Python 异常处理

异常处理在任何一门编程语言里都是值得关注的一个话题，良好的异常处理可以让你的程序更加健壮，清晰的错误信息更能帮助你快速修复问题。在Python中，和不分高级语言一样，使用了try/except/finally语句块来处理异常，如果你有其他编程语言的经验，实践起来并不难。

什么是异常?

1.错误

从软件方面来说，错误是语法或是逻辑上的。

语法错误指示软件的结构上有错误，导致不能被解释器解释或编译器无法编译。这些些错误必须在程序执行前纠正。

当程序的语法正确后，剩下的就是逻辑错误了。逻辑错误可能是由于不完整或是不合法的输入所致；

在其它情况下，还可能是逻辑无法生成、计算、或是输出结果需要的过程无法执行。这些错误通常分别被称为域错误和范围错误。

当python检测到一个错误时，python解释器就会指出当前流已经无法继续执行下去。这时候就出现了异常。

2.异常

对异常的最好描述是：它是因为程序出现了错误而在正常控制流以外采取的行为。

这个行为又分为两个阶段：首先是引起异常发生的错误，然后是检测（和采取可能的措施）阶段。

第一阶段是在发生了一个异常条件（有时候也叫做例外的条件）后发生的。

只要检测到错误并且意识到异常条件，解释器就会发生一个异常。引发也可以叫做触发，抛出或者生成。解释器通过它通知当前控制流有错误发生。

python也允许程序员自己引发异常。无论是python解释器还是程序员引发的，异常就是错误

发生的信号。

当前流将被打断，用来处理这个错误并采取相应的操作。这就是第二阶段。

对于异常的处理发生在第二阶段，异常引发后，可以调用很多不同的操作。

可以是忽略错误（记录错误但不采取任何措施，采取补救措施后终止程序。）或是减轻问题的影响后设法继续执行程序。

所有的这些操作都代表一种继续，或是控制的分支。关键是程序员在错误发生时可以指示程序如何执行。

python用异常对象(exception object)来表示异常。遇到错误后，会引发异常。

如果异常对象并未被处理或捕捉，程序就会用所谓的回溯（traceback）终止执行

异常处理

捕捉异常可以使用try/except语句。

try/except语句用来检测try语句块中的错误，从而让except语句捕获异常信息并处理。

如果你不想在异常发生时结束你的程序，只需在try里捕获它。

语法：

以下为简单的try....except...else的语法：

```
try:
<语句>        #运行别的代码
except <名字>:
<语句>        #如果在try部份引发了'name'异常
except <名字>, <数据>:
<语句>        #如果引发了'name'异常，获得附加的数据
else:
<语句>        #如果没有异常发生
```

Try的工作原理是，当开始一个try语句后，python就在当前程序的上下文中作标记，这样当异常出现时就可以回到这里，try子句先执行，接下来会发生什么依赖于执行时是否出现异常。

1. 如果当try后的语句执行时发生异常，python就跳回到try并执行第一个匹配该异常的except子句，异常处理完毕，控制流就通过整个try语句（除非在处理异常时又引发新的异常）。

2. 如果在try后的语句里发生了异常，却没有匹配的except子句，异常将被递交到上层的try，或者到程序的最上层（这样将结束程序，并打印缺省的出错信息）。

3. 如果在try子句执行时没有发生异常，python将执行else语句后的语句（如果有else的话），然后控制流通过整个try语句。

使用except而不带任何异常类型

你可以不带任何异常类型使用except，如下实例：

```
try:
    正常的操作
   ......................except:
    发生异常，执行这块代码
   ......................else:
    如果没有异常执行这块代码
```

以上方式try-except语句捕获所有发生的异常。但这不是一个很好的方式，我们不能通过该程序识别出具体的异常信息。因为它捕获所有的异常。

使用except而带多种异常类型

你也可以使用相同的except语句来处理多个异常信息，如下所示：

```
try:
    正常的操作
   ......................

except(Exception1[, Exception2[,...ExceptionN]]]):
   发生以上多个异常中的一个，执行这块代码
   ......................

else:
    如果没有异常执行这块代码
```

try-finally 语句

try-finally 语句无论是否发生异常都将执行最后的代码。

```
try:<语句>finally:<语句>    #退出try时总会执行raise
```

当在try块中抛出一个异常，立即执行finally块代码。

finally块中的所有语句执行后，异常被再次触发，并执行except块代码。

参数的内容不同于异常。

参数的内容不同于异常。

下面来看一个实例：

```
def div(a, b):
    try:
        print(a / b)
    except ZeroDivisionError:
        print("Error: b should not be 0 !!")
    except Exception as e:
        print("Unexpected Error: {}".format(e))
    else:
        print('Run into else only when everything goes well')
    finally:
        print('Always run into finally block.')

# tests
div(2, 0)
div(2, 'bad type')
div(1, 2)
# Mutiple exception in one line
try:
    print(a / b)
except (ZeroDivisionError, TypeError) as e:
    print(e)
# Except block is optional when there is finally
try:
    open(database)
finally:
    close(database)
# catch all errors and log it
try:
    do_work()
except:
    # get detail from logging module
    logging.exception('Exception caught!')

    # get detail from sys.exc_info() method
    error_type, error_value, trace_back = sys.exc_info()
    print(error_value)
    raise
```

总结如下：

1. except语句不是必须的，finally语句也不是必须的，但是二者必须要有一个，否则就没有try的意义了。
2. except语句可以有多个，Python会按except语句的顺序依次匹配你指定的异常，如果

异常已经处理就不会再进入后面的except语句。

3. except语句可以以元组形式同时指定多个异常，参见实例代码。
4. except语句后面如果不指定异常类型，则默认捕获所有异常，你可以通过logging或者sys模块获取当前异常。
5. 如果要捕获异常后要重复抛出，请使用raise，后面不要带任何参数或信息。
6. 不建议捕获并抛出同一个异常，请考虑重构你的代码。
7. 不建议在不清楚逻辑的情况下捕获所有异常，有可能你隐藏了很严重的问题。
8. 尽量使用内置的异常处理语句来 替换try/except语句，比如with语句，getattr()方法。

经验案例

传递异常 re-raise Exception
捕捉到了异常，但是又想重新引发它（传递异常），使用不带参数的raise语句即可：

```
def f1():
    print(1/0)

def f2():
    try:
        f1()
    except Exception as e:
        raise  # don't raise e !!!

f2()
```

在Python2中，为了保持异常的完整信息，那么你捕获后再次抛出时千万不能在raise后面加上异常对象，否则你的trace信息就会从此处截断。以上是最简单的重新抛出异常的做法。

还有一些技巧可以考虑，比如抛出异常前对异常的信息进行更新。

```
def f2():
    try:
        f1()
    except Exception as e:
        e.args += ('more info',)
        raise
```

如果你有兴趣了解更多，建议阅读这篇博客。

- http://www.ianbicking.org/blog/2007/09/re-raising-exceptions.html

- http://www.ianbicking.org/blog/2007/09/re-raising-exceptions.html

Python3对重复传递异常有所改进，你可以自己尝试一下，不过建议还是同上。

Exception 和 BaseException

当我们要捕获一个通用异常时，应该用Exception还是BaseException？我建议你还是看一下官方文档说明，这两个异常到底有啥区别呢？ 请看它们之间的继承关系。

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration...
      +-- StandardError...
      +-- Warning...
```

从Exception的层级结构来看，BaseException是最基础的异常类，Exception继承了它。BaseException除了包含所有的Exception外还包含了SystemExit，KeyboardInterrupt和GeneratorExit三个异常。

有此看来你的程序在捕获所有异常时更应该使用Exception而不是BaseException，因为另外三个异常属于更高级别的异常，合理的做法应该是交给Python的解释器处理。

except Exception as e和except Exception, e

代码示例如下：

```
try:
    do_something()
except NameError as e:  # should
    pass
except KeyError, e:  # should not
    pass
```

在Python2的时代，你可以使用以上两种写法中的任意一种。在Python3中你只能使用第一种写法，第二种写法被废弃掉了。第一个种写法可读性更好，而且为了程序的兼容性和后期移植的成本，请你也抛弃第二种写法。

raise “Exception string”

把字符串当成异常抛出看上去是一个非常简洁的办法，但其实是一个非常不好的习惯。

```
if is_work_done():
    pass
else:
    raise "Work is not done!" # not cool
```

```
if is_work_done():
    pass
else:
    raise "Work is not done!" # not cool
```

上面的语句如果抛出异常，那么会是这样的：

```
Traceback (most recent call last):
  File "/demo/exception_hanlding.py", line 48, in <module>
    raise "Work is not done!"
TypeError: exceptions must be old-style classes or derived from BaseException, not str
```

这在Python2.4以前是可以接受的做法，但是没有指定异常类型有可能会让下游没办法正确捕获并处理这个异常，从而导致你的程序挂掉。简单说，这种写法是是封建时代的陋习，应该扔了。

使用内置的语法范式代替try/except

Python 本身提供了很多的语法范式简化了异常的处理，比如for语句就处理的StopIteration异常，让你很流畅地写出一个循环。

with语句在打开文件后会自动调用finally中的关闭文件操作。我们在写Python代码时应该尽量避免在遇到这种情况时还使用try/except/finally的思维来处理。

```
# should not
try:
    f = open(a_file)
    do_something(f)
finally:
    f.close()

# should
with open(a_file) as f:
    do_something(f)
```

再比如，当我们需要访问一个不确定的属性时，有可能你会写出这样的代码：

```
try:
    test = Test()
    name = test.name  # not sure if we can get its name
except AttributeError:
    name = 'default'
```

其实你可以使用更简单的getattr()来达到你的目的。

```
name = getattr(test, 'name', 'default')
```

最佳实践

最佳实践不限于编程语言，只是一些规则和填坑后的收获。

1.只处理你知道的异常，避免捕获所有异常然后吞掉它们。

2.抛出的异常应该说明原因，有时候你知道异常类型也猜不出所以然的。

3.避免在catch语句块中干一些没意义的事情。

4.不要使用异常来控制流程，那样你的程序会无比难懂和难维护。

5.如果有需要，切记使用finally来释放资源。

6如果有需要，请不要忘记在处理异常后做清理工作或者回滚操作。

异常速查表

	描述
BaseException	所有异常的基类
SystemExit	解释器请求退出
KeyboardInterrupt	用户中断执行(通常是输入^C)
Exception	常规错误的基类
StopIteration	迭代器没有更多的值
GeneratorExit	生成器(generator)发生异常来通知退出
StandardError	所有的内建标准异常的基类
ArithmeticError	所有数值计算错误的基类
FloatingPointError	浮点计算错误
OverflowError	数值运算超出最大限制
ZeroDivisionError	除(或取模)零 (所有数据类型)
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EOFError	没有内建输入,到达EOF 标记
EnvironmentError	操作系统错误的基类

EnvironmentError	操作系统错误的基类
IOError	输入/输出操作失败
OSError	操作系统错误
WindowsError	系统调用失败
ImportError	导入模块/对象失败
LookupError	无效数据查询的基类
IndexError	序列中没有此索引(index)
KeyError	映射中没有这个键
MemoryError	内存溢出错误(对于Python 解释器不是致命的)
NameError	未声明/初始化对象 (没有属性)
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用(Weak reference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法
SyntaxError	Python 语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用
SystemError	一般的解释器系统错误
TypeError	对类型无效的操作
ValueError	传入无效的参数
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeTranslateError	Unicode 转换时错误
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
FutureWarning	关于构造将来语义会有改变的警告
OverflowWarning	旧的关于自动提升为长整型(long)的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
RuntimeWarning	可疑的运行时行为(runtime behavior)的警告

RuntimeWarning	可疑的运行时行为(runtime behavior)的警告
SyntaxWarning	可疑的语法的警告
UserWarning	用户代码生成的警告