学习Python Doc第七天: 错误和例外

张朝龙

目录

截至目前，我们见过 Python 的报错信息，但是我们还没有涉及 Python 的错误和例外处理机制。在python中有两类错误：语法错误和例外。

1 语法错误

语法错误可能是最常见的错误，也做好调试。

```
while True print('hello world')
```

显示报错信息：

```
while True print('hello world')
                 ^
SyntaxError: invalid syntax
```

解释器会提示报错信息。

2 例外

即便一个表达式语法正确，其执行过程中仍然可能导致错误。执行过程中的错误叫做例外。

先看几个例外。

1. 除零

```
>>> 10*(1/0)
ZeroDivisionError: division by zero
```

1. 变量未定义

```
>>> 4 + spam*3
NameError: name 'spam' is not defined
```

1. 类型不匹配

```
>>> '2' + 2
TypeError: Can't convert 'int' object to str implicitly
```

在以上的三个例子中，出现的例外有：`ZeroDivisionError`，`NameError`，`TypeError`. 这三个例外的名字是事先定义好的。Python有许多内置的例外信息可以点击这里 查询。

3 处理例外

编写程序时可以针对特定的例外执行相关操作。看代码：

```
while True:
    try:
        x = int(input("please␣enter␣a␣number:"))
    except ValueError:
        print("Oops!␣That␣was␣no␣valid␣number.␣Try␣again")
```

整个程序执行顺序是：

1. 在 `try` 和 `except` 之间的语句首先执行。
2. 如果没有例外发生， `except` 之后的语句就会被跳过。
3. 如果例外发生，并且类型和 `except` 之后的类型一样(本例中是 `ValueError`)。 `except` 后面的语句被执行。

4. 如果例外发生，但是类型和 except 后面的类型不同，那么这个例外就是未处理例外，程序停止执行。

一个 try 可以有多个例外语句。一个 except 语句可以包含多个例外关键词。看代码：

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

另外 try except 语句可以有 else ，看代码：

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

4 发起一个例外

raise 语句强制程序发起例外。看代码：

```
raise NameError('HiThere')
```

5 用户定义例外

通过创建一个新的例外类（一切都是对象，所有的例外类都应当从 Exception 继承而来。）程序可以命名自己的例外。

可以定义例外类，这个例外类可以做其他类能做的一切。通过创建一个基类，可以创建一个可以处理多个不同错误的模块。每个基于这个基类的子类负责处理特定的例外。看代码：


```
class Error(Exception):
    """Base␣class␣for␣exceptions␣in␣this␣module."""
    pass

class InputError(Error):
    """Exception␣raised␣for␣errors␣in␣the␣input.

␣␣␣␣Attributes:
␣␣␣␣␣␣␣␣expression␣--␣input␣expression␣in␣which␣the␣error␣occurred
␣␣␣␣␣␣␣␣message␣--␣explanation␣of␣the␣error
␣␣␣␣"""

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised␣when␣an␣operation␣attempts␣a␣state␣transition␣that's␣not
␣␣␣␣allowed.

␣␣␣␣Attributes:
␣␣␣␣␣␣␣␣previous␣--␣state␣at␣beginning␣of␣transition
␣␣␣␣␣␣␣␣next␣--␣attempted␣new␣state
␣␣␣␣␣␣␣␣message␣--␣explanation␣of␣why␣the␣specific␣transition␣is␣not␣all
␣␣␣␣"""

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

每个例外都是基于 Error 这个类来的。

6 定义 clean-up 动作

try 语句有另外一个可选的语句 finally ， 专门用来定义清理动作。清理动作在所有情况之后执行。看代码：

```
try:
    raise KeyboardInterrupt
finally:
    print('Goodbye,␣world')
```

finally 语句在离开 try 语句之后必须不管是否有例外发生都执行。看代码：

```
1 def divide(x,y):
2     try:
3         result = x/y
4     except ZeroDivisionError:
```


```
5           print("division␣by␣zero")
6       else:
7           print("result␣is␣",result)
8       finally:
9           print("executing␣finally␣clause")
```

输出是：

```
In [3]: divide(2,1)
result is  2.0
executing finally clause

In [11]: divide(2,0)
division by zero
executing finally clause
```

我们可以看到无论如何 `finally` 的语句都会执行。

7 预先定义 clean-up 动作

有些类定义了标准的 `clean-up` 动作。下面的代码打开一个文件，打印其内容到屏幕：

```
for line in open ("myfile.txt"):
    print(line,end="")
```

这段代码的问题是：代码执行结束之后，文件会保持打开状态一段时间。对于简单的脚本而言不是什么问题，但是对于较大的应用，这个可能会导致问题。 `with` 语句允许像 `file` 这样的对象一直处于clean up状态。

```
1 with open("myfile.txt") as f:
2     for line in f:
3         print(line, end="")
```

代码执行之后，文件 `f` 总是关闭的（即使在处理文件过程中发生问题。）。