8. 错误和异常

至此，本教程还未深入介绍错误信息，但如果您输入过本教程前文中的例子，应该已经看到过一些错误信息。目前，（至少）有两种不同错误：*句法错误* 和 *异常*。

8.1. 句法错误

句法错误又称解析错误，是学习 Python 时最常见的错误：

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

解析器会复现出现句法错误的代码行，并用小“箭头”指向行里检测到的第一个错误。错误是由箭头 *上方* 的 token 触发的（至少是在这里检测出的）：本例中，在 `print()` 函数中检测到错误，因为，在它前面缺少冒号（`':'`）。错误信息还输出文件名与行号，在使用脚本文件时，就可以知道去哪里查错。

8.2. 异常

即使语句或表达式使用了正确的语法，执行时仍可能触发错误。执行时检测到的错误称为 *异常*，异常不一定导致严重的后果：很快我们就能学会如何处理 Python 的异常。大多数异常不会被程序处理，而是显示下列错误信息：

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

错误信息的最后一行说明程序遇到了什么类型的错误。异常有不同的类型，而类型名称会作为错误信息的一部分中打印出来：上述示例中的异常类型依次是：`ZeroDivisionError`， `NameError` 和 `TypeError`。作为异常类型打印的字符串是发生的内置异常的名称。对于所有内置异常都是如此，但对于用户定义的异常则不一定如此（虽然这种规范很有用）。标准的异常类型是内置的标识符（不是保留关键字）。

此行其余部分根据异常类型，结合出错原因，说明错误细节。

错误信息开头用堆栈回溯形式展示发生异常的语境。一般会列出源代码行的堆栈回溯；但不会显示从标准输入读取的行。

内置异常 列出了内置异常及其含义。

8.3. 异常的处理

可以编写程序处理选定的异常。下例会要求用户一直输入内容，直到输入有效的整数，但允许用户中断程序（使用 `Control-C` 或操作系统支持的其他操作）；注意，用户中断程序会触发 `KeyboardInterrupt` 异常。

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

`try` 语句的工作原理如下：

- 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。

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，例如:

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

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 --- 列出派生类的 *except 子句* 与基类不兼容）。 例如，下面的代码将依次打印 B, C, D:

```
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")
```

请注意如果颠倒 *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*。

所有异常都继承自 BaseException，因此它可被用作通配符。 但这种用法要非常谨慎小心，因为它很容易掩盖真正的编程错误！ 它还可被用于打印错误消息然后重新引发异常（允许调用者再来处理该异常）:

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except BaseException as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

可以选择让最后一个 except 子句省略异常名称，但在此之后异常值必须从 `sys.exc_info()[1]` 获取。

try ... except 语句具有可选的 *else 子句*，该子句如果存在，它必须放在所有 *except 子句* 之后。 它适用于 *try 子句* 没有引发异常但又必须要执行的代码。 例如:

```
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()
```

使用 else 子句比向 try 子句添加额外的代码要好，可以避免意外捕获非 try ... except 语句保护的代码触发的异常。

发生异常时，它可能具有关联值，即异常 *参数* 。是否需要参数，以及参数的类型取决于异常的类型。

except 子句 可以在异常名称后面指定一个变量。 这个变量会绑定到一个异常实例并将参数存储在 `instance.args` 中。 为了方便起见，该异常实例定义了 `__str__()` 以便能直接打印参数而无需引用 `.args`。 也可以在引发异常之前先实例化一个异常并根据需要向其添加任何属性。:

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

如果异常有参数，则它们将作为未处理异常的消息的最后一部分（'详细信息'）打印。

异常处理程序不仅会处理在 *try* *子句* 中发生的异常，还会处理在 *try* *子句* 中调用（包括间接调用）的函数。 例如:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

8.4. 触发异常

`raise` 语句支持强制触发指定的异常。例如：

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

`raise` 唯一的参数就是要触发的异常。这个参数必须是异常实例或异常类（派生自 `Exception` 类）。如果传递的是异常类，将通过调用没有参数的构造函数来隐式实例化：

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

如果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('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere
```

8.5. 异常链

`raise` 语句支持可选的 `from` 子句，该子句用于启用链式异常。 例如：

```
# exc must be exception instance or None.
raise RuntimeError from exc
```

转换异常时，这种方式很有用。例如：

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
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
```

异常链会在 except 或 finally 子句内部引发异常时自动生成。 这可以通过使用 `from None` 这样的写法来禁用:

```
>>> try:
...     open('database.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError
```

异常链机制详见 内置异常。

8.6. 用户自定义异常

程序可以通过创建新的异常类命名自己的异常（Python 类的内容详见 类）。不论是以直接还是间接的方式，异常都应从 Exception 类派生。

异常类可以被定义成能做其他类所能做的任何事，但通常应当保持简单，它往往只提供一些属性，允许相应的异常处理程序提取有关错误的信息。

大多数异常命名都以 “Error” 结尾，类似标准异常的命名。

许多标准模块都需要自定义异常，以报告由其定义的函数中出现的错误。有关类的说明，详见 类。

8.7. 定义清理操作

try 语句还有一个可选子句，用于定义在所有情况下都必须要执行的清理操作。例如：

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

如果存在 finally 子句，则 finally 子句是 try 语句结束前执行的最后一项任务。不论 try 语句是否触发异常，都会执行 finally 子句。以下内容介绍了几种比较复杂的触发异常情景：

- 如果执行 try 子句期间触发了某个异常，则某个 except 子句应处理该异常。如果该异常没有 except 子句处理，在 finally 子句执行后会被重新触发。
- except 或 else 子句执行期间也会触发异常。 同样，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，异常将不会被重新引发。
- 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。

例如：

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

这是一个比较复杂的例子：

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
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'
```

如上所示，任何情况下都会执行 `finally` 子句。`except` 子句不处理两个字符串相除触发的 `TypeError`，因此会在 `finally` 子句执行后被重新触发。

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

8.8. 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。比如，下例要打开一个文件，并输出文件内容：

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

这个代码的问题在于，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，但对于较大的应用程序来说可能会出问题。`with` 语句支持以及时、正确的清理的方式使用文件对象：

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

语句执行完毕后，即使在处理行时遇到问题，都会关闭文件 *f*。和文件一样，支持预定义清理操作的对象会在文档中指出这一点。