重庆邮电大学

第9章 调试及异常

主 讲 人：

2019/3/23

目录

1.调试

在本节中，我们首先描述Python在发现语法错误时的处理方式，之后了解Python在发现未处理异常时生成的回溯信息，最后讲解怎样将科学的方法用于调试。

1.调试

1.1.1 处理编译时的错误

看一个实例：

```
File "blocks.py", line 383
        if BlockOutput.save_blocks_as_svg(blocks, svg)
                                                     ^
SyntaxError:invalid syntax
```

1.调试

1.1.1 处理编译时的错误

出现这样问题的原因是我们忘记在if语句条件结尾处放置一个括号。下面给出另一个相当常见的错误实例，但是从中看不出明显的错误。

```
File "blocks.py", line 385
        except ValueError as err:
               ^
SyntaxError: invalid syntax
```

1.调试

1.1.1 处理编译时的错误

```
try:
     blocks = parse(blocks)
     svg = file.replace(".blk", ".svg")
     if not BlockOutput.save_blocks_ as_ svg(blocks, svg):
                    print("Error: failed to save {0}".format(svg)
except ValueError as err:
     …
```

1.1.2 处理运行时的错误

如果运行时发生了未处理的异常，Python就将终止执行程序，并以堆栈回溯（Traceback，也称为向后追踪）的形式显示异常发生的上下文。下面给出一个未处理异常发生时打印出的回溯信息：

```
Traceback (most recent call last):
    File "blocks.py", line 392, in <module>
        main()
    File "blocks.py", line 381, in main
        blocks=parse(blocks)
    File "blocks.py", line 174, in recursive_ descent_parse
        return data.stack[1]
    IndexError: list index out of range
```

（这里由于代码太长无法给出，只是了解如何找到出错位置。）

1.调试

1.1.2 处理运行时的错误

尽管回溯信息初看之下让人困惑不解，但在理解了其结构之后我们会发现它是非常有用的。在上面的实例中，回溯信息告诉了我们应该去哪里寻找问题的根源，当然我们必须自己想办法去解决问题。

1.调试

1.1.2 处理运行时的错误

第2个例子

```
Traceback (most recent call last):
      File "blocks.py", line 392, in <module>
            main()
      File "blocks.py", line 383, in main
            if BIockOutput.save_blocks_ as_svg(blocks, svg):
      File "BltickOutput.py", line 141, in save_blocks as_ svg
            widths, rows=compute_widths_ and_rows(cells, SCALE Bl}
      File "BIockOutput.py", line 95;in compute_widths_ and_rows
            width=len(cell.text)//cell.columns
ZeroDivisionError: integer division or modulo by zero
```

1.1.2 处理运行时的错误

这里，问题出在blocks.py程序调用的BlockOutput.py模块中，这一回溯信息使得我们定位问题变得容易，但它并没有说明错误在哪里发生。第95行BlockOutput.py模块的compute_widths_ and_rows ()函数中，cell.columns的值明显是错误的。不管怎么说，这是导致ZeroDivisionError异常的问题所在，同时我们必须查看前面的错误信息来了解为什么cell.columns会被赋予错误的值。

1.调试

1.2.1 使用pdb调试

pdb是Python自带的一个包，为Python程序提供了一种交互的源代码调试功能，主要特性包括设置断点、单步调试、进入函数调试、查看当前代码、查看栈片段、动态改变变量的值等。pdb提供了一些常用的调试命令，详情如下表所示。

1.调试

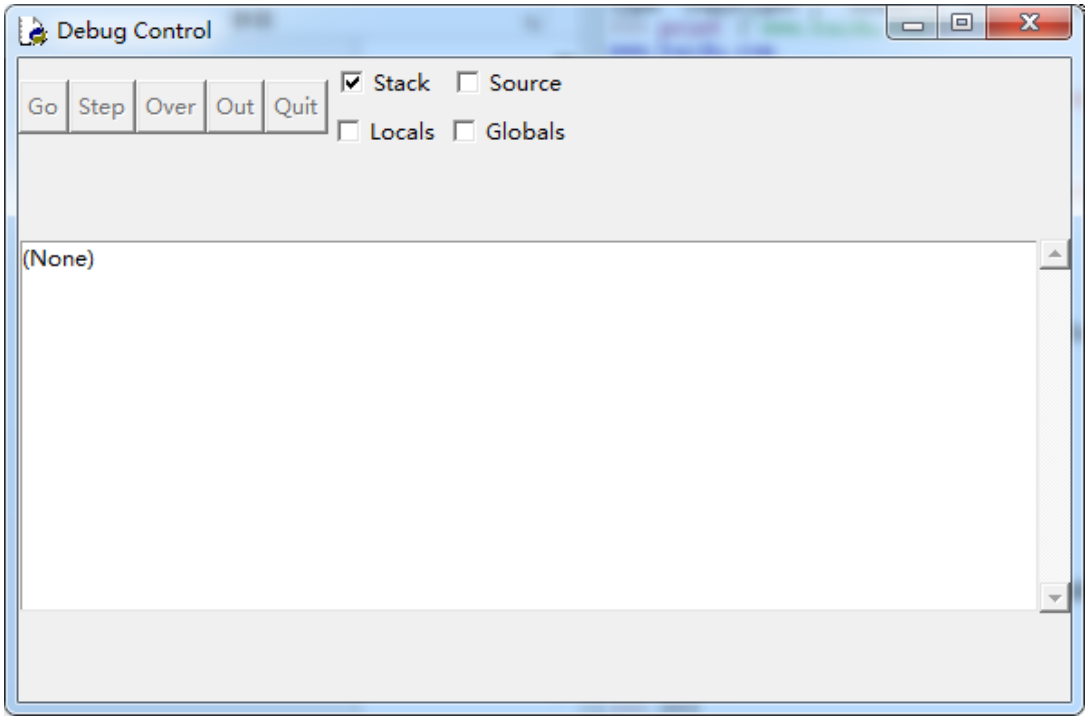
1.2.1 使用pdb调试

命令	解释
break或b	设置断点
continue或c	继续执行程序
list或l	查看当前行的代码段
step或s	进入函数
return或r	执行代码直到从当前函数返回
exit或q	终止并退出
next或n	执行下一行
pp	打印变量的值
help	帮助

1.调试

1.2.2 使用IDLE调试

IDLE中提供了一个调试器，帮助开发人员来查找逻辑错误。下面简单介绍IDLE的调试器的使用方法。



1.调试

1.2.2 使用IDLE调试

① 先在IDLE中写入完整源码

② 编辑保存之后，单击“Run”→“Python Shell”，打开Python Shell 窗口，在这个窗口菜单上，选择“Debug”→“Debuger”，打开“Debug Control”窗口

③ 接下来，在IDLE源码窗口中单击“Run”→“Run Module”或按F5键

④ 单击上面的“Step”按钮，就可以看到其一步一步的执行过程

目录

2.Python中的异常类

在这一节，我们将要面对异常，这是一种可以改变程序中控制流程的程序结构。在Python中，异常会根据错误自动地被触发，也能由代码触发和捕获。异常由四个相关语句进行处理，分别为：try、except、else和finally，接下来将对它们进行介绍。

2.Python中的异常类

2.1 什么是异常

当Python检测到一个错误时，解释器就会指出当前流已无法继续执行下去，这时候就出现了异常。异常是指因为程序出错而在正常控制流以外采取的行为。异常即是一个事件，该事件会在程序执行过程中发生，影响了程序的正常执行。异常处理器（try语句）会留下标识，并可执行一些代码。程序前进到某处代码时，产生异常，因而会使Python立即跳到那个标识，而放弃留下该标识之后所调用的任何激活的函数。

异常分为两个阶段：第一个阶段是引起异常发生的错误；第二个阶段是检测并进行处理的阶段。

2.Python中的异常类

2.2 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.Python中的异常类

2.3 Python的一些内建异常类

异常类名	描　述
Exception	所有异常的基类
NameError	尝试访问一个没有申明的变量
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（比如你要读的文件不存在）
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
EOFError	发现一个不期望的文件尾

3.捕获和处理异常

3.1 try…except…语句

try子句中的代码块放置可能出现异常的语句，except子句中的代码块处理异常：

```
try:
  try块                          #被监控的语句
except Exception as e:
  except块                       #处理异常的语句
```

下面的代码显示了使用try…except…语句诊断异常的过程。

```
list = ['China', 'America', 'England', 'France']
try:
  print(list[4])
except  IndexError as e:
  print('列表元素的下标越界')
```

2.Python中的异常类

3.2 try…except…else语句

如果try范围内捕获了异常，就执行except块；如果try范围内没有捕获异常，就执行else块。

下面的示例修改了上小节的例子，引入循环结构，可以实现重复输入字符串序号，直到检测序号不越界而输出相应的字符串。

```
list = ['China', 'America', 'England', 'France']
print('请输入字符串的序号')
while True:
    n = int(input( ))
    try:
        print(list[n])
    except IndexError as e:
        print('列表元素的下标越界，请重新输入字符串的序号')
    else:
        break
```

2.Python中的异常类

3.3 带多个except的try语句

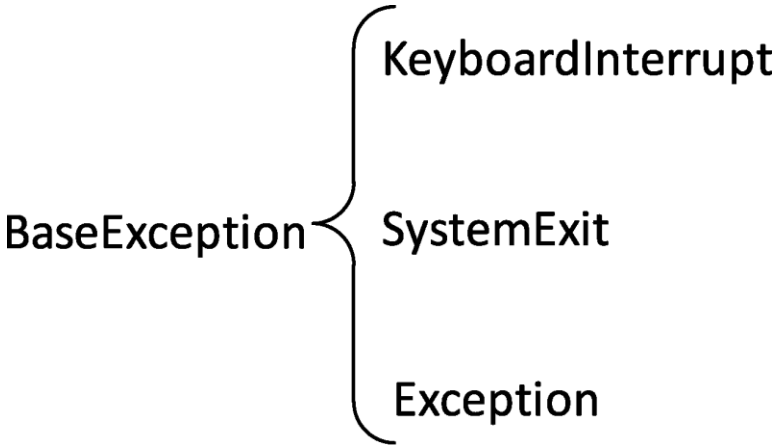
请看下面的例子：输入两数，求两数相除的结果。在数值输入时应检测输入的被除数和除数是否是数值，如果输入的是字符则视为无效。在进行除操作时，应检测除数是否为零。

```
try:
    x = float(input("请输入被除数："))
    y = float(input("请输入除数："))
    z = x / y
except ZeroDivisionError as e1:
    print("除数不能为零")
except ValueError as e2:
    print("被除数和除数应为数值类型")
else:
    print(z)
```

2.Python中的异常类

3.4 捕获所有异常

BaseException是所有内建异常的基类，通过它可以捕获所有类型的异常，KeyboardInterrupt、SystemExit和Exception是从它直接派生出来的子类。按Ctrl+C会抛出KeyboardInterrupt类型的异常，sys模块的sys.exit()会抛出SystemExit类型的异常。其他所有的内建异常都是Exception的子类。



3. 用例实现

3.5 finally子句

下面的示例通过try...finally...语句使得无论文件打开是否正确或是readline()调用失败，都能够正常关闭文件。

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

3. 用例实现

3.5.1 统一try/except/finally

现在，我们可以在同一个try语句中混合finally、except以及else子句。也就是说，我们现在可以编写下列形式的语句。

```
try:
     main-action
except Exception1 as e1:
     handler1
except Exception2 as e2:
     handler2
…
else:
     else-block
finally:
     finally-block
```

目录

4. 两种处理异常的特殊方法

4.1.1 assert语句

assert（断言）语句的语法如下。

```
assert  expression[,  reason]
```

当判断表达式expression为真时，什么都不做；如果表达式为假，则抛出异常。

换句话说，如果test计算为假，Python就会引发异常：data项（如果提供的话）是异常的额外数据。就像所有异常，引发的AssertinError异常如果没被try捕捉，就会终止程序，在此情况下数据项将作为出错消息的一部分显示。

4. 两种处理异常的特殊方法

4.1.1 assert语句

以下程序段举例说明了assert语句的用法。

```
try:
  assert 1 == 3 , "1 is not equal 2!"
except AssertionError as reason:
  print("%s:%s"%(reason.__class__.__name__, reason))
```

程序运行结果如下：

```
AssertionError:1 is not equal 2!
```

4. 两种处理异常的特殊方法

4.1.2 收集约束条件

assert语句通常是用于验证开发期间程序状况的。显示时，其出错消息正文会自动包括源代码的行消息，以及列在assert语句中的值。

```
def f(x):
  assert x < 0, 'x must be negative'
  return x **2
$ python
>>> import asserter
>>> asserter.f(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "asserter.py",line 2, in f
          assert x < 0, 'x must be negative'
AssertionError: x must be negative
```

4. 两种处理异常的特殊方法

4.2 with...as语句

基本使用

with…as语句的目的在于从流程图中把try、except、finally关键字和资源分配释放相关代码全部去掉，而不是像try…except…finally那样仅仅简化代码使之易于使用。with语句的语法如下。

```
with context_expr [as var]:
    with-block
```

在这里context_expr要返回一个对象。如果选用的as子句存在，此对象也返回一个值，赋值给变量名var。

4. 两种处理异常的特殊方法

4.2 with...as语句

2．with...as语句示例

假设在D盘根目录下有一个test.txt文件，该文件里面的内容如下。

How are you?

Fine, thank you.

执行以下程序段，观察运行结果，体会with语句的作用。程序代码如下。

```
#Exp9_6.py
with open('d:\\test.txt') as f:
    for line in f:
        print(line)
```

程序运行结果如下：

```
How are you?
Fine, thank you.
```

目录

5. raise语句

```
class ShortInputException(Exception):
#自定义的异常类。
    def __init__(self, length, atleast):
        Exception.__init__(self)
        self.length = length
        self.atleast = atleast
try:
    s = input('请输入  -->  ')
    if len(s) <3:
        raise ShortInputException(len(s), 3)
except EOFError:
    print('你输入了一个结束标记EOF')      #Ctrl+d
except ShortInputException as x:
    print('ShortInputException: 输入的长度是%d，长度至少应是%d'%(x.length,x.atleast))
else:
    print('没有异常发生。')
```

5. raise语句

5.1 raise语句

程序运行结果如下：

```
请输入-->
你输入了一个结束标记EOF
请输入-->df
ShortInputException：输入的长度是2，长度至少应是3
请输入--> sdfadfd
没有异常发生。
```

5. raise语句

5.2 raise…from语句

Python 3.0（而不是2.6）也允许raise语句拥有一个可选的from子句。

raise exception from otherexception

当使用from的时候，第二个表达式指定了另一个异常类或实例，它会附加到引发异常的__cause__属性。如果引发的异常没有捕获，Python把异常也作为标准出错消息的一部分打印出来：

5. raise语句

5.2 raise…from语句

```
try:
     1/0
except Exception as E:
     raise TypeError('Bad') from E
```

结果如下。

```
Tracback (most recent call last):
     file "<stdin>", line 2, in <module>
ZeroDivisionError: int division or modulo by zero
```

上面的异常是如下异常的直接原因。

```
Tracback (most recent call last):
     File "<stdin>", line 4, in <module>
TypeError: Bad!
```

目录

6.采用sys模块回溯最后的异常

6.1 关于sys.exc_info

sys.exc_info结果通常允许一个异常处理器获取对最近引发的异常的访问。当使用空的except子句来盲目地捕获每个异常以确定引发了什么的时候，将其放入except代码中会特别有用。

sys.exc_info()的返回值tuple是一个三元组(type，value/message，traceback)，这里的属性含义如下。

- type：常的类型。
- value/message：常的信息或者参数。
- traceback：含调用栈信息的对象。

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

6.采用sys模块回溯最后的异常

6.2 使用sys模块的例子

sys模块示例如下。

```
#Exp9_8.py
import sys
try:
     1/0
except:
     tuple  =  sys.exc_info()
print(tuple)
```

程序运行结果如下。

```
 (<type 'exceptions.ZeroDivisionError'>,
ZeroDivisionError('integer  division  or
modulo  by  zero',),  <traceback  object  at
0x01222940>)
```

6.采用sys模块回溯最的异常

6.2 使用sys模块的例子

sys模块示例如下。

```
#Exp9_8.py
import sys
try:
     1/0
except:
     tuple  =  sys.exc_info()
print(tuple)
```

程序运行结果如下。

```
 (<type 'exceptions.ZeroDivisionError'>,
ZeroDivisionError('integer  division  or
modulo  by  zero',),  <traceback  object  at
0x01222940>)
```