

# Python基础教程（第三版）（八）异常



c747190cc2f5 (/u/c747190cc2f5)

0.1 2019.05.17 23:03\* 字数 1921 阅读 1 评论 0 喜欢 1

(/u/c747190cc2f5)

学习笔记。

## 8.1 异常是什么

- Python使用异常 对象 表示异常状态，并在遇到错误时引发异常；
- 异常对象没有被捕获时，程序将 终止 并显示错误信息；
- 每个异常都是某个类的 实例；
- 通过捕获异常，可以采取相应的措施使程序 继续进行，而不是放任整个程序失败后终止。

## 8.2 让事情沿你指定的轨道出错

让错误尽在你的掌握之中。

### 8.2.1 raise语句

raise 语句用来引发异常。raise语句的参数是一个类（必须是Exception的子类）或者是实例。当用类作为参数时，将自动创建一个实例（归根结底就是一个异常实例）。

```
raise Exception
```

结果：

```
Traceback (most recent call last):
  File "D:/MyCrawler/venv/Include/test.py", line 3, in <module>
    raise Exception
Exception
```

可以在参数后加上自己设定的相关信息。

```
raise Exception("我是一个天大的错误!")
```

结果:

```
Traceback (most recent call last):
  File "D:/MyCrawler/venv/Include/test.py", line 4, in <module>
    raise Exception("我是一个天大的错误!")
Exception: 我是一个天大的错误!
```

关于异常类

Python内置了多个异常类，了解一下：

类 名	描 述
Exception	几乎所有的异常类都是从它派生而来的
AttributeError	引用属性或给它赋值失败时引发
OSError	操作系统不能执行指定的任务（如打开文件）时引发，有多个子类
IndexError	使用序列中不存在的索引时引发，为LookupError的子类
KeyError	使用映射中不存在的键时引发，为LookupError的子类
NameError	找不到名称（变量）时引发
SyntaxError	代码不正确时引发
TypeError	将内置操作或函数用于类型不正确的对象时引发
ValueError	将内置操作或函数用于这样的对象时引发：其类型正确但包含的值不合适
ZeroDivisionError	在除法或求模运算的第二个参数为零时引发

8.2.2 自定义的异常类

我们可以创建自己的异常类。比如模拟宇宙飞船飞行的python程序正在运行，突然飞船内的“超光速推进装置”过载，这时候就会引发异常。为了更好地表示错误，我们不必使用python内置的类，转而创建自己的子类HyperdriverError，用它来表示“超光速推进装置”的错误，这样就会更自然一些。

如何创建自己的异常类？

务必要直接或者间接的继承Exception（从它的子类继承就是间接）。类似于这样：

```
class HyperdriverError(Exception):  
    pass
```

## 8.3 捕获异常

如果出现了异常，可以逮住它们，把它们都“做掉”。

类似于java中的try/catch，python中使用try/except处理异常。

一个☹：比如要实现两个数相除的程序，但是如果分母为0，就会引发错误使得程序终止，我们可以捕捉ZeroDivisionError来处理异常。

```
try:  
    x = int(input("请输入分子: "))  
    y = int(input("请输入分母: "))  
    print(x/y)  
except ZeroDivisionError:  
    print("分母不能为0! ")
```

结果：

```
请输入分子: 1  
请输入分母: 0  
分母不能为0!
```

### 8.3.1 不用提供参数

捕获了异常之后，如果想要使得异常继续向上一层传播，可以再次使用没有参数的raise重新引发异常。

书中的一个☹：

```
class Calculator:  
    muffled = False  
  
    def calc(self, expr):  
        try:  
            return eval(expr)  
        except ZeroDivisionError:  
            if self.muffled:  
                print("分母不能为0! ")  
            else:  
                raise
```

类中定义了一个抑制开关muffled，当开关打开时，如果发生了除零错误，异常就会被捕捉到，打印“分母不能为0！”；开关关闭时，就会重新引发异常（raise），使异常向上传播。需要注意的时，在这个例子中当异常被捕捉到时，由于函数calc没有返回值，将返回None。

例子的运行演示：

```
# 正常情况
In [2]: calculator = Calculator()
...: calculator.calc("10/2")
Out[2]: 5.0
```

```
# 关闭抑制时
In [3]: calculator.calc("10/0")
-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-3-510121f4d783> in <module>
----> 1 calculator.calc("10/0")
ZeroDivisionError: division by zero
```

```
# 开启抑制时
In [4]: calculator.muffled = True

In [5]: calculator.calc("10/0")
分母不能为0！
```

可以看到，关闭抑制时，异常继续向上传播，如果在上一层没有对其进行捕捉，就会引发程序错误并终止。

在except子句中，可以通过raise引发别的异常。在这种情况下，引发except的原始异常会作为异常上下文储存起来，并会显示在最终的错误消息中。

栗子：

```
try:
    1/0
except ZeroDivisionError:
    raise ValueError("捕捉除零异常的时候再次引发了我...")
```

结果：

```
Traceback (most recent call last):
  File "D:/MyCrawler/venv/Include/test.py", line 26, in <module>
    1/0
ZeroDivisionError: division by zero
```

可以看到，原始异常和引发的异常都显示到了最终的消息当中。

可以通过 `raise...from...` 提供自己的上下文，也可以用`None`禁用上下文。

栗子：

```
try:
    1/0
except ZeroDivisionError:
    raise ValueError("捕捉除零异常的时候再次引发了我...") from ValueError
```

结果：

```
ValueError
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "D:/MyCrawler/venv/Include/test.py", line 28, in <module>
    raise ValueError("捕捉除零异常的时候再次引发了我...") from ValueError
ValueError: 捕捉除零异常的时候再次引发了我...
```

```
try:
    1/0
except ZeroDivisionError:
    raise ValueError("捕捉除零异常的时候再次引发了我...") from None
```

结果：

```
Traceback (most recent call last):
  File "D:/MyCrawler/venv/Include/test.py", line 28, in <module>
    raise ValueError("捕捉除零异常的时候再次引发了我...") from None
ValueError: 捕捉除零异常的时候再次引发了我...
```

### 8.3.2 多个Except字句

在8.3节下的示例中，只捕获了除零异常，如果用户输入了非数字，会引发TypeError，使程序发生错误终止，因此可以这样写：

```
try:
    x = int(input("请输入分子: "))
    y = int(input("请输入分母: "))
    print(x/y)
except ZeroDivisionError:
    print("分母不能为0!")
except TypeError:
    print("必须输入数字!")
```

这样就可以处理多个异常情况。相比频繁使用if，利用异常机制可以使代码更整洁。

### 8.3.3 一箭双雕

可以用括号把想要捕捉的异常写进一个元组，这样就不用频繁地去写except子句了。

```
try:
    x = int(input("请输入分子: "))
    y = int(input("请输入分母: "))
    print(x/y)
except (ZeroDivisionError, TypeError, NameError):
    print("输入错误!")
```

当然，缺点是每种错误都会打印同一种信息。

### 8.3.4 捕获对象

为了弥补上面的缺陷，可以显式的捕捉异常，然后打印此类异常的相关信息：

```
try:
    x = int(input("请输入分子: "))
    y = int(input("请输入分母: "))
    print(x/y)
except (ZeroDivisionError, TypeError, NameError) as e:
    print(e)
```

### 8.3.5 一网打尽

不指定异常类，就会捕捉所有的错误：

```
try:
    x = int(input("请输入分子: "))
    y = int(input("请输入分母: "))
    print(x/y)
except:
    print("出现了一些错误")
```

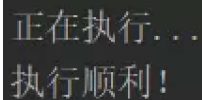
这样做并不能知道发生了什么错误，最好还是显式捕捉错误Exception as e然后打印之。

### 8.3.6 万事大吉时

可以添加 else 语句，如果程序正常运行，那么 else 中的语句就会被执行：

```
try:
    print("正在执行...")
except Exception as e:
    print(e)
else:
    print("执行顺利！")
```

结果：



利用else语句可以对上面的程序进行改良，如果用户输入有误，会让用户重复输入，直到对了为止：

```
while True:
    try:
        x = int(input("请输入分子: "))
        y = int(input("请输入分母: "))
        print(x / y)
    except Exception as e:
        print(e)
    else:
        break
```

运行：

```
请输入分子: afdsa
invalid literal for int() with base 10: 'afdsa'
请输入分子: 2
请输入分母: 0
division by zero
请输入分子: 1
请输入分母: 1
```

敲代码的时候犯了一个错误，写成了While(True)，不应该加括号。

### 8.3.7 最后

还有 finally 子句，可用于在发生异常时执行清理工作，这个子句通常是与try配套的，无论try中发生什么异常，最终都会执行 finally子句。

书中的栗子：

```
try:
    x = 1/0
finally:
    print("正在执行finally")
    del x
```

结果：

```
正在执行finally
Traceback (most recent call last):
  File "D:/MyCrawler/venv/Include/test.py", line 37, in <module>
    x = 1/0
ZeroDivisionError: division by zero
```

为什么要初始化x为None呢，因为在执行1/0时，由于发生了异常，值不会赋给x，而x又没有初始化，所以在del x时又会引发新的异常。

可以把 try、except、else、finally 结合使用。

## 8.4 异常和函数

函数中的异常会一层一层向上传递（到调用它的地方），如果都没有被处理，程序就会终止并打印TraceBack（栈追踪）信息。

栗子：



```
def faulty():  
    raise Exception("Something is wrong...")  
def no_handle():  
    faulty()  
def handle_exception():  
    try:  
        faulty()  
    except:  
        print("Exception handled")
```

运行no\_handle():

```
no_handle()
```

结果:

```
Traceback (most recent call last):  
  File "D:/MyCrawler/venv/Include/test.py", line 46, in <module>  
    no_handle()  
  File "D:/MyCrawler/venv/Include/test.py", line 38, in no_handle  
    faulty()  
  File "D:/MyCrawler/venv/Include/test.py", line 36, in faulty  
    raise Exception("Something is wrong...")  
Exception: Something is wrong...
```

运行handle\_exception():

```
handle_exception()
```

结果:

```
Exception handled
```

并不会打印栈追踪信息，而是执行except子句。  
自己瞎折腾：

```
def faulty():
    raise Exception("Something is wrong...")
def no_handle():
    try:
        faulty()
        print("555")
    except:
        print("666")
def handle_exception():
    try:
        no_handle()
        print("777")
    except:
        print("888")
handle_exception()
```

结果：



```
666
777
```

- 可以看出来，异常如果在某一层被捕捉，就不会再向上传递，所以打印出了666，而非888；
- 在try中如果出现了错误，那么try中剩下的代码就不会再执行，所以没有打印555；
- 在try中调用函数，如果被调用的函数中出现了异常但是已经在函数内部捕捉到，那么就不会向上传递，因此try依然会被执行下去，所以打印777。

## 8.5 异常之禅

两段代码：

```
def describe_person(person):
    print('Description of', person['name'])
    print('Age:', person['age'])
    if 'occupation' in person:
        print('Occupation:', person['occupation'])
```

输出：

```
Description of Throatwobbler Mangrove
Age: 42
```

```
def describe_person(person):  
    print('Description of', person['name'])  
    print('Age:', person['age'])  
    try:  
        print('Occupation:', person['occupation'])  
    except KeyError: pass
```

输出:

```
Description of Throatwobbler Mangrove  
Age: 42  
Occupation: camper
```

其中的person是一个字典，里面有姓名、年龄，可能有职业。

第二中代码效率更高些（高一点点），因为少筛查了一边occupation。

主要就是讲，要养成使用try/except的习惯，这样有时候会比使用其它的方法更加简洁高效。

## 8.6 不那么异常的情况

使用warn()引发一条警告，程序 不会被终止：

```
from warnings import warn  
warn("This is just a warning")  
print("go")
```

结果：

```
go  
D:/MyCrawler/venv/Include/test.py:60: UserWarning: This is just a warning  
warn("This is just a warning")
```

可以使用警告过滤filterwarnings()来忽略掉警告，或者将警告升高到异常。

比如：

```
from warnings import filterwarnings  
filterwarnings("ignore")  
warn("I will be ignored")  
filterwarnings("error")  
warn("I won't be ignored")
```

结果:

```
Traceback (most recent call last):
  File "D:/MyCrawler/venv/Include/test.py", line 63, in <module>
    warn("I won't be ignored")
UserWarning: I won't be ignored
```

警告也有类型，但都是Warning的子类，而Warning又是Exception的子类。发出或者过滤警告时可以指定类型：

```
filterwarnings("default", category=DeprecationWarning)
warn("This Function is really old...", DeprecationWarning)
```

结果:

```
D:/MyCrawler/venv/Include/test.py:61: DeprecationWarning: This Function is really old...
warn("This Function is really old...", DeprecationWarning)
```

```
filterwarnings("ignore", category=DeprecationWarning)
warn("This Function is really old...", DeprecationWarning)
warn("I am not a DeprecationWarning...")
```

结果:

```
D:/MyCrawler/venv/Include/test.py:62: UserWarning: I am not a DeprecationWarning...
warn("I am not a DeprecationWarning...")
```

关于warnings，更详细的内容请见Python文档 (<https://links.jianshu.com/go?to=https%3A%2F%2Fdocs.python.org%2F3.7%2Flibrary%2Fwarnings.html%23>)

## 8.7 小结

1. 异常对象：异常情况使用异常对象表示的，对于异常情况，有多种处理方式；如果忽略，将导致程序终止。
2. 引发异常：使用raise来引发异常。可指定参数来重新引发新的异常，如果没有指定参数，将重新引发该子句捕捉的异常。
3. 自定义异常类：可以通过Exception派生创建自定义异常类。

- 4. 捕获异常：使用except来捕获异常。如果没有要捕获的类，将捕获所有异常。可以指定多个异常类，方法是把它们放在元组中。在同一个try/except中，可以包含多个except子句。
- 5. else： try没有引发异常时执行else中的子句。
- 6. finally： 无论是否引发异常都将会执行。
- 7. 异常和函数： 在函数中引发异常时，异常会传播到调用它的函数。
- 8. 警告： 警告类似于异常，但（通常）只打印一条错误消息。你可以指定错误类别，它们是warning的子类。

小礼物走一走，来简书关注我

赞赏支持

 python基础教程（第三版） (/nb/36261492)

© 著作权归作者所有




c747190cc2f5 (/u/c747190cc2f5)


写了 25958 字，被 1 人关注，获得了 3 个喜欢 (/u/c747190cc2f5)

一个菜鸡的挣扎...

喜欢1



更多分享



写下你的评论...

评论 关闭评论

智慧如你，不想发表一点想法咩~