## 1.异常基础

**try/except:**捕捉由python或程序员引起的异常并恢复。

**try/finally:** 无论异常是否发生，执行清理语句

**raise：**手动在代码中触发异常

**assert：**有条件的在代码中触发异常

**with/as:**在python2.6和后续版本中实现环境管理器

### 1.1为什么使用异常？

异常处理对于错误处理、终止动作和事件通知有用。它可以简化特殊情况的处理，并且可以用来实现替代的控制流程。一般来讲，异常处理还可以减少程序所需的检测错误代码的数量，因为所有的错误都由处理器来过滤，你可能不需要测试每个操作的输出。

### 1.2异常的角色

**错误处理**

**事件通知**

**特殊情况处理**

**终止行为**

**非常规控制流程**

## 2.异常处理

### 2.1默认异常处理器

如果代码没有刻意的捕捉代码异常，它将会一直向上返回程序顶层，并启用**默认的异常处理器**:就是打印标准出错消息。这些消息引发的异常还有堆栈跟踪：也就是异常发生时激活的程序行和函数清单。

def **fetcher**(obj, index):

return obj[index]

x = *'spam'*

fetcher(x, 4)

**output：**

Traceback (most recent call last):

File "C:\E\workspace\PythonLearning\chapter17\exc.py", line 6, in <module>

fetcher(x, 4)

File "C:\E\workspace\PythonLearning\chapter17\exc.py", line 2, in fetcher

return obj[index]

IndexError: string index out of range

任何未捕获的异常都流入默认的异常处理器，python在程序的最顶端提供它。这个处理器打印类似的出错消息，并且退出程序。

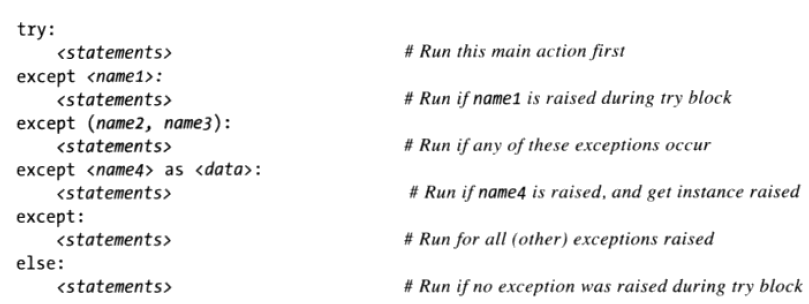
### 2.2捕获异常：try/except/else语句

try:

fetcher(x, 4)

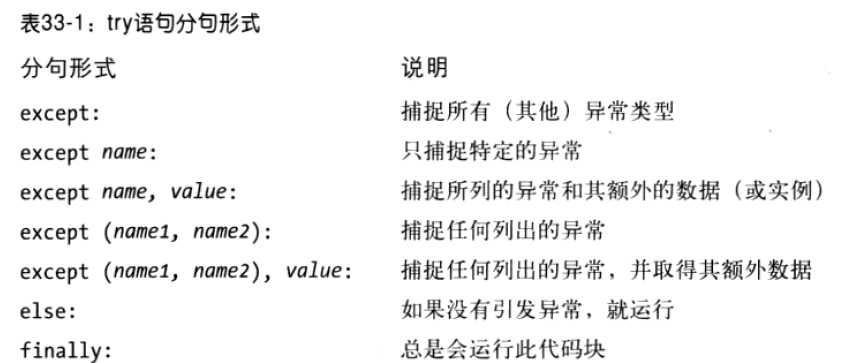
except IndexError:

print *'got exception'*

 try首行底下的代码块代表此语句的主要动作：试着执行程序代码。Except子句定义了try代码块内引发的异常的处理器，而else子句则是提供没发生异常时要执行的处理器。

* 如果try代码块语句执行时的确发生了异常，python就跳回try，执行第一个符合引发异常的except子句下面的语句。当except代码块执行后（除非except代码块又引发了另一异常），控制权就会到整个try语句后继续执行。
* 如果异常发生在try代码块内，没有符合的except子句，异常就会向上传递到程序中的之前进入的try中，或者如果它是第一条这样的语句，就传递到这个进程的顶层（这回使Python终止这个程序并打印默认的出错消息）。
* 如果try首行底下执行的语句没有发生异常，python就会执行else下的语句（如果有的话），控制权会在整个try语句下继续。

#### 2.2.1Try语句分句



Python3.0引入了一个替代方案，捕获一个名为Exception的异常几乎与一个空的except具有相同的效果。

#### 2.2.2Try/else分句

如果没有else，是无法知道控制流程（没有设置和检查布尔标志）是否已经通过try语句，因为没有异常引发或者因为异常发生了且已被处理过了。

### 2.3终止行为：try/finally语句

Try/finally组合，可以定义一定会在最后执行时的收尾行为，无论try代码块中是否发生了异常，finally代码块都会执行。

def **after**():

try:

fetcher(x, 4)

finally:

print *'after fetch'*

print *'after try ?'*

*Output：*

after fetch

Traceback (most recent call last):

File "C:\E\workspace\PythonLearning\chapter17\exc.py", line 13, in <module>

after()

File "C:\E\workspace\PythonLearning\chapter17\exc.py", line 8, in after

fetcher(x, 4)

File "C:\E\workspace\PythonLearning\chapter17\exc.py", line 3, in fetcher

return obj[index]

IndexError: string index out of range

Try/except的组合可用于捕捉异常从中恢复，而try/finally的组合则很方便，可以确保无论try代码块内的代码是否发生了任何异常，终止行为一定会运行。

* 如果try代码块运行时没有异常发生，python会跳至执行finally代码块，然后在整个try语句后继续执行下去。
* 如果try代码块运行时有异常发生，python依然会回来运行finally代码块，但是接着会把异常向上传递到较高的try语句或顶层默认处理器。程序不会再try语句下继续执行。

Finally子句的用途：指明一定要执行的“清理”动作，无论异常发生了没有。

class **MyError**(Exception):

pass

def **stuff**(file):

raise MyError()

file = open(*'data'*, *'w'*)

try:

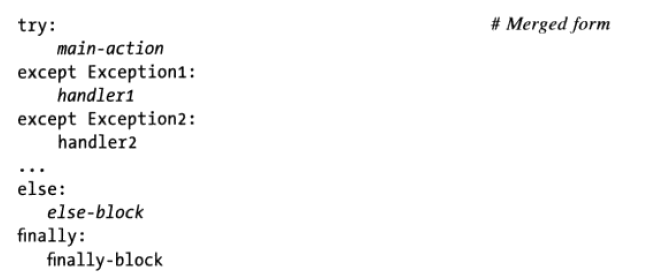
stuff(file)

finally:

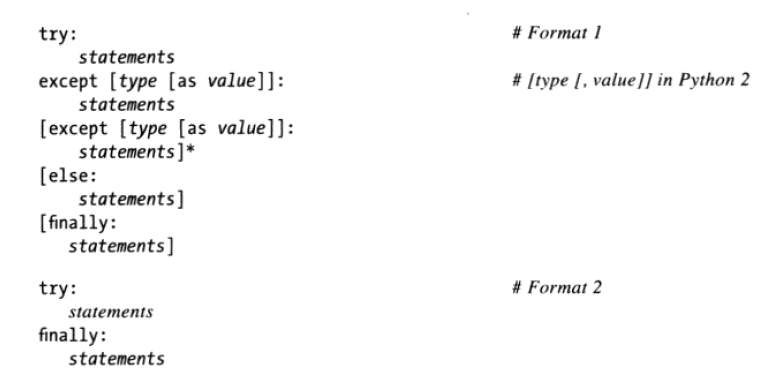
file.close()

print *'not reached'*

### 2.4统一try/except/finally语句



* 如果该程序代码引发异常，那么所有except代码块都会逐一测试，寻找与抛出异常相符的语句。如果没有引发异常则会执行else-block。
* 无论之前发生什么，当main-action代码块完成时，而任何引发的异常都已处理后，finally-block就会执行。即使异常处理器或者else-block内有错误发生而引起了新的异常，finally-block内的程序代码依然会执行。
* 当finally-block执行时，如果异常还存在，就会在finally-block代码块执行后继续传递，而控制权会跳到程序其他地方（到另一个try，或者默认的顶层处理器）。如果finally执行时，没有异常处于激活状态，控制权就会在整个try语句之后继续下去。



sep = *'-'*\*32 + *'\n'*

print sep + *'EXCEPTION RAISED AND CAUGHT'*

try:

x = *'spam'*[99]

except IndexError:

print *'except run'*

finally:

print *'finally run'*

print *'after run'*

print sep + *'NO EXCEPTION RAISED'*

try:

x = *'spam'*[3]

except IndexError:

print *'except run'*

finally:

print *'finally run'*

print *'after run'*

print sep + *'NO EXCEPTION RAISED,WITH ELSE'*

try:

x = *'spam'*[3]

except IndexError:

print *'except run'*

else:

print *'else run'*

finally:

print *'finally run'*

print *'after run'*

print sep + *'EXCEPTION RAISED BUT NOT CAUGHT'*

try:

x = 1/0

except IndexError:

print *'except run'*

finally:

print *'finally run'*

print *'after run'*

*Output：*

--------------------------------

EXCEPTION RAISED AND CAUGHT

except run

finally run

after run

--------------------------------

NO EXCEPTION RAISED

finally run

after run

--------------------------------

NO EXCEPTION RAISED,WITH ELSE

else run

finally run

after run

--------------------------------

EXCEPTION RAISED BUT NOT CAUGHT

finally run

Traceback (most recent call last):

File "C:\E\workspace\PythonLearning\chapter17\mergedexc.py", line 33, in <module>

x = 1/0

ZeroDivisionError: integer division or modulo by zero

### 2.5Raise语句

raise <instance>

raise <class>

raise

Raise语句的组成是：raise关键字，后面跟着可选的要引发的类或者类的实例，raise语句不包括异常名称和额外数据值时，就是重新引发当前异常。

### 2.6Assert语句

Assert <test>, <data>

Assert可视为条件式的raise语句，如果test为假，python就会引发AssertionError异常：data项是异常的额外数据。

Assert几乎都是用来收集用户定义的约束条件，而不是捕捉内在程序设计错误。

### 2.7With/as环境管理器

with语句的基本格式：

with expression [as variable]:

With-block

with/as语句的设计，是为了让必须在程序代码块周围发生的启动和终止活动一定会发生。和try/finally语句（无论异常是否发生，其离开动作都会执行）类似，但是with/as有更丰富的对象协议，可以定义进入和离开的动作。

#### 2.7.1环境管理协议

With语句实际的工作方式：

1. 计算表达式，所得到的对象称为环境管理器，它必须有\_\_enter\_\_和\_\_exit\_\_方法。
2. 环境管理器的\_\_enter\_\_方法会被调用。如果as子句存在，其返回值会赋值给As子句中的变量，否则丢弃。
3. 代码块中嵌套的代码会执行。
4. 如果with代码块引发异常，\_\_exit\_\_(type, value, traceback)方法就会被调用。
5. 如果with代码块没有引发异常，\_\_exit\_\_方法依然会被调用，其type、value以及traceback参数会以None传递。

class **TraceBlock**:

def **message**(*self*, arg):

print *'running'*, arg

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

print *'starting with block'*

return *self*

def **\_\_exit\_\_**(*self*, exc\_type,exc\_value, exc\_tb):

if exc\_type is None:

print *'exited normally\n'*

else:

print *'raise an exception!'*, exc\_type

return False

with TraceBlock() as action:

action.message(*'test 1'*)

print *'reached'*

with TraceBlock() as action:

action.message(*'test 2'*)

raise TypeError

print *'not reached'*

*Output：*

Traceback (most recent call last):

File "C:\E\workspace\PythonLearning\chapter17\withas.py", line 20, in <module>

raise TypeError

TypeError

starting with block

running test 1

reached

exited normally

starting with block

running test 2

raise an exception! <type 'exceptions.TypeError'>

## 3.异常对象

内置异常和用户定义的异常都可以通过类实例对象来表示。基于类的异常有如下特点：

* 提供类型分类，对今后的修改有更好的支持。
* 它们附加了状态信息
* 它们支持继承

### 3.1基于类的异常

Python2.6和python3.0都要求异常通过类来定义。此外python3.0要求异常类派生自BaseException内置异常超类，而不管直接还是间接。

字符串异常在python2.6和python3.0之中已经删除，因为它们都是通过简单的对象标识来匹配，所以它们没有直接的方式来把异常组织到更为灵活的领域。

class **General**(Exception):pass

class **Specific1**(General):pass

class **Specific2**(General):pass

def **raise0**():

X = General()

raise X

def **raise1**():

X = Specific1()

raise X

def **raise2**():

X = Specific2()

raise X

for func in (raise0, raise1, raise2):

try:

func()

except General:

import sys

print *'caught:'*, sys.exc\_info()

Output：

caught: (<class '\_\_main\_\_.General'>, General(), <traceback object at 0x00000000026E32C8>)

caught: (<class '\_\_main\_\_.Specific1'>, Specific1(), <traceback object at 0x00000000026E3348>)

caught: (<class '\_\_main\_\_.Specific2'>, Specific2(), <traceback object at 0x00000000026E3288>)

### 3.2为什么使用类异常？

Mathlib.py

class **NumErr**(Exception):pass

class **Divzero**(NumErr):pass

class **Oflow**(NumErr):pass

...

def **func**():

...

raise Divzero()

Client.py

import mathlib

...

try:

mathlib.func(...)

except mathlib.NumErr:

...report and recover...

### 3.3内置Exception类

BaseException：异常的顶级根类。这个类不能当作是由用户定义的类直接继承。

Exception：与应用相关的异常的顶层根超类。

ArithmeticError：所有数值错误的超类。

OverflowError：识别特定的数值错误的子类。

Exception是python中所有应用程序级别的异常的超类，通常可以使用它作为一个全捕获。

#### 3.3.1默认打印和状态

内置异常还提供了默认打印显示和状态保持，传递给异常类的任何构造函数参数都会保存在实例的args元组属性中，并且打印该实例的时候自动显示。

>>> raise IndexError

Traceback (most recent call last):

File "<pyshell#0>", line 1, in <module>

raise IndexError

IndexError

>>> raise IndexError('spam')

Traceback (most recent call last):

File "<pyshell#1>", line 1, in <module>

raise IndexError('spam')

IndexError: spam

>>> I = IndexError('spam')

>>> I.args

('spam',)

>>>

class **E**(Exception):pass

try:

raise E(*'spam'*, *'eggs'*, *'ham'*)

except E as X:

print X,X.args

Output：

('spam', 'eggs', 'ham') ('spam', 'eggs', 'ham')

### 3.4定制打印显示

class **MyBad**(Exception):

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

return *"Always look on the bright side of life ..."*

try:

raise MyBad()

except MyBad as X:

print X

raise MyBad()

Output：

Traceback (most recent call last):

File "C:\E\workspace\PythonLearning\chapter17\mathlib.py", line 10, in <module>

raise MyBad()

\_\_main\_\_.MyBad: Always look on the bright side of life ...

Always look on the bright side of life ...

### 3.5定制数据和行为

#### 3.5.1提供异常细节

class **FormatError**(Exception):

def **\_\_init\_\_**(*self*, line, file):

*self*.line = line

*self*.file = file

def **parser**():

raise FormatError(42, file=*'spam.txt'*)

try:

parser()

except FormatError as X:

print *"Error at"*, X.file, X.line

print X.args

Output：

Error at spam.txt 42

() #默认构造函数被重写

#### 3.5.2提供异常方法

class **FormatError**(Exception):

logfile = *'formaterror.txt'*

def **\_\_init\_\_**(*self*, line, file):

*self*.line = line

*self*.file = file

def **logerror**(*self*):

log = open(*self*.logfile, *'a'*)

print >>log, *'Error at {0} {1}'*.format(*self*.file,*self*.line)

def **parser**():

raise FormatError(42, file=*'spam.txt'*)

try:

parser()

except FormatError as X:

print *"Error at"*, X.file, X.line

print X.args

X.logerror()

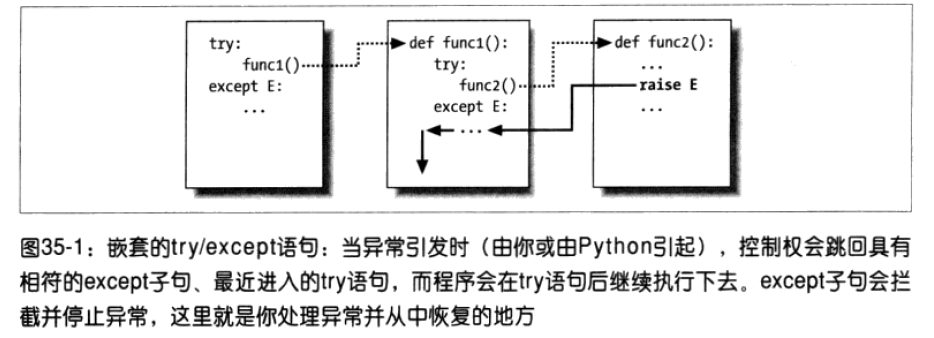
Output：

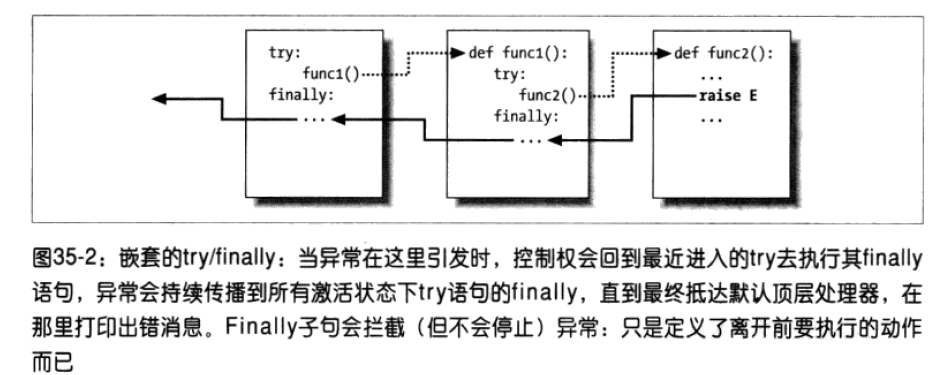
Error at spam.txt 42

()

## 4.异常的设计

### 4.1嵌套异常处理器





### 4.2 异常的习惯用法

#### 4.2.1 异常不总是错误

* 内置的input函数在每次调用时，则是从标准输入串流sys.stdin读取一行文字，并且在文件末尾时引发内置的EOFError。
* Input的空字符串是指空行。除了EOFError的名称，这个异常在这种环境下也只是信号而已，不是错误。
* 调用sys.exit()并在键盘上按下ctrl-C，会分别引发SystemExit和KeyboardInterput。

#### 4.2.2 函数信号条件和raise

class **Found**(Exception):pass

def **searcher**():

if True:

raise Found()

else:

return

try:

searcher()

except Found:

print *"success"*

else:

print *"failure"*

这种代码结构，可用于任何无法返回经是值以表明成功或失败函数。异常提供一种方式来传达结果信号，而不是使用返回值。

#### 4.2.3 关闭文件和服务器连接

异常处理工具通常用来确保系统资源终结，不管在处理过程中是否发生了错误。

确保一个代码块的终止操作的更通用和显式的方式式try/finally语句：

myfile = open(*r'C:\misc\script'*, *'w'*)

try:

...process myfile...

finally:

myfile.close()

With/as语句运行的环境管理器：

with open(*r'C:\misc\script'*, *'w'*) as myfile:

...process myfile...

#### 4.2.4 在try外进行调试

try:

...run program...

except:

import sys

print *"uncaught!"*, sys.exc\_info()[0], sys.exc\_info()[1]

#### 4.2.5 运行进程中的测试

import sys

log = open(*'testlog'*, *'a'*)

from testapi import import moreTests, runNextTest, testName

def **testdriver**():

while moreTests():

try:

runNextTest()

except:

print *'FAILED'*, testName(), sys.exc\_info()[:2]

else:

print *'PASSED'*, testName()

testdriver()

#### 4.2.6 关于sys.exc\_info()

Sys.exc\_info()允许一个异常处理器获取对最近引发的异常的访问。如果没有处理器正在处理，就返回了三个None值的元组。否则，将会返回（type、value和traceback）。

* Type是正在处理的异常的异常类型
* Value是引发的异常的异常类实例
* Traceback是一个traceback对象，代表异常最初发生时所调用的堆栈。

l = [1,2]

try:

l[3]

except:

import sys

print sys.exc\_info()

Output：

(<type 'exceptions.IndexError'>, IndexError('list index out of range',), <traceback object at 0x0000000001D93748>)