.num_count

aa = NewClass("aa")
bb = NewClass("bb")
cc = NewClass("cc")
print locals()

#此处没有手动指定回收流程，故 Python 会自动调用_del_方法回收

print "Over"
```

代码运行结果：

```
aa 1
```

```
bb 2
cc 3
{'aa': <__main__.NewClass object at 0x7fc0ad8216d0>, 'NewClass': <class
'__main__.NewClass'>, 'bb': <__main__.NewClass object at 0x7fc0ad821710>,
'__builtins__': <module '__builtin__' (built-in)>, '__file__':
'jstex.py', '__package__': None, 'cc': <__main__.NewClass object at
0x7fc0ad821750>, '__name__': '__main__', '__doc__': None}
Over
Deleting aa ...
aa is deleted, num_count= 2
Deleting bb ...
Exception AttributeError: "'NoneType' object has no attribute
'num_count'" in <bound method NewClass.__del__ of <__main__.NewClass
object at 0x7fc0ad821710>> ignored
Deleting cc ...
Exception AttributeError: "'NoneType' object has no attribute
'num_count'" in <bound method NewClass.__del__ of <__main__.NewClass
object at 0x7fc0ad821750>> ignored
```

正确示例 1：手动指定回收顺序

```
# encoding:utf8

class NewClass(object):
    num_count = 0 # 所有的实例都共享此变量，即不单独为每个实例分配
    def __init__(self,name):
        self.name = name
        NewClass.num_count += 1
        print name,NewClass.num_count
    def __del__(self):
        print "Deleting",self.name, "..."
        NewClass.num_count -= 1
        print self.name,"is deleted, num_count=",NewClass.num_count

aa = NewClass("aa")
bb = NewClass("bb")
cc = NewClass("cc")
print locals()

del aa
del bb
del cc
print locals()

#此处没有手动指定回收流程，故 Python 会自动调用_del_方法回收
```

```
print "Over"
```

代码运行结果：

```
aa 1
bb 2
cc 3
{'aa': <__main__.NewClass object at 0x7fe0b16306d0>, 'NewClass': <class
'__main__.NewClass'>, 'bb': <__main__.NewClass object at 0x7fe0b1630710>,
'__builtins__': <module '__builtin__' (built-in)>, '__file__':
'jstex1.py', '__package__': None, 'cc': <__main__.NewClass object at
0x7fe0b1630750>, '__name__': '__main__', '__doc__': None}
Deleting aa ...
aa is deleted, num_count= 2
Deleting bb ...
bb is deleted, num_count= 1
Deleting cc ...
cc is deleted, num_count= 0
{'NewClass': <class '__main__.NewClass'>, '__builtins__': <module
'__builtin__' (built-in)>, '__file__': 'jstex1.py', '__package__': None,
'__name__': '__main__', '__doc__': None}
Over
```

正确示例 2：在_del_方法中指定调用自身的变量

```
# encoding:utf8

class NewClass(object):
    num_count = 0 # 所有的实例都共享此变量，即不单独为每个实例分配
    def __init__(self,name):
        self.name = name
        NewClass.num_count += 1
        print name,NewClass.num_count
    def __del__(self):
        print "Deleting",self.name, "..."
        self.__class__.num_count -= 1
        print self.name,"is deleted,
num_count=",self.__class__.num_count

aa = NewClass("aa")
bb = NewClass("bb")
cc = NewClass("cc")
print locals()

#此处没有手动指定回收流程，故 Python 会自动调用_del_方法回收

print "Over"
```

代码运行结果：

```
aa 1
bb 2
cc 3
{'aa': <__main__.NewClass object at 0x7f5c496786d0>, 'NewClass': <class
'__main__.NewClass'>, 'bb': <__main__.NewClass object at 0x7f5c49678710>,
'__builtins__': <module '__builtin__' (built-in)>, '__file__':
'jstex2.py', '__package__': None, 'cc': <__main__.NewClass object at
0x7f5c49678750>, '__name__': '__main__', '__doc__': None}
Over
Deleting aa ...
aa is deleted, num_count= 2
Deleting bb ...
bb is deleted, num_count= 1
Deleting cc ...
cc is deleted, num_count= 0
```

由于不能保证对象的__del__()方法在程序结束时一定会执行，一个比较好的办法就是在程序结束时显式的清除某些对象。例如打开的文件以及网络连接等。你可以给一个自定义对象写一个专门的销毁方法(例如 close())，也可以写一个终止函数，并通过 atexit 模块将其注册到系统中:

```
import atexit
connection = open_connection("deaddot.com")

def cleanup():
    print "Going away..."
    close_connection(connection)

atexit.register(cleanup)
```

也可以以这种方式来调用垃圾回收器:

```
import atexit gc
atexit.register(gc.collect)
```

参考资料

1. https://docs.python.org
2. http://chenjiee815.github.io/simplexmlrpcserverbiao-zhun-ku-yuan-ma-xue-xi.html
3. http://stackoverflow.com/questions/15462016/injecting-arbitrary-code-into-a-python-simplexmlrpc-server
4. http://chenjiee815.github.io/simplexmlrpcserverbiao-zhun-ku-yuan-ma-xue-xi.html
5. http://www.csdn.net/article/1970-01-01/2819716
6. http://blog.chinaunix.net/uid-15868758-id-2758593.html
7. https://docs.python.org/2.7/tutorial/controlflow.html#default-argument-values
8. http://www.wellho.net/mouth/956_python-security-trouble-with-input.html

附录 A

下表中总结了常用数据库中与 SQL 注入攻击相关的特殊字符：

数据库	特殊字符	描述	转义序列
Oracle	%	百分比：任何包括 0 或更多字符的字符串	/% escape '/'
	_	下划线：任意单个字符	/_ escape '/'
	/	斜线：转义字符	// escape '/'
	'	单引号	''
MySQL	'	单引号	\'
	"	双引号	\"
	\	反斜杠	\\
	%	百分比：任何包括 0 或更多字符的字符串	\%
	_	下划线：任意单个字符	_
DB2	'	单引号	''
	;	冒号	.
SQL Server	'	单引号	''
	[	左中括号：转义字符	[[]
	_	下划线：任意单个字符	[_]
	%	百分比：任何包括 0 或更多字符的字	[%]

		符串	
	^	插入符号：不包括以下字符	[^]

附录 B

下表中总结了常用的与命令注入相关的特殊字符：

类型	举例	常见注入模式和结果
管道	|	| shell_command -执行命令并返回命令输出信息
内联	;	; shell_command -执行命令并返回命令输出信息
	&	& shell_command -执行命令并返回命令输出信息
逻辑运算符	$	$(shell_command) -执行命令
	&&	&& shell_command -执行命令并返回命令输出信息
	||	|| shell_command -执行命令并返回命令输出信息
重定向运算符	>	> target_file -使用前面命令的输出信息写入目标文件
	>>	>> target_file -将前面命令的输出信息附加到目标文件
	<	< target_file -将目标文件的内容发送到前面的命令