8. 错误和异常

至此，本教程还未深入介绍错误信息，但如果您尝试过本教程前文中的例子，应该已经看到过一些错误信息。错误可（至少）被分为两种：*语法错误* 和 *异常*。

8.1. 语法错误

语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
```

解析器会重复出错的行并显示指向检测到错误的位置的小箭头。 请注意这并不一定是需要被修复的位置。 在这个例子中，错误在 `print()` 上被检测到，原因则是在它之前缺少一个冒号 (`':'`)。

将会打印文件名 (在我们的例子中为 `<stdin>`) 和行号以便你在输入是来自文件时能知道要去哪里查看。

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>
    10 * (1/0)
         ~^~
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    4 + spam*3
        ^^^^
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    '2' + 2
    ~~~~^~~
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，则它会被传递到外层的 `try` 语句中；如果没有找到处理器，则它是一个 *未处理异常* 且执行将停止并输出一条错误消息。

`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
```

一个 `except` 子句中的类匹配的异常将是该类本身的实例或其所派生的类的实例（但反过来则不可以 --- 列出派生类的 *except 子句* 不会匹配其基类的实例）。 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*。

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 可能会在异常名称后面指定一个变量。 这个变量将被绑定到异常实例，该实例通常会有一个存储参数的 `args` 属性。 为了方便起见，内置异常类型定义了 `__str__()` 来打印所有参数而不必显式地访问 `.args`。

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # 异常的类型
...     print(inst.args)     # 参数保存在 .args 中
...     print(inst)          # __str__ 允许 args 被直接打印,
...                          # 但可能在异常子类中被覆盖
...     x, y = inst.args     # 解包 args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

未处理异常的 `__str__()` 输出会被打印为该异常消息的最后部分 ('detail')。

`BaseException` 是所有异常的共同基类。它的一个子类， `Exception` ，是所有非致命异常的基类。不是 `Exception` 的子类的异常通常不被处理，因为它们被用来指示程序应该终止。它们包括由 `sys.exit()` 引发的 `SystemExit` ，以及当用户希望中断程序时引发的 `KeyboardInterrupt` 。

`Exception` 可以被用作通配符，捕获（几乎）一切。然而，好的做法是，尽可能具体地说明我们打算处理的异常类型，并允许任何意外的异常传播下去。

处理 `Exception` 最常见的模式是打印或记录异常，然后重新提出（允许调用者也处理异常）：

```
import sys

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

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。

异常处理程序不仅会处理在 *try 子句* 中立刻发生的异常，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>
    raise NameError('HiThere')
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 BaseException 类，例如 Exception 或其子类）。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  # 'raise ValueError()' 的简化
```

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>
    raise NameError('HiThere')
NameError: HiThere
```

8.5. 异常链

如果一个未处理的异常发生在 except 部分内，它将会有被处理的异常附加到它上面，并包括在错误信息中：

```
>>> try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    open("database.sqlite")
    ~~~~^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

为了表明一个异常是另一个异常的直接后果， raise 语句允许一个可选的 from 子句：

```
# exc 必须为异常实例或为 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>
    func()
    ~~~~^^
  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>
    raise RuntimeError('Failed to open database') from exc
RuntimeError: Failed to open database
```

它还允许使用 `from None` 表达禁用自动异常链:

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

许多标准模块定义了自己的异常，以报告他们定义的函数中可能出现的错误。

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!
Traceback (most recent call last):
```

```
  File "<stdin>", line 2, in <module>
    raise KeyboardInterrupt
KeyboardInterrupt
```

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 子句执行 break、 continue 或 return 语句，异常不重新引发。 这可能会引起混淆，因此不鼓励使用。 从 3.14 版开始，编译器会为它发出一个 SyntaxWarning (参见 **PEP 765**)。
- 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。
- 如果一个 finally 子句包含一个 return 语句 ，返回的值将是来自 finally 子句的 return 语句 ，而不是来自 try 子句的 return 语句 。这可能会引起混淆，因此不提倡使用。从版 3.14 开始，编译器会为它发出一个 SyntaxWarning （参见 **PEP 765** ）。

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>
```

```
    divide("2", "1")
    ~~~~~~^^^^^^^^^^
  File "<stdin>", line 3, in divide
    result = x / y
             ~~^~~
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。

8.9. 引发和处理多个不相关的异常

在有些情况下，有必要报告几个已经发生的异常。这通常是在并发框架中当几个任务并行失败时的情况，但也有其他的用例，有时需要是继续执行并收集多个错误而不是引发第一个异常。

内置的 `ExceptionGroup` 打包了一个异常实例的列表，这样它们就可以一起被引发。它本身就是一个异常，所以它可以像其他异常一样被捕获。

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
...
>>> f()
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |     f()
  |     ~^^
  |   File "<stdin>", line 3, in f
  |     raise ExceptionGroup('there were problems', excs)
```

```
  | ExceptionGroup: there were problems (2 sub-exceptions)
  +-+---------------- 1 ----------------
    | OSError: error 1
    +---------------- 2 ----------------
    | SystemError: error 2
    +------------------------------------
>>> try:
...     f()
... except Exception as e:
...     print(f'caught {type(e)}: e')
...
caught <class 'ExceptionGroup'>: e
>>>
```

通过使用 `except*` 代替 `except`，我们可以有选择地只处理组中符合某种类型的异常。在下面的例子中，显示了一个嵌套的异常组，每个 `except*` 子句都从组中提取了某种类型的异常，而让所有其他的异常传播到其他子句，并最终被重新引发。

```
>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
...
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |     f()
  |     ~^^
  |   File "<stdin>", line 2, in f
  |     raise ExceptionGroup(
  |     ...<12 lines>...
  |     )
  | ExceptionGroup: group1 (1 sub-exception)
  +-+---------------- 1 ----------------
    | ExceptionGroup: group2 (1 sub-exception)
    +-+---------------- 1 ----------------
      | RecursionError: 4
      +------------------------------------
>>>
```

注意，嵌套在一个异常组中的异常必须是实例，而不是类型。这是因为在实践中，这些异常通常是那些已经被程序提出并捕获的异常，其模式如下:

```
>>> excs = []
... for test in tests:
...     try:
...         test.run()
...     except Exception as e:
...         excs.append(e)
...
>>> if excs:
...    raise ExceptionGroup("Test Failures", excs)
...
```

8.10. 用注释细化异常情况

当一个异常被创建以引发时，它通常被初始化为描述所发生错误的信息。在有些情况下，在异常被捕获后添加信息是很有用的。为了这个目的，异常有一个 `add_note(note)` 方法接受一个字符串，并将其添加到异常的注释列表。标准的回溯在异常之后按照它们被添加的顺序呈现包括所有的注释。

```
>>> try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise TypeError('bad type')
TypeError: bad type
Add some information
Add some more information
>>>
```

例如，当把异常收集到一个异常组时，我们可能想为各个错误添加上下文信息。在下文中，组中的每个异常都有一个说明，指出这个错误是什么时候发生的。

```
>>> def f():
...     raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Happened in Iteration {i+1}')
...         excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |     raise ExceptionGroup('We have some problems', excs)
  | ExceptionGroup: We have some problems (3 sub-exceptions)
```

```
  +-+---------------- 1 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 1
    +---------------- 2 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 2
    +---------------- 3 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 3
    +------------------------------------
>>>
```