



第8章 异常处理结构

刘 卉

huiliu@fudan.edu.cn

异常(exception)

程序运行时引发的错误: 被零除、下标越界、文件不存在、网络异常、类型错误、名字错误、字典键错误、磁盘空间不足,

若错误得不到正确处理, 会导致程序终止运行. 而合理地使用异常处理, 可使程序更加健壮, 具有容错性, 不会因为用户的不当输入或其它运行时原因造成程序终止.

使用异常处理结构可为用户提供更加友好的提示.

程序出现异常或错误之后, 能否调试程序并快速定位和解决问题, 也是程序员综合水平和能力的重要体现.

主要内容

7.1 基本概念

7.2 Python异常类与自定义异常

7.3 Python中的异常处理结构

7.4 断言与上下文管理

7.5 用sys模块回溯最后的异常

8.1 什么是异常

- 语法错误和逻辑错误不属于异常⇒往往会导致异常
e.g. 由于拼写错误而访问不存在的对象⇒NameError.
- 当Python检测到错误时, 解释器就会指出当前流已无法继续执行下去⇒出现异常.
- 异常: 因为程序出错而在**正常控制流以外**采取的行为.
- 异常分为两个阶段:
 - 1) 异常发生阶段;
 - 2) 检测处理阶段.

8.2 Python中的异常类

□ 常用的异常类

异常	描述
NameError	尝试访问一个没有声明的变量
ZeroDivisionError	除数为 0
SyntaxError	语法错误
IndexError	索引超出序列范围
KeyError	请求一个不存在的字典关键字
IOError	输入输出错误（例：读取不存在的文件）
AttributeError	尝试访问未知的对象属性
ValueError	传给函数的参数类型不正确（例：给 int()函数传入字符串）
AssertionError	断言异常

raise语句——显式抛出异常

□ 主动抛出异常, 程序终止

- 不属于标准异常, 而是根据要解决的问题自定义的异常.
- 仅抛出异常, 并不处理异常.

□ raise语法

```
raise [SomeException[, args[, traceback]]]
```

e.g. `raise Exception('spam', 'eggs')`

- SomeException: 必须是一个异常类或异常类的实例.
- Args: 传递给SomeException的参数, 必须是一个元组.
- Traceback: 很少用, 提供一个traceback对象.

Exception类: 所有异常类的基类

[例] 输入某门课程的选课名单(学号/姓名/专业/.....), 发现有重复学号时, 抛出异常.

- 简单起见, 例程中仅输入学号, 其它信息省略.

```
print('Enter the student-IDs for Python Programming:')
info = []
while True:
    ID = input()
    if not ID:
        break
    if ID in info:
        raise Exception('{} is already in the class.'.format(ID))
    info.append(ID)
print(sorted(info))
```



8.3 Python中的异常处理结构

常见的异常处理结构

- try.....except结构
- try.....exceptelse结构
- 带有多个except的try结构
- try.....exceptfinally结构

8.3.1 try.....except结构

- 最基本的处理结构
- 两种形式

形式一 ✓

```
try:  
    try_block #被监控的代码  
except Exception[, reason]:  
    except_block #异常处理代码
```

形式二

```
try:  
    ...  
except BaseException:  
    except_block
```

优势:能处理所有异常

- 建议

- 尽量显式捕捉可能出现的各种异常,并编写具有针对性的代码.
- 最后一个except用来捕捉BaseException.

示例1: 输入校验

```
>>> x = int(input('Enter an integer: '))
Enter an integer: 12.56
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    x = int(input('Enter an integer: '))
ValueError: invalid literal for int() with base 10: '12.56'
>>> x = int(input('Enter an integer: '))
Enter an integer: 1e10
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    x = int(input('Enter an integer: '))
ValueError: invalid literal for int() with base 10: '1e10'
>>> x = int(input('Enter an integer: '))
Enter an integer: ten
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    x = int(input('Enter an integer: '))
ValueError: invalid literal for int() with base 10: 'ten'
```

方法一: 增加额外的逻辑

```
>>> import string
>>> while True:
    x = input('Enter an integer: ')
    for ch in x:
        if ch not in string.digits:
            print('Invalid. Try again.')
            break
    else:
        x = int(x)
        break
```

```
Enter an integer: 12.56
Invalid. Try again.
Enter an integer: 1e10
Invalid. Try again.
Enter an integer: ten
Invalid. Try again.
Enter an integer: 100
>>>
```

方法二: 异常处理(try...except)


```
>>> while True:
    try:    # 将可能引发异常的语句放入try结构
        x = int(input('Enter an integer: '))
        break    # 若无异常则中止循环
    except ValueError:
        print('Invalid. Try again.')
```

```
Enter an integer: 12.56
Invalid. Try again.
Enter an integer: 1e10
Invalid. Try again.
Enter an integer: ten
Invalid. Try again.
Enter an integer: 100
>>>
```

Copy&Run

```
import string

while True:
    x = input('Enter an integer: ')
    for ch in x:
        if ch not in string.digits:
            print('Invalid. Try again.')
            break
    else:
        x = int(x)
        break
```



```
while True:
    try:
        x = int(input('Enter an integer: '))
        break
    except ValueError:
        print('Invalid. Try again.')
```

□ 处理意外情况时:

1. 增加额外的逻辑来进行处理⇒增加程序复杂性.

- 随着必须考虑的错误越来越多, 这种方案的复杂性也随之增加⇒可能会掩盖程序的本来作用.

2. 允许错误发生, 在错误发生时进行处理.

- 程序不会因为发生异常而中止
- 通过使用Python的异常处理机制, 关注代码真正需要做什么, 而不必操心哪里可能出问题.
- 谨慎地使用try语句让代码更易读, 更易写, 出问题时更容易修正.

□ 避免过于依赖异常处理机制.

- 不能使用异常代替常规检查(e.g. if...else).

8.3.2 try.....exceptelse

```
a_list = ['China', 'America', 'England', 'France']
while True:
    n = int(input('请输入字符串的序号(0~3): '))
    try:
        print(a_list[n])
    except IndexError as err:
        print(err)
    else:
        break
```

没有考虑：输入非
整数时产生的异常



indexError_v1&v2.py

□ 分析

- 下标错误: 输入的数值超出序列的下标范围, 产生异常; 执行print(err)语句, 并继续循环;
- 其它情况: 输出列表中对应该序号的字符串, 并退出循环.

Copy&Run

try.....exceptelse: 示例2

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r', encoding = 'UTF-8')
    except FileNotFoundError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

□ 分析

- 功能: 以命令行方式读取并输出文件的行数.
- 出现例外"FileNotFoundError"时, 执行红色的print语句;
- 没有例外时, 执行else块的语句.



8.3.3 带有多个except的try结构

```
try:
    try_block                #被监控的语句
except Exception1:
    except_block_1          #处理异常1的语句
except Exception2:
    except_block_2          #处理异常2的语句
...
```

□ 功能:

- 根据实际产生的例外种类, 执行对应的程序块.



IndexError_v3.py

示例1: 除法

```
try:
    x = eval(input('请输入被除数: '))
    y = eval(input('请输入除数: '))
    z = x/y
except ZeroDivisionError:
    print('除数不能为零')
except TypeError:
    print('被除数和除数应为数值类型')
except NameError:
    print('变量不存在')
else:
    print(x, '/', y, '=', z)
```



division.py

Copy&Run

示例2

- 当有多个except块而且处理相同时,可使用元组的形式处理.

```
import sys
try:
    f = open('sample.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
#输出异常类型
raise #再次抛出该异常
```

```
import sys
try:
    f = open('sample.txt')
    s = f.readline()
    i = int(s.strip())
except (OSError, ValueError,
RuntimeError, NameError):
    pass #忽略该异常
```



multiExcept.py

Copy&Run

8.3.4 try.....exceptfinally结构

□ 特点

- Finally中的语句总会执行.
- 可用于清理工作, 释放资源.

□ 典型结构

```
try:
    try_block    #被监控的代码
except:
    except_block #例外处理程序块
finally:
    finally_block #无论如何都会执行
```

```
>>> try:
        3/0
    except:
        print(3)
    finally:
        print(5)

3
5
```

示例1: 文件的读取

□ 完美代码?

```
try:
    f = open('sample.txt', 'r')
    line = f.readline( )
    print(line)
finally:
    f.close()
```

□ 若文件没有创建, 则在finally中会产生异常.

```
Traceback (most recent call last):
  File "<pyshell#17>", line 6, in <module>
    f.close()
NameError: name 'f' is not defined
```

示例2

- ❑ 例外产生以后, 需有相应的处理.
- ❑ 如果没有相应的except处理块, 代码的执行顺序会发生改变, 直到找到相应的except处理块或者程序退出为止.

```
def divide(x, y):  
    try:  
        result = x/y  
    except ZeroDivisionError:  
        print("division by zero!")  
    else:  
        print("result is", result)  
    finally:  
        print("executing finally clause")
```

```
>>> divide(2, 1)  
result is 2.0  
executing finally clause  
>>> divide(2, 0)  
division by zero!  
executing finally clause  
>>> divide("2", "1")  
executing finally clause  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<stdin>", line 3, in divide  
TypeError: unsupported operand type(s)  
for /: 'str' and 'str'
```

Copy&Run

示例3

□ finally代码中: 返回值要慎重!

```
def demo_div(a, b):  
    try:  
        return a/b  
    except:  
        pass  
    finally:  
        return -1
```

```
>>> demo_div(1, 0)  
-1  
>>> demo_div(1, 2)  
-1  
>>> demo_div(10, 2)  
-1
```

Copy&Run

try:
 <statements> # 运行**try**语句块, 并试图捕获异常

except <name1>:
 <statements> # 如果异常**name1**发现, 那么执行该语句块

except (name2, name3):
 <statements> # 如果元组内的任意异常发生, 那么捕获它

except <name4> **as** <variable>:
 <statements> "**如果name4异常发生, 那么进入该语句块, 并把异常实例命名为variable**"

except:
 <statements> # 发生了以上所有列出的异常之外的异常

else:
 <statements> # 如果没有异常发生, 那么执行该语句块

finally:
 <statement> # 无论是否有异常发生, 均会执行该语句块

8.4 断言与上下文处理

两种特殊的异常处理形式

形式上比通常的异常处理简单

8.4.1 断言——assert语句

□ 语法

`assert expression[, reason]`

- `expression`为真时, 什么都不做; 若为假, 则抛出异常.

□ 用途

- 开发程序时, 对特定的、必须满足的条件进行验证, 仅当`_debug_`为True时有效.
- 当Python脚本以`-O`选项编译为字节码文件时, `assert`语句将被移除以提高运行速度.

示例1

```
try:
    assert 1 == 2, "1 is not equal 2!" # assert expr[,reason]
except AssertionError as reason:
    print("{}:{}".format(reason.__class__.__name__, reason))
```

AssertionError:1 is not equal 2!

Copy&Run

示例2

```
def sum_to_n(n):  
    # precondition: n >= 0  
    assert(n >= 0), "N must be greater than or equal to 0."  
    r = 0  
    for i in range(1, n+1):  
        r += i  
    return r  
    # postcondition: returned sum of 1 to n  
  
if __name__ == '__main__':  
    n = int(input('Enter an integer(>0):'))  
    print('sum of 1 to n is: {}'.format(sum_to_n(n)))
```

- ❑ 例程中assert作用: 确保函数使用的参数符合要求.
- ❑ 不符合要求时: 提示用户存在的问题.



Copy&Run

示例3

- 返回列表中非唯一的所有元素, 要求不能改变元素的顺序.

e.g. $[1, 2, 3, 1, 3] \Rightarrow [1, 3, 1, 3]$

```
def non_unique(data):  
    return [i for i in data if data.count(i)>1]  
  
if __name__ == "__main__":  
    # These "asserts" using for self-checking  
    assert list(non_unique([1, 2, 3, 1, 3])) == [1, 3, 1, 3], "1st example"  
    assert list(non_unique([1, 2, 3, 4, 5])) == [], "2nd example"  
    assert list(non_unique([5, 5, 5, 5, 5])) == [5, 5, 5, 5, 5], "3rd example"  
    assert list(non_unique([10, 9, 10, 10, 9, 8])) == [10, 9, 10, 10, 9], "4th  
example"  
    print("It is all good. Let's check it now")
```

- assert功能: 自检程序



non-unique-elements.py

8.4.2 上下文管理——with语句

□ 语法

```
with context_expr [as obj]:  
    with_block
```

```
obj = context_expr  
obj.__enter__()  
try:  
    with_block  
finally:  
    obj.__exit__()
```

□ 作用

- 提供了一种简单的方法, 解决try...finally结构中的资源释放问题.



文件操作模式

□ 模式一

```
handler = open(filename)
lines = handler.readlines()
handler.close()
print(lines)
```

□ 优缺点

- 常规操作方式: 三步;
- 如有例外产生, 程序运行不正常; 同时资源不能释放.

□ 模式二

```
with open(filename) as handler:
    lines = handler.readlines()
print(lines)
```

□ 优缺点

- 比较安全的操作方式;
- 自动进行资源管理.

□ 模式三

```
try:
    handler = open(filename)
    lines = handler.readlines()
finally:
    handler.close()
print(lines)
```

□ 优缺点

- 较好的操作方式;
- 保证资源的释放.

示例

□ 文件读写

- 下面的代码把文件myfile.txt内容复制到myfile.txt-bk中.
- 思考: 代码是否足够简洁?

```
with open("sample.txt") as fr, open("sample-bk.txt", "w") as fw:  
    for line in fr:  
        fw.write(line)
```



fileBakeup.py

Copy&Run

8.5 用sys模块回溯最后的异常

□ 发生异常时

- Python回溯异常, 给出大量提示信息.
- 可用sys模块回溯最近一次的异常.

```
import sys
try:
    block
except:
    errors = sys.exc_info()
    print(errors)
```

sys.exc_info()可以直接定位错误

```
>>> def A():  
    1/0
```

```
>>> def B():  
    A()
```

```
>>> def C():  
    B()
```

```
>>> C()
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
File "<stdin>", line 2, in C
```

```
File "<stdin>", line 2, in B
```

```
File "<stdin>", line 2, in A
```

```
ZeroDivisionError: division by zero
```

```
>>> try:
```

```
    C()
```

```
except:
```

```
    r = sys.exc_info()
```

```
    print(r)
```

```
(<type 'exceptions.ZeroDivisionError'>,
```

```
ZeroDivisionError('division by zero', ),
```

```
<traceback object at 0x0134C990>)
```

Copy&Run