

## 导引问题

在实际工作中，我们遇到的情况不可能是非常完美的。比如：你写的某个模块，用户输入不一定符合你的要求；你的程序要打开某个文件，这个文件可能不存在或者文件格式不对；你要读取数据库的数据，数据可能是空的；我们的程序再运行着，但是内存或硬盘可能满了等等。

软件程序在运行过程中，非常可能遇到刚刚提到的这些问题，我们称之为异常，英文是：Exception，意思是例外。遇到这些例外情况，或者叫异常，我们怎么让写的程序做出合理的处理，而不至于程序崩溃呢？我们本章就要讲解这些问题。

如果我们要拷贝一个文件，在没有异常机制的情况下，我们需要考虑各种异常情况，伪代码如下：

### 【示例 6-1】伪代码使用 if 处理程序中可能出现的各种情况

```
#将 d:/a.txt 拷贝到 e:盘  
if "d:/a.txt"这个文件存在：  
    if e 盘的空间大于 a.txt 文件长度：  
        if 文件复制一半 IO 流断掉：  
            停止 copy，输出：IO 流出问题！  
        else:  
            copyFile("d:/a.txt", "e:/a.txt")  
    else:  
        print("e 盘空间不够存放 a.txt！")
```

```
else:  
    print("a.txt 不存在!")
```

**这种方式，有两个坏处：**

1. 逻辑代码和错误处理代码放一起!
2. 程序员本身需要考虑的例外情况较复杂，对程序员本身要求较高!

那么，我们如何解决应对异常情况呢? python 的异常机制给我们提供了方便的处理方式。如上情况，如果是用 python 的异常机制来处理，示意代码如下(仅限示意，不能运行)：

```
#将 d:/a.txt 复制到 e:盘  
try:  
    copyFile("d:/a.txt", "e:/a.txt")  
except:  
    print("文件无法拷贝")
```

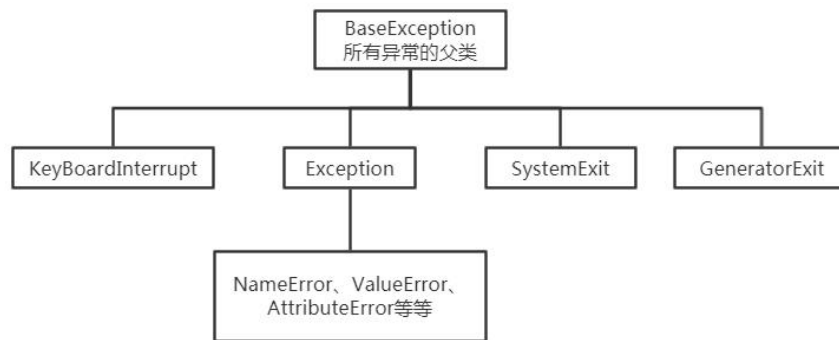
## 异常机制本质

异常指程序运行过程中出现的非正常现象，例如用户输入错误、除数为零、需要处理的文件不存在、数组下标越界等。

所谓异常处理，就是指程序在出现问题时依然可以正确的执行剩余的程序，而不会因为异常而终止程序执行。

python 中，引进了很多用来描述和处理异常的类，称为异常类。异常类定义中包含了该类异常的信息和对异常进行处理的方法。下面较为完整的展示了

python 中内建异常类的继承层次：



我们处理一下，遇到的第一个异常：

```
#测试简单的 0 不能做除数异常

a = 3/0
```

执行上面程序，控制台打印：

```
Traceback (most recent call last):

  File "C:/Users/Administrator/PycharmProjects/mypro_exception/my01.py", line 1, in <module>

    a = 3/0

ZeroDivisionError: division by zero

Process finished with exit code 1
```

**python 中一切都是对象，异常也采用对象的方式来处理。处理过程：**

1. **抛出异常**：在执行一个方法时，如果发生异常，则这个方法生成代表该异常的一个对象，停止当前执行路径，并把异常对象提交给解释器。

2. **捕获异常**：解释器得到该异常后，寻找相应的代码来处理该异常。

## 解决异常问题的态度

学习完异常相关知识点，只是开始对异常有些认识，不意味着你会调试任何异常；调试异常，需要大量的经验作为基础。因此，大家不要在此停留，继续往后学习。碰到每个异常，都要花心思去解决而不要动不动张口问人。通过自己的努力无法解决，再去找老师同学帮助解决。

解决每一个遇到的异常，建议大家遵循如下三点：

1. 不慌张，细看信息，定位错误。看清楚报的错误信息，并定位发生错误的地方。
2. 百度并查看十个相关帖子。将异常类信息进行百度，至少查看十个以上的相关帖子。
3. 以上两步仍然无法解决，找老师和同学协助解决。

正常情况，自己遵循如上步骤解决 30 个以上的错误，就能积累初步的调试经验，以后遇到的大部分错误都能独立完成。

当遇到任何一个问题，  
我第一个想到的是：  
度娘



## 异常解决的关键：定位

当发生异常时，解释器会报相关的错误信息，并会在控制台打印出相关错误信息。我们只需按照从上到下的顺序即可追溯（Trackback）错误发生的过程，最终定位引起错误的那一行代码。

【示例】追溯异常发生的过程

```
def a():  
    print("run in a() start! ")  
    num = 1/0  
    print("run in a() end! ")  
def b():  
    print("run in b() start!")  
    a()
```

```
print("run in b() end! ")

def c():
    print("run in c() start!")
    b()
    print("run in c() end! ")

print("step1")
c()
print("step2")
```

控制台打印结果:

```
step1
run in c() start!
run in b() start!
run in a() start!
Traceback (most recent call last):
  File "C:/Users/Administrator/PycharmProjects/mypro_exception/my01.py", line 17, in <module>
    c()
  File "C:/Users/Administrator/PycharmProjects/mypro_exception/my01.py", line 12, in c
    b()
  File "C:/Users/Administrator/PycharmProjects/mypro_exception/my01.py", line 7, in b
    a()
  File "C:/Users/Administrator/PycharmProjects/mypro_exception/my01.py", line 3, in a
    num = 1/0
ZeroDivisionError: division by zero

Process finished with exit code 1
```

## try...一个 except 结构

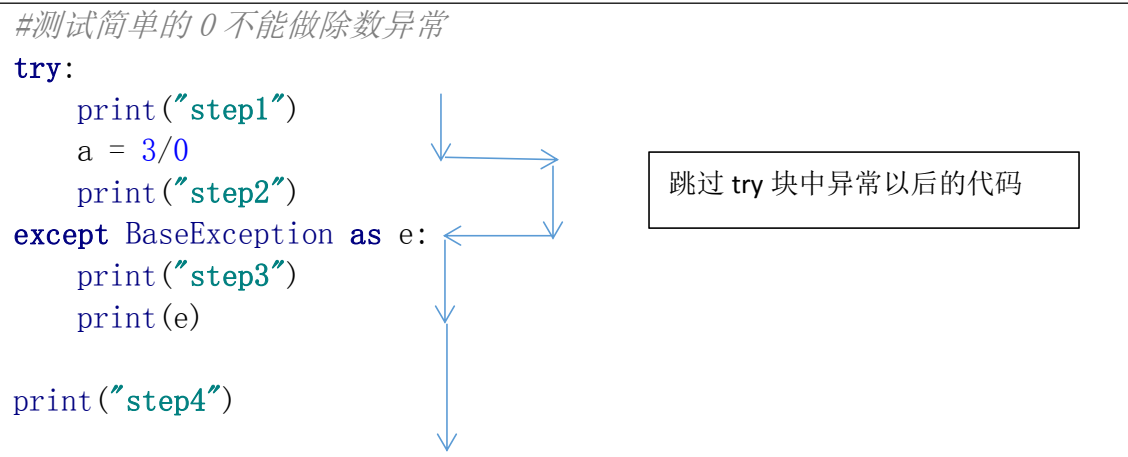
try...except 是最常见的异常处理结构。结构如下:

```
try:
    被监控的可能引发异常的语句块
except BaseException [as e]:
    异常处理语句块
```

try 块包含着可能引发异常的代码，except 块则用来捕捉和处理发生的异常。执行的时候，如果 try 块中没有引发异常，则跳过 except 块继续执行后续代码；执行的时候，如果 try

块中发生了异常，则跳过 try 块中的后续代码，跳到相应的 except 块中处理异常；异常处理完后，继续执行后续代码。

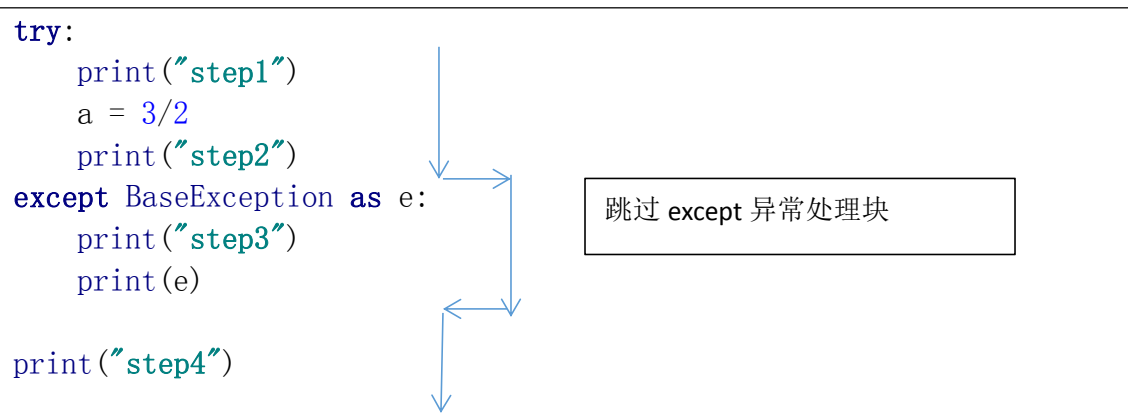
• 遇到异常的执行顺序：



如上的执行结果是：

```
step1
step3
division by zero
step4
```

我们可以看到，程序执行到引发异常的语句时，直接掉到 except 异常处理语句块进行处理；处理完毕后，继续执行下面的程序。



如上代码的执行结果是：

```
step1
step2
step4
```

try 语句块没有发生异常，则正常执行完 try 块后跳过 except 异常处理语句块，继续往下执行。

【示例】循环输入数字，如果不是数字则处理异常；直到输入 88，则结束循环。

```
while True:
    try:
        x = int(input("请输入一个数字："))
        print("您入的数字是", x)
        if x==88:
            print("退出程序")
            break
    except:
        print("异常：输入的不是一个数字")
```

执行结果：

请输入一个数字：10

您入的数字是 10

请输入一个数字：abc

异常：输入的不是一个数字

请输入一个数字：88

您入的数字是 88

退出程序

## try...多个 except 结构

上面的结构可以捕获所有的异常，工作中也很常见。但是，从经典理论考虑，一般建议尽量捕获可能出现的多个异常（按照先子类后父类的顺序），并且针对性的写出异常处理代码。为了避免遗漏可能出现的异常，可以在最后增加 `BaseException`。结构如下：

try:

被监控的、可能引发异常的语句块

except Exception1:

处理 Exception1 的语句块

except Exception2:

处理 Exception2 的语句块

...

except BaseException:

处理可能遗漏的异常的语句块

【示例】多个 except 结构

```
try:
    a = input("请输入被除数：")
```

```
b = input("请输入除数：")
c = float(a)/float(b)
print(c)

except ZeroDivisionError:
    print("异常：除数不能为 0")
except TypeError:
    print("异常：除数和被除数都应该为数值类型")
except NameError:
    print("异常：变量不存在")
except BaseException as e:
    print(e)
    print(type(e))
```

执行结果：

请输入被除数：10

请输入除数：0

异常：除数不能为 0

## try...except...else 结构

try...except...else 结构增加了“else 块”。如果 try 块中没有抛出异常，则执行 else 块。如果 try 块中抛出异常，则执行 except 块，不执行 else 块。

【示例】try...except...else 结构执行测试

```
try:
    a = input("请输入被除数：")
    b = input("请输入除数：")
    c = float(a)/float(b)
except BaseException as e:
    print(e)
else:
    print("除的结果是：", c)
```

发生异常的执行情况（执行 except 块，没有执行 else）：

请输入被除数：5

请输入除数：0

float division by zero

没有发生异常的执行情况（执行完 try 块后，执行 else）：



请输入被除数：10  
请输入除数：5  
除的结果是： 2.0

## try...except...finally 结构

try...except...finally 结构中，finally 块无论是否发生异常都会被执行；通常用来释放 try 块中申请的资源。

【示例】try...except...finally 结构简单测试

```
try:
    a = input("请输入一个被除数：")
    b = input("请输入一个除数：")
    c = float(a)/float(b)
except BaseException as e:
    print(e)
else:
    print(c)
finally:
    print("我是 finally 中的语句，无论发生异常与否，都执行！")

print("程序结束！")
```

执行结果如下：

请输入被除数：10  
请输入除数：0  
float division by zero  
我是 finally 中的语句，无论是否发生异常都执行

【示例】读取文件，finally 中保证关闭文件资源

```
try:
    f = open("d:/a.txt", 'r')
    content = f.readline()
    print(content)
except BaseException as e:
    print(e)
finally:
    f.close()      #释放资源。此处也可能会发生异常。若发生异常，则程序终止，不会继续往下执行

print("step4")
```

发生异常的执行结果:

Traceback (most recent call last):

[Errno 2] No such file or directory: 'd:/a.txt'

File "C:/PycharmProjects/mypro\_exception/my01.py", line 8, in <module>

f.close() #释放资源。此处也可能会发生异常。若发生异常,则程序终止,不会继续往下执行

NameError: name 'f' is not defined

Process finished with exit code 1

## return 语句和异常处理问题

由于 `return` 有两种作用: 结束方法运行、返回值。我们一般不把 `return` 放到异常处理结构中, 而是放到方法最后。

【示例】`return` 和异常结构的正确处理方式

```
def test01():
    print("step1")
    try:
        x = 3/0
        # return "a"
    except:
        print("step2")
        print("异常: 0 不能做除数")
        #return "b"
    finally:
        print("step4")
        #return "d"

    print("step5")
    return "e" #一般不要将 return 语句放到 try、except、else、
              #finally 块中, 会发生一些意想不到的错误。建议放到方法最后。
print(test01())
```

执行结果:

step1

step2

异常：0 不能做除数

step4

step5

e

## 常见异常的解决

Python 中的异常都派生自 `BaseException` 类，本节我们测试和列出常见的一些异常，方便初学者掌握。

### 1. `SyntaxError`: 语法错误

<code>int a = 3</code>
<code>int a =3</code> <sup>^</sup>
<code>SyntaxError: invalid syntax</code>

### 2. `NameError`: 尝试访问一个没有声明的变量

<code>print(a)</code>
<code>print(a)</code> <code>NameError: name 'a' is not defined</code>

### 3. `ZeroDivisionError`: 除数为 0 错误（零除错误）

<code>a = 3/0</code>
<code>a = 3/0</code> <code>ZeroDivisionError: division by zero</code>

### 4. `ValueError`: 数值错误

<code>float("gaoqi")</code>
<code>float("gaoqi")</code> <code>ValueError: could not convert string to float: 'gaoqi'</code>

### 5. `TypeError`: 类型错误

<code>123+"abc"</code>
<code>123+"abc"</code> <code>TypeError: unsupported operand type(s) for +: 'int' and 'str'</code>

### 6. `AttributeError`: 访问对象的不存在的属性

a=100 a.sayhi()
a.sayhi() AttributeError: 'int' object has no attribute 'sayhi'

#### 7. IndexError: 索引越界异常

a = [4, 5, 6] a[10]
a[10] IndexError: list index out of range

#### 8. KeyError: 字典的关键字不存在

a = {'name': "gaoqi", 'age': 18} a['salary']
a['salary'] KeyError: 'salary'

## 常见异常汇总

建议大家通读，把异常相关的单词背下来熟悉一下。这样可以克服“畏难情绪”。

异常名称	说明
ArithmeticError	所有数值计算错误的基类
AssertionError	断言语句失败
AttributeError	对象没有这个属性
BaseException	所有异常的基类
DeprecationWarning	关于被弃用的特征的警告
EnvironmentError	操作系统错误的基类
EOFError	没有内建输入, 到达 EOF 标记
Exception	常规错误的基类
FloatingPointError	浮点计算错误
FutureWarning	关于构造将来语义会有改变的警告
GeneratorExit	生成器(generator)发生异常来通知退出
ImportError	导入模块/对象失败
IndentationError	缩进错误
IndexError	序列中没有此索引(index)
IOError	输入/输出操作失败

KeyboardInterrupt	用户中断执行(通常是输入^C)
KeyError	映射中没有这个键
LookupError	无效数据查询的基类
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)
NotImplementedError	尚未实现的方法
OSError	操作系统错误
OverflowError	数值运算超出最大限制
OverflowWarning	旧的关于自动提升为长整型(long)的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
ReferenceError	弱引用(Weak reference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
RuntimeWarning	可疑的运行时行为(runtime behavior)的警告
StandardError	所有的内建标准异常的基类
StopIteration	迭代器没有更多的值
SyntaxError	Python 语法错误
SyntaxWarning	可疑的语法的警告
SystemError	一般的解释器系统错误
SystemExit	解释器请求退出
TabError	Tab 和空格混用
TypeError	对类型无效的操作
UnboundLocalError	访问未初始化的本地变量
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeError	Unicode 相关的错误
UnicodeTranslateError	Unicode 转换时错误
UserWarning	用户代码生成的警告
ValueError	传入无效的参数
Warning	警告的基类
WindowsError	系统调用失败
ZeroDivisionError	除(或取模)零 (所有数据类型)

## with 上下文管理

finally 块由于是否发生异常都会执行，通常我们放释放资源的代码。其实，我们可以通过 with 上下文管理，更方便的实现释放资源的操作。

with 上下文管理的语法结构如下：

```
with context_expr [ as var]:  
    语句块
```

**with** 上下文管理可以自动管理资源，在 **with** 代码块执行完毕后自动还原进入该代码之前的现场或上下文。不论何种原因跳出 **with** 块，不论是否有异常，总能保证资源正常释放。极大的简化了工作，在文件操作、网络通信相关的场合非常常用。

【示例】with 上下文管理文件操作

```
with open("d:/bb.txt") as f:  
    for line in f:  
        print(line)
```

执行结果：

```
gaoqi  
sxt  
baizhan
```

## traceback 模块

【示例】使用 Traceback 模块打印异常信息

```
#coding=utf-8  
import traceback  
  
try:  
    print("step1")  
    num = 1/0  
except:  
    traceback.print_exc()
```

运行结果：

```
step1
```

Traceback (most recent call last):

```
File "C:/Users/Administrator/PycharmProjects/mypro_exception/my01.py", line 7, in <module>  
    num = 1/0
```

ZeroDivisionError: division by zero

Process finished with exit code 0

【示例】使用 traceback 将异常信息写入日志文件

```
#coding=utf-8
import traceback

try:
    print("step1")
    num = 1/0
except:
    with open("d:/a.log","a") as f:
        traceback.print_exc(file=f)
```

## 自定义异常类

程序开发中，有时候我们也需要自己定义异常类。自定义异常类一般都是运行时异常，通常继承 `Exception` 或其子类即可。命名一般以 `Error`、`Exception` 为后缀。

自定义异常由 `raise` 语句主动抛出。

【示例】自定义异常类和 `raise` 语句

```
#coding=utf-8
#测试自定义异常类

class AgeError(Exception): #继承 Exception
    def __init__(self, errorInfo):
        Exception.__init__(self)
        self.errorInfo = errorInfo
    def __str__(self):
        return str(self.errorInfo)+"，年龄错误！应该在 1-150 之间"

#####测试代码#####
if __name__ == "__main__": #如果为 True，则模块是作为独立文件运行，
    可以执行测试代码
    age = int(input("输入一个年龄："))
    if age<1 or age>150:
        raise AgeError(age)
    else:
        print("正常的年龄：", age)
```

执行结果如下：

输入一个年龄:200

Traceback (most recent call last):

File "C:/Users/Administrator/PycharmProjects/mypro\_exception/my10.py", line 16, in <module>

raise AgeError(age)

\_\_main\_\_.AgeError: 200,年龄错误! 应该在 1-150 之间

## Pycharm 开发环境的调试

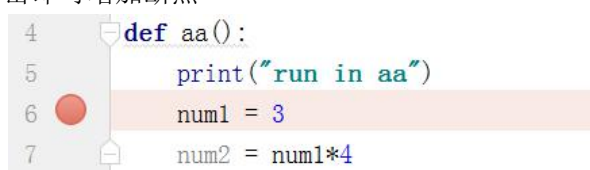
进行调试的核心是设置断点。程序执行到断点时，暂时挂起，停止执行。就像看视频按下停止一样，我们可以详细的观看停止处的每一个细节。

### 断点

程序运行到此处，暂时挂起，停止执行。我们可以详细在此时观察程序的运行情况，方便做出进一步的判断。

#### 1. 设置断点:

- (1) 在行号后面单击即可增加断点



- (2) 在断点上再单击即可取消断点

### 进入调试视图

我们通过如下三种方式都可以进入调试视图:

- (1) 单击工具栏上的按钮: 
- (2) 右键单击编辑区，点击: debug '模块名'
- (3) 快捷键: shift+F9

进入调试视图后，布局如下:






左侧为“浏览帧”：

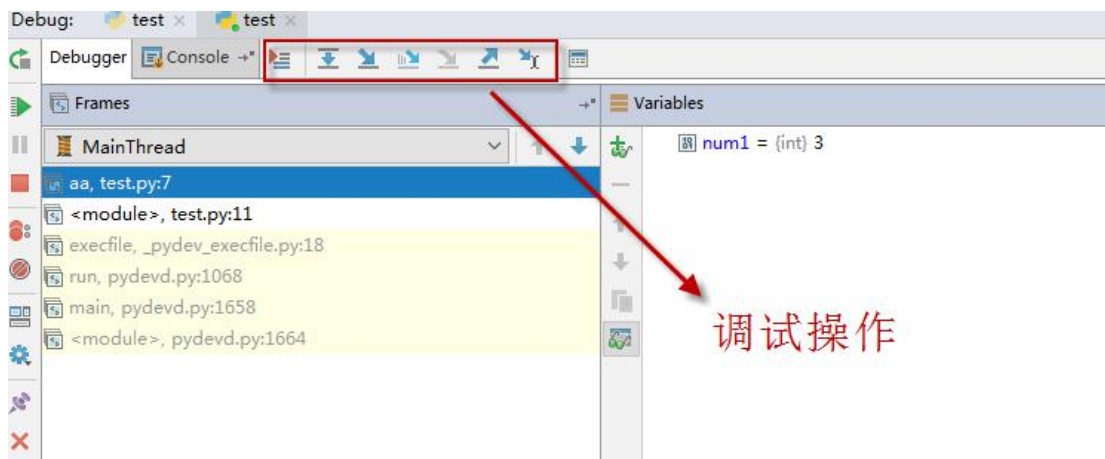
调试器列出断点处，当前线程正在运行的方法，每个方法对应一个“栈帧”。最上面的是当前断点所处的方法。

变量值观察区：

调试器列出了断点处所在方法相关的变量值。我们可以通过它，查看变量的值的变化。

也可以通过 ，增加要观察的变量。

## 调试操作区



我们通过上图中的按钮进行调试操作，它们的含义如下：

中文名称	英文名称	图标和快捷键	说明
显示当前所有断点	show Execution Point	 Alt+F10	
单步调试： 遇到函数跳过	step over	 F8	若当前执行的是一个函数，则会把这个函数当做整体一步执行完。不会进入这个函数内部

单步调试： 遇到函数进入	step into	 F7	若当前执行的是一个函数，则会进入这个函数内部
跳出函数	step out	 Shift+F8	当单步执行到子函数内时，用 step out 就可以执行完子函数余下部分，并返回到上一层函数
执行的光标处	run to cursor	 Alt+F9	一直执行，到光标处停止，用在循环内部时，点击一次就执行一个循环