10.01 异常概述

CONTENTS

什么是异常

- 异常即是一个事件，该事件会在程序执行过程中发生，影响程序的正常执行。异常发生之后，异常之后的代码就不执行了。
- 异常事件在Python中是一个对象，表示一个错误。当Python脚本发生异常时需要捕获处理它，否则程序会终止执行。

什么是异常处理

Python解释器检测到错误，触发异常（也允许程序员自己触发异常），程序员编写特定的代码，专门用来捕捉这个异常（这段代码与程序逻辑无关，与异常处理有关），如果捕捉成功则进入另外一个处理分支，执行为其定制的逻辑，使程序不会崩溃，这就是异常处理。

为什么要进行异常处理

Python解析器去执行程序，检测到一个错误时，触发异常，异常触发后且没被处理的情况下，程序就在当前异常处终止，后面的代码不会运行。

所以必须提供一种异常处理机制，以此来增强程序的健壮性与容错性。

标准异常

- 常见异常

异 常 名 称	描 述
BaseException	所有异常的基类
SystemExit	解释器请求退出
KeyboardInterrupt	用户中断执行（通常是输入^C）
Exception	常规错误的基类
StopIteration	迭代器没有更多的值
GeneratorExit	生成器（generator）发生异常来通知退出
StandardError	所有的内建标准异常的基类
ArithmeticError	所有数值计算错误的基类

- 其他异常

异 常 名 称	描 述
FloatingPointError	浮点计算错误
OverflowError	数值运算超出最大限制
ZeroDivisionError	除（或取模）零（所有数据类型）
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EOFError	没有内建输入，到达 EOF 标记
EnvironmentError	操作系统错误的基类
IOError	输入/输出操作失败
OSError	操作系统错误
WindowsError	系统调用失败
ImportError	导入模块/对象失败
LookupError	无效数据查询的基类
IndexError	序列中没有此索引（index）
KeyError	映射中没有这个键
MemoryError	内存溢出错误（对于 Python 解释器不是致命的）
NameError	未声明/初始化对象（没有属性）
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法
SyntaxError	Python 语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用
SystemError	一般的解释器系统错误
TypeError	对类型无效的操作
ValueError	传入无效的参数
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeTranslateError	Unicode 转换时错误
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
FutureWarning	关于构造将来语义会有改变的警告
OverflowWarning	旧的关于自动提升为长整型的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
RuntimeWarning	可疑的运行时行为的警告
SyntaxWarning	可疑的语法的警告
UserWarning	用户代码生成的警告

异常处理

异常是由程序的错误引起的，语法上的错误跟异常处理无关，必须在程序运行前进行修正。

IF 判断方式处理异常

```
num = input('输入一个字符串试试 >>: ')      # 输入一个字符串试试
if num.isdigit():
    int(num)   # 正确程序，其余的都是异常处理
    print("输入的是 数字字符串，正确")
elif num.isspace():
    print('若输入的是空格，就执行此代码')
elif len(num)==0:
    print('若输入的是空，就执行此代码')
else:
    print("其他情况，执行次代码")
```

缺点

if 是可以解决异常的，只是存在以上两个问题：

01. if判断式的异常处理只能针对某一段代码，对于不同的代码段的相同类型的错误需要写重复的if来进行处理。
02. 在实际的程序中频繁地写与程序本身无关而与异常处理有关的if，会使得代码可读性降低。

```
def test():
    print("test running")

choice_dic={
    '1':test
}

while True:
    choice=input('>>: ').strip()
    if not choice or choice not in choice_dic: continue # 异常处理机制的一种
    choice_dic[choice_dic]()
```

TRY 捕获语法处理异常

基本语法

```
# format
try:
    code    # 运行别的代码
except<name>:
    code    # 如果在 try 部分引发了异常
except<name>,<data>:
    code    # 如果引发了异常，获得附加的数据
else:
    code    # "try 内代码块没有异常则执行 else 代码"
finally:
    code    # 无论异常与否，都会执行该模块，通常是进行清理工作

```

try 的工作原理

当开始一个 try 语句后， Python 就在当前程序的上下文中做标记，这样当异常出现时就可以回到这里，try 字句先执行，接下来会发生什么依赖于执行时是否出现异常。

如果当try后的语句执行时发生异常，Python就跳回到try并执行第一个匹配该异常的except子句，异常处理完毕，控制流就通过整个try语句（除非在处理异常时又引发新的异常）。

如果在try后的语句里发生了异常，却没有匹配的except子句，异常将被递交到上层的try，或者到程序的最上层（这样将结束程序，并打印默认的出错信息）。

如果在try子句执行时没有发生异常，Python将执行else语句后的语句（如果有else的话），然后控制流通过整个try语句。

例子

- 打开一个文件，在该文件中的写入内容

```
try:
    fh = open("testfile","w")
    fh.write("这是一个测试文件，用于测试异常！！！")
except IOError:
    print("Error: 没有找到文件或 读取文件失败")
else:
    print("内容写入文件成功")
    fh.close()
```

```
# Output

内容写入文件成功
```

总结

try…except这种异常处理机制就是取代if，让程序在不牺牲可读性的前提下增强健壮性和容错性。

异常处理中为每一个异常定制了异常类型（Python中统一了类与类型，类型即类），对于同一种异常，一个except就可以捕捉到，可以同时处理多段代码的异常（无须写多个if判断式），减少了代码，增强了可读性。