10. 异常

程序运行过程中，由于程序本身设计问题或者外界环境改变而引发的错误称为异常。引发异常的原因有很多，如 下标越界 、 文件不存在 、 网络异常 、 数据类型错误 等。如果这些异常得不到正确处理就会导致程序终止运行，而合理地使用异常处理可以使得程序更加健壮，并具有更强的容错性。

10.1. 语法错误和异常

在刚学习 Python 编程的时候，经常会遇到到一些报错信息。Python 中有两种常见的错误： 语法错误 和 异常 。

1. 语法错误

   语法错误也称为**解析错误**，是初学者经常会遇到的问题，例如：

   ```
   a = 1
   if a == 1 print("Yes")
   ```

   得到一个`SyntaxError:invalid syntax`的提示。

提示语法错误时，会首先打印出出现语法错误的语句，然后在离语法错误最近的位置标记一个小小的箭头`^`，该例子中，`print()`函数被检查到有错误，是它前面缺少了一个冒号`:`。

2. 异常

即使 Python 程序的语法是正常的，在运行时也有可能发生错误，**这种在运行期间检测到的错误称为 异常 。**

例如：

```
a = '2' + 2
```

会抛出异常`TypeError`异常。

10.2. Python 中的异常

在前面的章节中，每次执行程序遇到异常时，如果没有对该异常对象进行处理和捕获，程序就会用所谓的回溯（Traceback，一种错误信息）终止执行，这些信息包括错误的名称（如`TypeError`）、原因和错误发生的行号。以下为 10 个常见异常。

1. TypeError

当将不同的类型的数据进行运算操作时，有时会引发 TypeError（不同类型间的无效操作）异常。

```
birth = input('birth:')
if birth < 2000:
    print('00前')
else:
    print('00后')
```

上述例子中，`input()`函数返回的数据类型是字符串，当与`2000`进行比较时，出现如下错误信息：

```
birth:1998
Traceback(most recent call last):
    File "1.py",line 2, in <module>
        if birth < 2000:
TypeError: '<' not supported between
instances of 'str' and 'int'
```

`str`型数据不能直接与`int`型数据进行比较。

2. ZeroDivisionError

当除数为零时，会引发`ZeroDivisionError`（除数为零）异常。

```
print(1 / 0)
```

错误信息：

```
Traceback (most recent call last):
  File "E:/PythonProject/C201004/1.py",
line 1, in <module>
    print(1/0)
ZeroDivisionError: division by zero
```

除数不能为零。

3. NameError

当尝试访问一个未声明的变量时，会引发`NameError`（尝试访问一个不存在的变量）异常。

```
a = 1
c = a + b
print(c)
```

错误信息：

```
Traceback (most recent call last):
  File "E:/PythonProject/C201004/1.py",
line 2, in <module>
    c = a + b
NameError: name 'b' is not defined
```

解释器在任何命名空间里都没有找到变量`b`。

4. SyntaxError

当解释器发现语法错误时，会引发`SyntaxError`（Python 语法错误）异常。

```
list_1 = [1, 2, 3, 4]
for i in list_1
    print(i)
```

由于`for`循环的后面缺少冒号，所以导致程序出现如下错误：

```
  File "E:/PythonProject/C201004/1.py",
line 2
    for i in list_1
                  ^
SyntaxError: invalid syntax
```

在`list_1`附近有一个错误，当遇到`SyntaxError`时，Python 代码并不能继续执行，需要先找到并改正错误。

5. IndentationError

Python 最具特色的就是依靠代码块的缩进来体现代码之间逻辑关系，当缩进错误时，会引发`IndentationError`（缩进错误）异常。

```
list_1 = [1, 2, 3, 4]
for i in list_1:
print(i)
```

由于`for`循环语句块没有缩进，所以导致程序出现如下错误：

```
  File "E:/PythonProject/C201004/1.py",
line 3
    print(i)
    ^
IndentationError: expected an indented
block
```

`print(i)`语句缩进错误。

6. IndexError

当使用序列中不存在的索引时，会引发`IndexError`（索引超出序列的范围）异常。

```
list_1 = [1, 2, 3, 4]
print(list_1[4])
```

`list_1`列表中没有索引为`4`的元素，使用索引`4`访问列表元素时，出现如下错误：

```
Traceback (most recent call last):
  File "E:/PythonProject/C201004/1.py",
line 2, in <module>
    print(list_1[4])
IndexError: list index out of range
```

列表的索引值超出了列表的范围。

7. KeyError

当使用字典中不存在的键时，会引发`KeyError`（字典中查找一个不存在的关键字）异常。

```
dict_1 = {'one': 1, 'two': 2}
print(dict_1['one'])
print(dict_1['three'])
```

字典中只有`one`和`two`两个键，获取`three`键对应的值时，出现如下错误：

```
Traceback (most recent call last):
  File "E:/PythonProject/C201004/1.py",
line 3, in <module>
    print(dict_1['three'])
KeyError: 'three'
1
```

使用了字典中没有的键。

8. ValueError

当传给函数的参数类型不正确时，会引发`ValueError`（传入无效参数）异常。

```
a = int('b')
```

错误信息：

```
Traceback (most recent call last):
  File "E:/PythonProject/C201004/1.py",
line 1, in <module>
    a = int('b')
ValueError: invalid literal for int() with
base 10: 'b'
```

`int()`函数传入了无效的字符串型参数。

9. FileNotFoundError

   当试图用只读方式打开一个不存在的文件时，会引发`FileNotFoundError`（Python 3.2以前是`IOError`）异常。

   ```
   file = open('1.txt')
   ```

   使用`open()`方法打开名为`1.txt`的文件，当该文件不存在时：

   ```
   Traceback (most recent call last):
     File "E:/PythonProject/C201004/1.py",
   line 1, in <module>
       file = open('1.txt')
   FileNotFoundError: [Errno 2] No such file
   or directory: '1.txt'
   ```

   没有找到名为`1.txt`的文件或目录。

10. AttributeError

    当尝试访问未知的对象属性时，会引发`AttributeError`（尝试访问未知的对象属性）异常。

```
class Car:
    color = 'black'
car = Car()
print(car.color)
print(car.name)
```

`Car`没有定义`name`属性，在创建`Car`类的实例后，访问它的`name`属性时，报错：

```
black
Traceback (most recent call last):
  File "E:/PythonProject/C201004/1.py",
line 5, in <module>
    print(car.name)
AttributeError: 'Car' object has no
attribute 'name'
```

在`Car`类中没有定义`name`属性，所以访问`name`属性时出现异常。

10.3. 异常检测和处理

Python 提供了多种不同形式的异常处理结构，思路基本一致：首先尝试运行代码，没有问题就是正常执行，如果发生错误就尝试捕获和处理异常。

10.3.1. try-except

1. 捕获单个异常

   try-except 语句用于检测和处理异常

```
try:
    # 可能发生异常的代码块
except:
    # 出现异常后执行的代码块
```

其中，Exception 为异常类的名称。该结构类似于**单分支选择结构**，其中 try子句 中的代码块包含可能会引发异常的子句，而 except子句 则用来捕获相应的异常。如果 try子句 中的代码引发异常并被 except子句 捕获，就执行 except子句 的代码块；如果 try 中的代码块没有出现异常，就继续往下执行异常处理结构后面的代码；如果出现的异常没有被 except 捕获并处理，程序崩溃并将该异常呈现给用户。

【示例】捕获两数相除除数为`0`的异常。

```
try:
    a = float(input('请输入被除数：'))
    b = float(input('请输入除数：'))
    c = a / b
    print('商为：', c)
except ZeroDivisionError:
    print('除数不能为0！')
```

【说明】

在 try子句 中要求输入被除数和除数，当除数为`0`时，程序引发`ZeroDivisionError`异常，此时，except子句 就会捕获这个异常并输出`除数不能为0！`。程序产生异常时，不会再出现终止程序的情况，而是将程序中设定的提示信息输出。

> 运行程序时，只要捕获到异常，程序就会执行`except`子句，并且不会再执行`try`子句中未执行的语句。

2. 针对不同异常设置多个`except`

```
try:
    a = float(input('请输入被除数：'))
    b = float(input('请输入除数：'))
    c = a / b
    print('商为：', c)
except ZeroDivisionError:
    print('除数不能为0！')
```

在运行上述的代码时，如果输入的为**非数字类型**的值，就会产生另一个传入无效参数的异常。

```
请输入被除数：a
Traceback (most recent call last):
  File "E:/PythonProject/C201004/1.py",
line 2, in <module>
    a = float(input('请输入被除数：'))
ValueError: could not convert string to
float: 'a'

Process finished with exit code 1
```

上述信息表明，由于输入了一个无效参数，导致程序出现`ValueEirror`异常。这是因为代码中的`except`语句只能捕获`ZeroDivisionError`异常，程序没有处理新异常的语句而导致终止运行。为了让程序能够检测到`ValueError`异常，可以再增加一个处理该异常的`except`语句。此时，需要用到**处理多个异常**的`try-except`语句，其语法格式如下：

```
try:
    # 可能发生异常的代码块
except Exception1:
    # 处理异常类型 1 的代码块
except Exception2:
    # 处理异常类型 2 的代码块
except Exception3:
    # 处理异常类型 3 的代码块
    ...
```

【改进】

```
try:
    a = float(input('请输入被除数：'))
    b = float(input('请输入除数：'))
    c = a / b
    print('商为：', c)
except ZeroDivisionError:
    print('除数不能为0！')
except ValueError:
    print('除数和被除数应为数值类型！')
```

3. 对多个异常统一处理

在实际开发中，有时会为几种不同的异常设计相同的异常处理代码。为减少代码量，Python 允许将多个异常类型放到一个元组中，然后使用一个`except`子句同时捕捉多种异常，并且共用同一段异常处理代码。

【示例】将 2 中两个异常合并到一个`except`子句中。

```
try:
    a = float(input('请输入被除数:'))
    b = float(input('请输入除数:'))
    c = a / b
    print('商为:', c)
except (ZeroDivisionError, ValueError):
    print('捕获到异常！')
```

无论捕获到的是哪种异常，都打印出一样的错误信息，这样我们无法从打印出的信息中获取有效信息。为了区分不同的错误信息，可以使用`as`获取系统反馈的错误信息。

【示例】捕获异常的描述信息。

```
try:
    a = float(input('请输入被除数:'))
    b = float(input('请输入除数:'))
    c = a / b
    print('商为:', c)
except (ZeroDivisionError, ValueError) as
r:
    print(f'捕获到异常：{r}')
```

当监控到`ZeroDivisionError`或者`ValueError`异常，就会将异常描述信息保存到变量`r`中，然后进行输出。

4. 捕获所有异常

如果无法确定要对哪一类异常进行处理，只是希望在`try`语句块出现任何异常时，都给用户一个提示信息，那么可以在`except`子句中不指明异常类型。

【示例】捕获所有异常

```
try:
    a = float(input('请输入被除数:'))
    b = float(input('请输入除数:'))
    c = a / b
    print('商为:', c)
except:
    print('出错啦！')
```

由于 Exception 类是左右异常类的父类，因此，还可以在 except 语句后使用 Exception 类表示将所有异常捕获。

【示例】

```
try:
    a = float(input('请输入被除数:'))
    b = float(input('请输入除数:'))
    c = a / b
    print('商为:', c)
except Exception as r:
    print('捕获到异常: {}'.format(r))
```

10.3.2. try-except-else

`try-except`语句还有一个可选的`else`子句，如要使用该子句，必须将其放在所有`except`子句之后。该子句将在`try`子句没有发生任何异常时执行。该结构的语法格式如下:

```
try:
    #可能会引发异常的代码块
except Exception [as reason]:
    #出现异常后执行的代码块
else:
    #如果 try 子句中的代码没有引发异常，则执行该代码块
```

【示例】以只读方式打开文件，并统计文件中文本的行数，如果文件不存在则给出提示信息。

```
arg = '1.txt'
try:
    f = open(arg, 'r')
except FileNotFoundError:
    print(arg,'文件不存在')
else:
    print(arg, '文件有', len(f.readlines()),
'行')
    f.close()
```

> 提示
>
> 建议在 try 子句中只放真的有可能会引发异常的代码，将其余代码放在 else 子句中。

10.3.3. try-except-finally

运行以下代码时，如果`1.txt`文件存在，`open()`函数正常返回文件对象，但异常却发生在成功打开文件后的`print(a)`语句上，此时 Python 将直接执行 except 语句。也就是说，文件打开了，但并没有执行关闭文件的操作。

```
try:
    f = open('1.txt')
    print(a)
    f.close()
except:
    print('出错啦！')
```

在程序中，类似上述情况，无论是否捕获到异常，都需要执行一些终止行为（如关闭文件），Python 引入了 finally 子句来扩展 try，该结构的语法格式如下：

```
try:
    #可能会引发异常的代码块
except Exception [as reason]:
    #出现异常后执行的代码块
finally:
    #无论 try 子句中的代码有没有引发异常，都会执行的代码块
```

【示例】

```
try:
    f = open('1.txt')
    print(a)
except:
    print('出错啦！')
finally:
    f.close()
```

【程序说明】如果 try 子句中没有出现异常，会跳过 except 子句执行 finally 子句的内容。如果出现异常，则会先执行 except 子句的内容，再执行 finally 子句的内容。**总之，finally子句中的内容就是无论如何都要被执行的内容。**

> 提示
>
> 异常处理结构不是万能的，并不是采用了异常处理结构就万事大吉了， finally子句中的代码也可能会引发异常。例如，运行上述示例时，如果“1.txt”文件不存在，就会在 finally 子句中关闭文件时引发异常。

10.3.4. 同时包含多个 except、 else 和 finally 子句

Python 异常处理结构中可以同时包含多个`except`子句、`else`子句和`finally`子句，其语法格式如下:

```
try:
    #可能会引发异常的代码块
except Exception1:
    #处理异常类型 1 的代码块
except Exception2:
    #处理异常类型 2 的代码块
else:
    #如果 try 子句中的代码没有引发异常，则执行该代码块
finally:
    #无论 try 子句中的代码有没有引发异常，都会执行的代码块
```

【示例】同时包含多个 except、else 和 finally 子句的异常处理。

```
try:
    a = float(input('请输入被除数：'))
    b = float(input('请输入除数：'))
    c = a/b
    print('商为:',c)
except ZeroDivisionError:
    print('除数不能为0！')
except ValueError:
    print('被除数和除数应为数值类型！')
except:
    print('其他错误！')
else:
    print('运行没有错误！')
```

```
finally:
    print('运行结束！')
```

【程序说明】首先执行 try 子句中的语句块，如果发生异常，则中断当前在 ty 子句中的执行，跳转到对应的异常处理块中开始执行；如果没有发生异常，则程序在执行完 ty子句后会进入 else 子句中执行；无论是否发生异常，程序执行的最后一步总是执行 finally子句中的语句块。

> 提示
>
> （1）在上述语句中，异常处理结果必须以“try”→“except”→“else”→“finally”的顺序出现，即所有的 except 必须在 else 和 finally 之前，else 必须在 finally 之前，否则会出现语法错误。
>
> （2）else 和 finally 都是可选的。
>
> （3）else 的存在必须以 except 语句为前提。就是说，如果在没有 except 语句的 try 语句中使用 else 语句会引发语法错误。

10.4. 抛异常

在 Python 中，程序运行出现错误时会引发异常。在程序中主动抛出异常，可以使用`raise`和`assert`语句。

10.4.1. raise 语句

Python 使用`raise`语句抛出一个指定异常。

1. 使用异常名引发异常

   语法：

   ```
   raise 异常名称
   ```

   当`raise`语句指定异常名时，会创建该类的实例对象，然后引发异常。

   ```
   raise NameError
   ```

   结果：

   ```
   Traceback (most recent call last):
     File "E:/PythonProject/C200105/1.py",
   line 1, in <module>
       raise NameError
   NameError
   ```

2. 使用异常类的实例引发异常

```
raise 异常名称('命名描述')
```

通过显式地创建异常类的实例，直接使用该实例对象来引发异常，同时还能给异常类指定描述信息。

```
raise NameError('命名错误')
```

结果：

```
Traceback (most recent call last):
  File "E:/PythonProject/C200105/1.py",
line 1, in <module>
    raise NameError('命名错误')
NameError: 命名错误
```

如果没有`try`和`except`语句覆盖抛出异常得到`raise`语句，该程序就会崩溃，并显示异常的出错信息。因此，我们通常将`raise`语句放在一个函数中，在`try`和`except`语句块中调用该函数，用于判断传入的参数是否满足要求，如果不满足要求则抛出异常。

【示例】raise语句使用实例。

```
#定义输出矩形的函数
def boxPrint(s, w, h):
    if len(s) != 1:                    #当
输入的字符不为单个字符时抛出异常
```

```
            raise Exception('输入的符号必须是单个字
符！')
        if w <= 2:                          #当输入
的宽度小于等于2时抛出异常
            raise Exception('宽必须大于2！')
        if h <= 2:                          #当输入
的高度小于等于2时抛出异常
            raise Exception('高必须大于2！')
    #输出矩形
        print(s * w)
        for i in range(h - 2):
            print(s + (' ' * (w - 2)) + s)
        print(s * w)
    #给s, w, h赋不同的值
    for s, w, h in (('*', 3, 3), ('#', 4, 4),
    ('$$', 3, 3), ('@', 1, 3),('+', 3, 2)):
    #异常处理
        try:
            boxPrint(s, w, h)
        except Exception as err:
            print('发生了一个异常： ' + str(err))
```

【程序说明】在函数内部设置了三个条件用于抛出异常，即当输入值的字符不为单个字符时抛出异常，当输入宽度小于等于`2`时抛出异常，当输入高度小于等于`2`时抛出异常。在调用函数的过程中使用了`try`和`except`语句用于异常处理。

3. 传递异常

捕获到了异常，但是又想重新引发它（传递异常），可以使用**不带参数**的`raise`语句。

```
try:
    raise NameError('命名错误')
except NameError:
    print('出现了一个异常！')
    raise
```

上述示例中，`try`子句中使用了`raise`语句抛出了`NameError`异常，程序会跳转到`except`子句中执行，输出打印语句，然后使用`raise`再次引发刚刚发生的异常，导致程序出现错误而终止程序。

```
出现了一个异常！
Traceback (most recent call last):
  File "E:/PythonProject/C200105/1.py",
line 2, in <module>
    raise NameError('命名错误')
NameError: 命名错误
```

10.4.2. assert 语句

`assert`语句又称为**断言**，断言表示为一些逻辑表达式，程序员相信，在程序中的某个特定点该表达式为真，如果为假，就会触发`AssertionError`异常。assert语句的基本语法如下：

```
assert 逻辑表达式 [,参数]
```

`assert`后面紧跟逻辑表达式，参数为一个字符串，当表达式的值为假时，作为异常类的描述信息使用。逻辑上等同于：

```
if not 逻辑表达式:
    raise AssertionError(参数)
```

例如：

```
a = 3
assert a > 5, 'a 的值应该大于5'
```

上述示例中，结果如下：

```
Traceback (most recent call last):
  File "E:/PythonProject/C200105/1.py", line 2, in <module>
    assert a > 5, 'a 的值应该大于5'
AssertionError: a 的值应该大于5
```

`assert`语句用来收集用户定义的约束条件，而不是捕获内在的程序设计错误。

```
while True:
    try:
        score = int(input('请输入百分制成绩：'))
        assert 0 <= score <= 100, '分数必须在1~100之间'
        if score >= 90:
            print('优')
        elif score >= 80:
            print('良')
        elif score >= 70:
            print('中')
        elif score >= 60:
            print('及格')
        else:
            print('不及格')
    except Exception as r:
        print('发生异常：', r)
        break
```

【程序说明】整个程序位于`while`循环内部，循环中通过`try-except`进行异常处理在`try`子句中，通过键盘获取了`int`类型的数据`score`，然后断言`score`的值必须是在`0~100`分之间。如果输入的数据不在`0～100`之间，则会抛出`AssertionError`异常，从而执行`except`子句，输出提示信息并跳出循环，结束程序。

10.5. 自定义异常

Python 的异常分为两种：一种是**内建异常**，就是系统内置的异常，在某些错误出现时自动触发；另一种是用户自定义异常，就是用户根据自己的需求设置的异常。

`Exception`类是所有异常的基类，因此，用户自定义异常类需从`Exception`类继承。

【示例】用户注册账户时，输入的性别只能是`男`或`女`，要求自定义异常，当输入数据不是`男`或`女`时抛出异常。

```
# 用户自定义异常类
class SexException(Exception):
    def __init__(self, msg, value):
        self.msg = msg
        self.value = value


# 定义函数，用于输入性别并判断是否输入的是“男”或“女”
def f():
    sex = input('请输入性别:')
    if sex != '男' and sex != '女':
        raise SexException('性别只能输入男或者女',
sex)  # 抛出异常
```

```
# 异常处理
try:
    f()
except Exception as ex:
    print('错误信息是:%s,输入的性别是: %s' %
(ex.msg, ex.value))
```