Python程序设计：异常处理

2023-2024

王斌辉 副教授

南开大学软件学院

错误和异常

```
while True
    print("Hello world")
```

```
  while True
             ^
SyntaxError: expected ':'
```

- **错误**:
  - 语法错误:
    - 如拼写错误、漏了符号…(IDE能给出提示，编译时报SyntaxError)
  - 逻辑错误:
    - 代码可执行，但执行结果不符合要求
    - IDE、解释器无法给出提示，只有在代码测试时进行排查
  - 错误可以人为避免

```
if score < 60:
    print("你的成绩是优秀")
```

错误和异常

异常Exception：

- 异常是指程序语法正确，但执行中因一些意外而导致的错误，异常并不是一定会发生，例如两数相除时，大部分情况下都能正常执行，但除数为0时将会发生异常
- 默认情况下，程序运行中遇到异常时将会终止，并在控制台打印出异常出现的堆栈信息

```
File "<stdin>", line 8, in <module>
TypeError: Cannot create a consistent method resolution order (MRO) for bases F, E
```

异常类型　　异常解释信息

- Python中，异常是一个类，可以处理和使用，通过异常处理可避免因异常导致的程序终止问题

异常处理

- 异常处理，是在指程序设计时便考虑到可能出现的意外情况，为避免因异常导致程序终止，在程序运行出现错误时，让 Python 解释器执行事先准备好的除错程序，进而尝试恢复程序的执行
- 异常处理，可以有效提升用户体验，甚至在程序崩溃前也可以做一些必要的工作，如将内存中的数据写入到文件、关闭打开的文件、释放分配的内存等
- 异常处理语法框架与逻辑：

```
try:
    pass  # do something
except Exception as error:
    pass  # error handling
else:
    pass  # optional, do sth if there's no error
finally:
    pass  # optional, clean up
```

通常将可能发生异常的代码放在try语句中，如发生异常则通过except语句来捕获并采取一些额外处理，如没有发生异常则执行后面的else语句，最后执行finally语句做一些收尾的操作

异常处理

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

- 首先执行 try 子句 （try 和 except 关键字之间的（多行）语句）
- 如果没有触发异常，则跳过 except 子句，try 语句执行完毕
- 如果在执行 try 子句时发生了异常，则跳过该子句中剩下的部分。 如果异常的类型与 except 关键字后指定的异常相匹配，则会执行 except 子句，然后跳到 try/except 代码块之后继续执行
- 如果发生的异常与 except 子句 中指定的异常不匹配，则它会被传递到外部的 try 语句中；如果没有找到处理程序，则它是一个 ***未处理异常*** 且执行将终止并输出如上所示的消息

异常处理

- try 语句可以有多个 except 子句，来为不同的异常指定处理程序，但最多只有一个处理程序会被执行
- 处理程序只处理对应的 try 子句中发生的异常，而不处理同一 try 语句内其他处理程序中的异常
- except 子句 可以用带圆括号的元组来指定多个异常

```
except(RuntimeError, TypeError, NameError):
    pass
```

异常处理

- 如果发生的异常与 except 子句中的类是同一个类或是它的基类时，则该类与该异常相兼容（反之则不成立）右边依次输出 B, C, D
- 如果颠倒 except 子句 的顺序（把 except B 放在最前），则会输出 B, B, B（即触发了第一个匹配的 except 子句）

```
class B(Exception): pass

class C(B): pass

class D(C): pass


for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

异常处理

- Exception 类为内置非致命错误类的基类
- 最后的except子句，使用Exception兜底，以拦截其它未可知的异常，是一种不错的处理方式

```
import sys

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

异常处理

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

- try … except 语句具有可选的 else 子句，该子句适用于 try 子句没有引发异常但又必须要执行的代码
- else 子句如果存在，它必须放在所有 except 子句之后
- 使用 else 子句比向 try 子句添加额外的代码要好，这样可以避免意外捕获非 try … except语句保护的代码触发的异常

异常处理

```
def this_fails():
    x = 1 / 0
```

- try … except 语句不仅能捕获try之句中的异常，也能捕获间接的（如函数内部的）异常

```
try:
    this_fails()
except ZeroDivisionError as err:
    print("Handling run-time error:", err)
```

异常处理

异常的传导

```
Traceback (most recent call last):
  File "<stdin>", line 13, in <module>
    main()
  File "<stdin>", line 3, in main
    def main(): first_method()
  File "<stdin>", line 5, in first_method
    def first_method(): second_method()
  File "<stdin>", line 7, in second_method
    def second_method(): third_method()
  File "<stdin>", line 10, in third_method
    raise SelfException("自定义异常信息")
__main__.SelfException: 自定义异常信息
```

```
class SelfException(Exception): pass

def main(): first_method()

def first_method(): second_method()

def second_method(): third_method()

def third_method():
    raise SelfException("自定义异常信息")


main()
```

手动触发异常

- raise 语句可以强制触发指定的异常
- raise语句可以单独抛出内置的异常类

```
raise NameError("Hi There")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: Hi There
```

```
raise ValueError  # shorthand for "raise ValueError()"
```

- 如只想判断是否触发了异常，但并不打算处理该异常，则可以使用更简单的raise 语句重新触发异常

```
try:
    raise NameError("Hi There")
except NameError:
    print("An exception flew by!")
    raise
```

```
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: Hi There
```

异常链

- 如except子句中有未触发的异常，该异常的 __context__ 属性会被自动设为已处理的异常，并随try子句异常一并抛出

```
try:
    open("database.sqlite")
except OSError:
    raise RuntimeError("unable to handle error")
```

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'
During handling of the above exception, another exception occurred:
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: unable to handle error
```

- 为了显示某个异常来自于其它异常的结果，可使用raise...from...的语法结构：

```
raise RuntimeError from exc
```

exc(original) must be exception instance or None

异常链

- raise new_exc from original_exc结构，在转换异常(隐式转为显示)时非常有用

```
def func():
    raise ConnectionError

try:
    func()
except ConnectionError as exc:
    raise RuntimeError("Failed to open database") from exc
```

隐式异常（原original）

显式异常（新）

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in func
ConnectionError
The above exception was the direct cause of the following exception:
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Failed to open database
```

- 可使用 from None 的方式让新异常替换原异常以显示其目的，同时让原异常在__context__ 中保持可用状态以便在调试时内省

```
try:
    open("database.sqlite")
except OSError:
    raise RuntimeError from None
```

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

异常处理

- **清理异常**：无论 try 语句是否触发异常，都会执行 finally 子句

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

- 如果执行try子句期间触发了某个异常，则该except子句应处理此异常，若该异常无except子句处理，则在finally子句执行后，该异常会被重新触发
- except 或 else 子句执行期间也会触发异常， 同理，该异常会在 finally 子句执行之后被重新触发
- 如果 finally 子句中包含 break、continue 或 return 等语句，异常将不会被重新触发
- 如果执行 try 语句时遇到 break、continue 或 return 语句，则 finally 子句在执行 break、continue 或 return 语句之前执行
- 如果 finally 子句中包含 return 语句，则返回值来自 finally 子句的某个 return 语句的返回值，而不是来自 try 子句的 return 语句的返回值

异常处理

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

```
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'
```

```
divide(2, 1)
divide(2, 0)
divide("2", "1")
```

在实际应用中，finally子句对于释放外部资源（如文件或者网络连接）非常有用，无论是否成功使用资源

异常处理

```
def bool_return():
    try:
        return True
    finally:
        return False


print(bool_return())
```

```
False
```

异常处理

- **预定义的清理操作**：某些对象定义了不需要该对象时要执行的标准清理操作，无论使用该对象的操作是否成功，都会执行清理操作，如打开文件并输出内容

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

这段代码的问题在于：执行完代码后，文件在一段不确定的时间内处于打开状态。在简单脚本，中这没有问题，但对于较大的程序来说可能会出问题

- **with语句**保证在其子句执行完毕后，即使处理行时遇到问题，都会关闭文件 f

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

异常处理

- 内置异常类层次结构
  - **BaseException是所有内置异常类的基类**
  - 直接继承BaseException的内置异常类有：
    - BaseExceptionGroup
    - GeneratorExit：关闭generator或coroutine时引发
    - KeyboardInterrupt:当用户按下中断键 (通常为 Control-C 或 Delete) 时将被引发
    - SystemExit
    - Exception
  - **用户自定义的异常不应直接继承BaseException，而应继承Exception类或其子类，**除直接继承BaseException的异常外，其它内置异常皆继承自Exception，且为非致命错误
- 大多数内置异常都基于 C 实现以保证运行效率(详见: Objects/exceptions.c)

- args：传给异常构造器的参数元组，大部分异常只接受一个给出错误信息的单独字符串
- with_traceback(tb)：raise...from...替换
- __suppress_context__
- __context__
- __cause__
- add_note(note): Python3.11新增
- __notes__：notes列表

继承Exception的内置异常

- ExceptionGroup [BaseExceptionGroup]

继承Exception的内置异常

- ArithmeticError：各种算术类错误而引发的内置异常
  - FloatingPointError：目前未被使用
  - OverflowError
  - ZeroDivisionError
- AssertionError：当 assert 语句失败时将被引发
- AttributeError：当属性引用或赋值失败时将被引发，当一个对象根本不支持属性引用或属性赋值时则将引发 TypeError
- BufferError：当与 缓冲区 相关的操作无法执行时将被引发
- EOFError：当 input() 函数未读取任何数据便达到文件结束条件 (EOF) 时将被引发

继承Exception的内置异常

- ImportError：当 import 语句加载模块遇到问题时将被引发
  - ModuleNotFoundError
- AttributeError：
  - IndexError：当序列抽取超出范围时将被引发，若指定索引不是整数则引发TypeError
  - KeyError：当在现有键集合中找不到指定的映射（字典）键时将被引发
  - LookupErrorBufferError
- MemoryError:当一个操作耗尽内存但仍可（通过删除一些对象）挽救时将被引发
- NameError：当某个局部或全局名称未找到时将被引发
  - UnboundLocalError

继承Exception的内置异常

- ReferenceError
- RuntimeError：
  - NotImplementedError:抽象方法在派生类中没被实现时触发，与NotImplemented不可互换
  - RecursionErrorAttributeError：
- StopAsyncIteration
- StopIteration
- SyntaxErrorNameError：
  - IndentationError
    - TabError

继承Exception的内置异常

- SystemError
- TypeError
- ValueError：
  - UnicodeError：
    - UnicodeDecodeError
    - UnicodeEncodeError
    - UnicodeTranslateError

继承Exception的内置异常

- OSError：
  - BlockingIOError
  - ChildProcessError
  - ConnectionErrorAttributeError ：
    - BrokenPipeError
    - ConnectionAbortedError
    - ConnectionRefusedError
    - ConnectionResetError
  - FileExistsError
  - FileNotFoundError
  - InterruptedError
  - IsADirectoryError
  - NotADirectoryError
  - PermissionError
  - ProcessLookupError
  - TimeoutError

继承Exception的内置异常

- Warning：
  - BytesWarning
  - DeprecationWarning
  - EncodingWarning
  - FutureWarning
  - ImportWarning
  - ResourceWarning
  - RuntimeWarning
  - SyntaxWarning
  - UnicodeWarning
  - UserWarning
  - PendingDeprecationWarning