

异常处理

黄书剑





程序中的错误：

- **逻辑错误：**
 - 程序因为设计不当等原因不能完成预期的功能
 - 例如：
 - 把两个数相加写成了相乘
 - 排序功能未能正确排序
 -
- **可以通过静态分析和动态测试发现**
 - 代码逻辑分析、assert、doctest等

OJ: Wrong Answer



程序中的错误：

- **语法错误：**

- 指程序的书写不符合语言的语法规则
- 例如：括号不匹配，漏写复合语句的冒号，缩进不一致等

- **往往可以由静态检查发现**

- 如C++编译器，编辑器语法检查插件等（LBYL）
- 注意：语法错误将引发python的运行时异常（EAFP）

OJ: Compile
Error

OJ: non-zero

```
x = 5
y = x ( 3 ) )
```

```
y = x ( 3 ) )
```

^

SyntaxError: invalid syntax

```
x = 5
y = x + 3
```

```
y = x + 3
```

^

IndentationError: unexpected indent



程序中的错误

- **运行时异常：**
 - 程序运行过程中产生的错误
 - 例如：除数为0、文件不存在、语法不正确、找不到名字、类型错误、值错误等

```
x = eval(input())  
y = eval(input())  
print(x/y)
```

```
10  
0  
Traceback (most recent call last):  
  File  
    "/Users/huangshujian/Working/python/test  
    .py", line 3, in <module>  
      print(x/y)  
ZeroDivisionError: division by zero
```



程序中的错误

- **运行时异常：**
 - 程序运行过程中产生的错误
 - 例如：除数为0、文件不存在、语法不正确、找不到名字、类型错误、值错误等
- **往往由程序运行过程中的环境和状态导致**
 - 环境状态与预期不一致（EAFP）
- **运行时异常并不总是发生，往往无法避免**
 - 危害程序的鲁棒性

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: list.remove(x): x not in list
```

异常的处理



处理预期和状态的不一致

- 可以通过逻辑判断进行状态检查

```
print(x/y)
```

```
if y != 0:  
    print(x/y)  
else:  
    print("ZeroDivisionError")
```

- 如何处理不一致？
 - 控制台打印出错信息
 - 中止程序
 - 重新获取y？
- 当前代码可能无法处理
 - 就地处理 v.s. 异地处理

```
else:  
    exit(1)
```

OJ: non-zero

```
def funcDiv(x , y):  
    if y != 0:  
        return x/y  
    else:  
        pass
```



处理预期和状态的不一致

- 通过逻辑判断进行状态检查

```
print(x/y)
```

```
if y != 0:  
    print(x/y)  
else:  
    return "ERROR"
```

- 如何处理不一致？

- 通过返回值标识错误
 - 调用者增加额外代码
 - 多层函数调用时处理困难

```
def data_processing(x, y, comp):  
    return comp(x, y)  
  
def compute(x, y):  
    funcDiv(y, x) + funcDiv(x, y)
```

就地处理 v.s. 异地处理



处理预期和状态的不一致

- 通过逻辑判断进行状态检查

```
print(x/y)
```

```
if y != 0:  
    print(x/y)  
else:  
    return "ERROR"
```

- 如何处理不一致？

- 通过返回值标识错误
- 通过全局变量标识
- 通过对象、函数的状态标识
-

```
else:  
    global ERRCODE  
    ERRCODE = 1
```

当嵌套关系较为复杂时，
传递状态标识也较为繁琐

就地处理 v.s. 异地处理



结构化异常处理机制

- 将异常发现与异常处理的分离

```
def funcDiv(x , y):  
    if y != 0:  
        return x/y  
    else:  
        raise Exception("div-zero")  
  
def data_processing(x, y, comp):  
    return comp(x, y)  
  
def compute(x, y):  
    return funcDiv(y, x) + funcDiv(x, y)  
  
data_processing(1, 0, compute)
```

raise <exception>

发现异常!

C++: throw



结构化异常处理机制

- 将异常发现与异常处理的分离

raise <exception>

发现异常!

```
def funcDiv(x , y):  
    if y != 0:  
        return x/y  
    else:  
        raise Exception
```

```
def data_processing(x  
    return comp(x, y)
```

```
def compute(x, y):  
    return funcDiv(y,
```

```
data_processing(1, 0, compute)
```

```
Traceback (most recent call last):  
  File ".py", line 26, in <module>  
    data_processing(1, 0, compute)  
  File ".py", line 21, in data_processing  
    return compute(x, y)  
  File "n.py", line 24, in compute  
    funcDiv(y, x) + funcDiv(x, y)  
  File "n.py", line 16, in funcDiv  
    raise Exception("div-zero")  
Exception: div-zero
```

按照调用链依次输出信息!



结构化异常处理机制

- 将异常发现与异常处理的分离

```
def funcDiv(x , y):  
    return x/y  
  
def data_processing(x, y, comp):  
    return comp(x, y)  
  
def compute(x, y):  
    return funcDiv(y, x) + funcDiv(x, y)  
  
data_processing(1, 0, compute)
```

raise <exception>

发现异常!

解释器会自动返回
计算中的异常 (即
自动发现异常)



异常对象

- **被抛出的异常可以用类型进行区别**
 - NameError (变量名未绑定)
 - TypeError (类型错误)
 - ValueError (值错误)
 - AttributeError (对象没有指定的属性)
- **根据不同类型的异常可能可以采取不同的操作**
- **built-in异常都是Exception类的派生类**



```
+-- Exception
+-- StopIteration
+-- StopAsyncIteration
+-- ArithmeticError
|   +-- FloatingPointError
|   +-- OverflowError
|   +-- ZeroDivisionError
+-- AssertionError
+-- AttributeError
+-- BufferError
+-- EOFError
+-- ImportError
|   +-- ModuleNotFoundError
+-- LookupError
|   +-- IndexError
|   +-- KeyError
+-- MemoryError
+-- NameError
|   +-- UnboundLocalError
```

```
+-- OSError
+-- BlockingIOError
+-- ChildProcessError
+-- ConnectionError
|   +-- BrokenPipeError
|   +-- ConnectionAbortedError
|   +-- ConnectionRefusedError
|   +-- ConnectionResetError
+-- FileExistsError
+-- FileNotFoundError
+-- InterruptedError
+-- IsADirectoryError
+-- NotADirectoryError
+-- PermissionError
+-- ProcessLookupError
+-- TimeoutError
```



```
+-- ReferenceError
+-- RuntimeError
|   +-- NotImplementedError
|   +-- RecursionError
+-- SyntaxError
|   +-- IndentationError
|       +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|   +-- UnicodeError
|       +-- UnicodeDecodeError
|       +-- UnicodeEncodeError
|       +-- UnicodeTranslateError
+-- Warning
|   +-- DeprecationWarning
|   +-- PendingDeprecationWarning
|   +-- RuntimeWarning
|   +-- SyntaxWarning
|   +-- UserWarning
|   +-- FutureWarning
|   +-- ImportWarning
|   +-- UnicodeWarning
|   +-- BytesWarning
|   +-- ResourceWarning
```



异常处理

- 通过复合语句进行异常处理代码声明
- try语句
 - 声明可能出现异常的语句
- except语句
 - 用于按**类型**捕获可能出现的异常
 - 声明对特定异常进行处理的操作
 - 可以为捕获的异常对象建立绑定
 - except <eclass> as <name>

```
try:  
    <try suite>  
except <exception class>:  
    <except suite>
```




```
def funcDiv(x , y):  
    return x/y  
def data_processing(x, y, comp):  
    return comp(x, y)  
def compute(x, y):  
    return funcDiv(y, x) + funcDiv(x, y)
```

还有可能出现其他问题!

```
try:  
    x = data_processing(0, 1, compute)  
    print("Results: ", x)  
except: # or except Exception:  
    print("handling div-zero error!")
```

捕获任意类型的异常

```
except ZeroDivisionError:
```

捕获某个具体类型的异常



```
def funcDiv(x , y):  
    return x/y  
def data_processing(x, y, comp):  
    return comp(x, y)  
def compute(x, y):  
    return funcDiv(y, x) + funcDiv(x, y)
```

还有可能出现其他问题！

```
try:  
    x = data_processing("1", 1, compute)  
    print("Results: ", x)  
except ZeroDivisionError:  
    print("handling div-zero error!")  
except TypeError:  
    print("handling type error!")
```

可以利用不同类型的
异常调用不同的处理
程序
如果需要，也可以在不
同位置处理不同的
异常



异常处理

- 通过复合语句进行异常处理代码声明
- try语句
- except语句
- else语句
 - 没有发生异常时执行的操作
- finally语句
 - 无论是否发生异常都执行的操作
 - 特别用于资源清理等场合
 - 关闭文件、释放占有的资源、输出特定信息

```
try:  
    <try suite>  
except <exception class>:  
    <except suite>  
else:  
    <else suite>  
finally:  
    <final suite>
```



python中的异常

- **运行时的值和类型相关异常**
 - NameError, TypeError, ValueError, AttributionError
- **文件输入输出中的异常**
 - FileNotFoundError、 FileExistsError.....
-
- **Python解释器解释执行过程中，对发生的语法错误抛出异常**
 - SyntaxError
- **断言用于判断表达式的值，在表达式值为false的时候触发异常**
 - AssertionError

用户自定义异常



- 继承Exception类或其派生类，存储相关的异常信息



回顾：继承

- 一个类所描述的对象特征可以从其它的类**继承**获得。

派生类（子类）

基类（父类）

```
class CheckingAccount(Account):  
    def withdraw_charge(self, amount):  
        self.balance -= (amount + 1)
```

- CheckingAccount的对象拥有Account对象的属性

```
jack = CheckingAccount("Jack")  
print(jack.balance)  
jack.deposit(100)
```

在派生类中找不到的名绑定关系，会在其基类中继续寻找

```
class CheckingAccount(Account):  
    def withdraw_charge(self, amount):  
        Account.withdraw(self, amount + 1)
```

回顾：派生类中的重写

- 如果在派生类中需要对现有功能进行改变或扩充，可以重新进行成员函数的定义

```
class CheckingAccount(Account):  
    def withdraw_charge(self, amount):  
        self.withdraw(self, amount + 1)  
        super().withdraw(amount + 1)
```

- 保持函数名称一致有利于保持接口的一致性
 - 同时新取代了原先的方法，防止了调用错误的发生
- 在派生类的对象发生成员函数调用时，根据成员访问规则，将优先访问派生类中定义的成员函数



回顾：__init__函数的继承和重写

- __init__函数可以继承
- 如果__init__函数发生重写，派生类应负责全部数据的初始化

```
class CheckingAccount(Account):  
    def __init__(self, acc_holder, balance):  
        self.balance = balance  
        self.holder = acc_holder
```

```
class CheckingAccount(Account):  
    def __init__(self, acc_holder, balance):  
        Account.__init__(self, acc_holder)  
        self.balance = balance
```

```
super().__init__(acc_holder)
```

可以显式调用基类的初始化函数

或通过super函数找到该类的基类



用户自定义异常

- 继承Exception类，区分异常类型、存储相关的异常信息

```
class ZeroExc(Exception):  
    pass
```

```
try:  
    x = data_processing("10", 1, compute)  
    print("Results: ", x)  
except ZeroExc as ze:  
    print("handling error ",ze)  
except:  
    print("unhandled error!")
```



用户自定义异常

- 继承Exception类，区分异常类型、存储相关的异常信息

```
class DividingError(Exception):  
    def __init__(self, str, x, y):  
        self.x = x  
        self.y = y  
        super().__init__(str)
```

```
try:  
    x = data_processing(0, 1, compute)  
    print("Results: ", x)  
except DividingError as de:  
    print(de)  
    print("dividing {} by {}".format(de.x, de.y))  
except:  
    print("unhandled error!")
```



上下文管理器

- **with表达式**

- 计算<expr>的结果对象
- 将结果与<name>绑定
- 在进入<suite>时执行开始特定操作
 - 由 `__enter__` 约定
- 在结束<suite>时执行退出特定操作
 - 由 `__exit__` 约定

```
with <expr> as <name>:  
    <suite>
```

```
with open('foo.txt') as f:  
    # do something with f
```

https://docs.python.org/3/reference/compound_stmts.html#with



上下文管理器

- **一种用于描述运行时状态（上下文）的对象。**
 - 该对象处理进入和退出上述运行时状态（上下文）的代码块
 - 通常使用 with 语句调用，也可直接调用其方法来使用
- **上下文管理器的典型用法包括：**
 - 保存和恢复各种全局状态
 - 锁定和解锁资源
 - 打开关闭文件等

```
with <expr> as <name>:  
    <suite>
```

https://docs.python.org/3/reference/compound_stmts.html#with



*上下文管理器（with语句的实现）

```
with EXPRESSION as TARGET:  
    SUITE
```

实际保证了不管调用过程是否发生异常，都执行对应对象的exit方法，因而可以保证资源的回收/释放

```
manager = (EXPRESSION)  
enter = type(manager).__enter__  
exit = type(manager).__exit__  
value = enter(manager)  
hit_except = False  
  
try:  
    TARGET = value  
    SUITE  
except:  
    hit_except = True  
    if not exit(manager, *sys.exc_info()):  
        raise  
finally:  
    if not hit_except:  
        exit(manager, None, None, None)
```

```
class MyContextManager:
    def __init__(self, name):
        self.name = name
        print("in the constructor of", self.name)
    def __del__(self):
        print("in the destructor of", self.name)
    def __enter__(self):
        print("entering the process of", self.name)
    def __exit__(self, type, value, trace):
        print("leaving the process of", self.name)
```

```
with MyContextManager("myObj") as tc:
    print("during processing")
    raise Exception("Some Error")
print("period")
```

```
for i in range(5):
    tc = MyContextManager ( \
        "forObj" + str(i))
    print("comma", i)
    print("period2")
```



*上下文管理器（定义管理器对象）

- 管理多个上下文对象 v.s. 嵌套的with表达式
 - A、B都应该具有约定的实现

```
with A() as a, B() as b:  
    SUITE
```

```
with A() as a:  
    with B() as b:  
        SUITE
```

- Lib/contextlib.py
 - 继承AbstractContextManager类
 - 采用contextmanager装饰器

<https://docs.python.org/3/library/contextlib.html#module-contextlib>

```
from contextlib import contextmanager  
  
@contextmanager  
def managed_resource(*args, **kwargs):  
    # Code to acquire resource, e.g.:  
    resource = acquire_resource(*args, **kwargs)  
    try:  
        yield resource  
    finally:  
        # Code to release resource, e.g.:  
        release_resource(resource)  
  
>>> with managed_resource(timeout=3600) as resource:  
...     # Resource is released at the end of this block,  
...     # even if code in the block raises an exception
```

- **异常处理基本概念**
 - 错误和异常
 - 就地处理和异地处理
- **结构化异常处理机制**
 - raise try except else final
 - Exception类及其派生类、自定义异常
- **python中的异常**
 - 语法错误、类型相关异常、断言异常等
- **阅读资料**
 - <https://docs.python.org/3/tutorial/errors.html>