内置异常

在 Python 中，所有异常必须为一个派生自 `BaseException` 的类的实例。 在带有提及一个特定类的 `except` 子句的 `try` 语句中，该子句也会处理任何派生自该类的异常类（但不处理 *它* 所派生出的异常类）。 通过子类化创建的两个不相关异常类永远是不等效的，既使它们具有相同的名称。

本章中列出的内置异常可由解释器或内置函数来生成。 除非另有说明，它们都会具有一个提示导致错误详细原因的"关联值"。 这可以是一个字符串或由多个信息项（例如一个错误码和一个解释该错误码的字符串）。 关联值通常会作为参数被传给异常类的构造器。

用户代码可以引发内置异常。 这可被用于测试异常处理程序或报告错误条件，"就像" 在解释器引发了相同异常的情况时一样；但是请注意，没有任何机制能防止用户代码引发不适当的错误。

内置异常类可以被子类化以定义新的异常；鼓励程序员从 `Exception` 类或它的某个子类而不是从 `BaseException` 来派生新的异常。 关于定义异常的更多信息可以在 Python 教程的 用户自定义异常 部分查看。

异常上下文

异常对象上的三个属性提供了有关引发异常所在上下文的信息：

BaseException.**__context__**
BaseException.**__cause__**
BaseException.**__suppress_context__**

当有其他异常已经被处理的情况下又引发一个新异常的时候，新异常的 `__context__` 属性会被自动设为已经被处理的异常。 异常可以在使用了 `except` 或 `finally` 子句，或者 `with` 语句的时候被处理。

这个隐式异常上下文可以通过使用 `from` 配合 `raise` 来补充一个显式的原因:

```
raise new_exc from original_exc
```

跟在 `from` 之后的表达式必须为一个异常或 `None`。 它将在所引发的异常上被设为 `__cause__`。 设置 `__cause__` 还会隐式地将 `__suppress_context__` 属性设为 `True`，这样使用 `raise new_exc from None` 可以有效地将旧异常替换为新异常来显示其目的 (例如将 `KeyError` 转换为 `AttributeError`)，同时让旧异常在 `__context__` 中保持可用以便在调试时执行内省。

除了异常本身的回溯以外，默认的回溯还会显示这些串连的异常。 `__cause__` 中的显式串连异常如果存在将总是显示。 `__context__` 中的隐式串连异常仅在 `__cause__` 为 `None` 且 `__suppress_context__` 为假值时显示。

不论在哪种情况下，异常本身总会在任何串连异常之后显示，以便回溯的最后一行总是显示所引发的最后一个异常。

从内置异常继承

用户代码可以创建继承自某个异常类型的子类。 建议每次仅子类化一个异常类型以避免多个基类处理 `args` 属性的不同方式，以及内存布局不兼容可能导致的冲突。

大多数内置异常都用 C 实现以保证运行效率，参见: Objects/exceptions.c。 其中一些具有自定义内存布局，这使得创建继承自多个异常类型的子类成为不可能。 一个类型的内存布局属于实现细节并可能随着 Python 版本升级而改变，导致在未来可能产生新的冲突。 因此，建议完全避免子类化多个异常类型。

基类

下列异常主要被用作其他异常的基类。

exception **BaseException**

所有内置异常的基类。 它不应该被用户自定义类直接继承 (这种情况请使用 `Exception`)。 如果在此类的实例上调用 `str()`，则会返回实例的参数表示，或者当没有参数时返回空字符串。

args

传给异常构造器的参数元组。 某些内置异常 (例如 `OSError`) 接受特定数量的参数并赋予此元组中的元素特殊的含义，而其他异常通常只接受一个给出错误信息的单独字符串。

with_traceback(*tb*)

此方法会将 *tb* 设为新的异常回溯信息并返回异常对象。 它在 **PEP 3134** 的异常链特性可用之前更为常用。 下面的例子演示了我们如何将一个 `SomeException` 实例转换为 `OtherException` 实例而保留回溯信息。 异常一旦被引发，当前帧会被推至 `OtherException` 的回溯栈顶端，就像当我们允许原始 `SomeException` 被传播给调用方时它的回溯栈将会发生的情形一样。:

```
try:
    ...
except SomeException:
    tb = sys.exception().__traceback__
    raise OtherException(...).with_traceback(tb)
```

__traceback__

保存关联到该异常的 回溯对象 的可写字段。 另请参阅: raise 语句。

add_note(*note*)

将字符串 `note` 添加到在异常字符串之后的标准回溯中显示的注释中。 如果 `note` 不是一个字符串则会引发 `TypeError`。

> *Added in version 3.11.*

__notes__

由此异常的注释组成的列表，它是通过 `add_note()` 添加的。 该属性是在调用 `add_note()` 时创建的。

> *Added in version 3.11.*

exception **Exception**
所有内置的非系统退出类异常都派生自此类。 所有用户自定义异常也应当派生自此类。

exception **ArithmeticError**
此基类用于派生针对各种算术类错误而引发的内置异常: `OverflowError`, `ZeroDivisionError`, `FloatingPointError`。

exception **BufferError**
当与 缓冲区 相关的操作无法执行时将被引发。

exception **LookupError**
此基类用于派生当映射或序列所使用的键或索引无效时引发的异常: `IndexError`, `KeyError`。 这可以通过 `codecs.lookup()` 来直接引发。

具体异常

以下异常属于经常被引发的异常。

exception **AssertionError**
当 `assert` 语句失败时将被引发。

exception **AttributeError**
当属性引用 (参见 属性引用) 或赋值失败时将被引发。 (当一个对象根本不支持属性引用或属性赋值时则将引发 `TypeError`。)

可选的 *name* 和 *obj* 仅限关键字参数设置相应的属性:

name
尝试访问的属性的名称。

obj
针对指定属性被访问的对象。

> *在 3.10 版本发生变更:* 增加了 `name` 和 `obj` 属性。

exception **EOFError**
当 `input()` 函数未读取任何数据即达到文件结束条件（EOF）时被引发。 (注意: `io.IOBase.read()` 和 `io.IOBase.readline()` 方法在遇到 EOF 时将返回一个空字符串。)

exception **FloatingPointError**
目前未被使用。

exception **GeneratorExit**

当一个 generator 或 coroutine 被关闭时将被引发；参见 `generator.close()` 和 `coroutine.close()`。 它直接继承自 `BaseException` 而不是 `Exception`，因为从技术上来说它并不是一个错误。

exception **ImportError**

当 `import` 语句尝试加载模块遇到麻烦时将被引发。 并且当 `from ... import` 中的 "from list" 存在无法找到的名称时也会被引发。

可选的 *name* 和 *path* 仅限关键字参数设置相应的属性：

name

尝试导入的模块的名称。

path

指向任何触发异常的文件的路径。

> *在 3.3 版本发生变更:* 添加了 `name` 与 `path` 属性。

exception **ModuleNotFoundError**

`ImportError` 的子类，当一个模块无法被定位时将由 `import` 引发。 当在 `sys.modules` 中找到 `None` 时也会被引发。

> *Added in version 3.6.*

exception **IndexError**

当序列抽取超出范围时将被引发。 （切片索引会被静默截短到允许的范围；如果指定索引不是整数则 `TypeError` 会被引发。）

exception **KeyError**

当在现有键集合中找不到指定的映射（字典）键时将被引发。

exception **KeyboardInterrupt**

当用户按下中断键 (通常为 Control-C 或 Delete) 时将被引发。 在执行期间，会定期检测中断信号。 该异常继承自 `BaseException` 以确保不会被处理 `Exception` 的代码意外捕获，这样可以避免退出解释器。

> **备注:** 捕获 `KeyboardInterrupt` 需要特别考虑。 因为它可能会在不可预知的点位被引发，在某些情况下，它可能使运行中的程序陷入不一致的状态。 通常最好是让 `KeyboardInterrupt` 尽快结束程序或者完全避免引发它。 (参见 有关信号处理器和异常的注释。)

exception **MemoryError**

当一个操作耗尽内存但情况仍可（通过删除一些对象）进行挽救时将被引发。 关联的值是一个字符串，指明是哪种（内部）操作耗尽了内存。 请注意由于底层的内存管理架构（C 的 `malloc()` 函数），解释器也许并不总是能够从这种情况下完全恢复；但它毕竟可以引发一个异常，这样就能打印出栈回溯信息，以便找出导致问题的失控程序。

exception **NameError**

当某个局部或全局名称未找到时将被引发。 此异常仅用于非限定名称。 关联的值是一条错误信息，其中包含未找到的名称。

可选的 *name* 仅限关键字参数设置该属性：

name

尝试访问的变量的名称。

> *在 3.10 版本发生变更:* 增加了 `name` 属性。

exception **NotImplementedError**

此异常派生自 `RuntimeError`。 在用户自定义的基类中，抽象方法应当在其要求所派生类重写该方法，或是在其要求所开发的类提示具体实现尚待添加时引发此异常。

> **备注:** 它不应当用来表示一个运算符或方法根本不能被支持 -- 在此情况下应当让特定运算符 / 方法保持未定义，或者在子类中将其设为 `None`。

> **小心:** `NotImplementedError` 和 `NotImplemented` 不能互相替代。 此异常应当仅以上文所描述的方式使用；请参阅 `NotImplemented` 了解正确使用该内置常量的相关细节。

exception **OSError**([*arg*])
exception **OSError**(*errno*, *strerror*[, *filename*[, *winerror*[, *filename2*]]])

此异常在一个系统函数返回系统相关的错误时将被引发，此类错误包括 I/O 操作失败例如 "文件未找到" 或 "磁盘已满" 等（不包括非法参数类型或其他偶然性错误）。

构造器的第二种形式可设置如下所述的相应属性。 如果未指定这些属性则默认为 `None`。 为了能向下兼容，如果传入了三个参数，则 `args` 属性将仅包含由前两个构造器参数组成的 2 元组。

构造器实际返回的往往是 `OSError` 的某个子类，如下文 OS exceptions 中所描述的。 具体的子类取决于最终的 `errno` 值。 此行为仅在直接或通过别名来构造 `OSError` 时发生，并且在子类化时不会被继承。

errno

来自于 C 变量 `errno` 的数字错误码。

winerror

在 Windows 下，此参数将给出原生的 Windows 错误码。 而 `errno` 属性将是该原生错误码在 POSIX 平台下的近似转换形式。

在 Windows 下，如果 *winerror* 构造器参数是一个整数，则 `errno` 属性会根据 Windows 错误码来确定，而 *errno* 参数会被忽略。 在其他平台上，*winerror* 参数会被忽略，并且 `winerror` 属性将不存在。

strerror

由操作系统提供的相应错误消息。 它在 POSIX 平台上由 C 函数 `perror()` 进行格式化，而在 Windows 下则是由 `FormatMessage()` 进行。

filename

filename2

对于与文件系统路径有关 (例如 `open()` 或 `os.unlink()`) 的异常，`filename` 是传给函数的文件名。 对于涉及两个文件系统路径的函数 (例如 `os.rename()`)，`filename2` 将是传给函数的第二个文件名。

> *在 3.3 版本发生变更:* `EnvironmentError`, `IOError`, `WindowsError`, `socket.error`, `select.error` 和 `mmap.error` 已并合并到 `OSError`，而构造器可能返回一个子类。

> *在 3.4 版本发生变更:* `filename` 属性现在是传给函数的原始文件名，而不是基于 filesystem encoding and error handler 进行编码或解码之后的名称。 此外，还添加了 *filename2* 构造器参数和属性。

exception **OverflowError**

当算术运算的结果大到无法表示时将被引发。 这对整数来说不可能发生（宁可引发 `MemoryError` 也不会放弃尝试）。 但是出于历史原因，有时也会在整数超出要求范围的情况下引发 OverflowError。 因为在 C 中缺少对浮点异常处理的标准化，大多数浮点运算都不会做检查。

exception **PythonFinalizationError**

该异常派生自 `RuntimeError`。 它会在解释器关闭或称 Python 终结化 期间当有操作被阻止时被引发。

在 Python 终结化期间操作被阻止并引发 `PythonFinalizationError` 的例子:

- 新建一个 Python 线程。
- 并入 一个正在运行的守护线程。
- `os.fork()`。

另请参阅 `sys.is_finalizing()` 函数。

> *Added in version 3.13:* 在此之前将只引发 `RuntimeError`。

> *在 3.14 版本发生变更:* 现在 `threading.Thread.join()` 可以引发该异常。

exception **RecursionError**

此异常派生自 `RuntimeError`。 它会在解释器检测发现超过最大递归深度 (参见 `sys.getrecursionlimit()`) 时被引发。

> *Added in version 3.5:* 在此之前将只引发 `RuntimeError`。

exception **ReferenceError**

此异常将在使用 `weakref.proxy()` 函数所创建的弱引用来访问该引用的某个已被作为垃圾回收的属性时被引发。 有关弱引用的更多信息请参阅 `weakref` 模块。

exception **RuntimeError**

当检测到一个不归属于任何其他类别的错误时将被引发。 关联的值是一个指明究竟发生了什么问题的字符串。

exception **StopIteration**

由内置函数 `next()` 和 iterator 的 `__next__()` 方法所引发，用来表示该迭代器不能产生下一项。

value

该异常对象只有一个属性 `value`，它在构造该异常时作为参数给出，默认值为 `None`。

当一个 generator 或 coroutine 函数返回时，将引发一个新的 `StopIteration` 实例，函数返回的值将被用作异常构造器的 `value` 形参。

如果某个生成器代码直接或间接地引发了 `StopIteration`，它会被转换为 `RuntimeError` (并将 `StopIteration` 保留为导致新异常的原因)。

> *在 3.3 版本发生变更:* 添加了 `value` 属性及其被生成器函数用作返回值的功能。

> *在 3.5 版本发生变更:* 引入了通过 `from __future__ import generator_stop` 来实现 RuntimeError 转换，参见 **PEP 479**。

> *在 3.7 版本发生变更:* 默认对所有代码启用 **PEP 479**: 在生成器中引发的 `StopIteration` 错误将被转换为 `RuntimeError`。

exception **StopAsyncIteration**

必须由一个 asynchronous iterator 对象的 `__anext__()` 方法来引发以停止迭代操作。

> *Added in version 3.5.*

exception **SyntaxError**(*message*, *details*)

当解析器遇到语法错误时引发。 这可以发生在 `import` 语句，对内置函数 `compile()`, `exec()` 或 `eval()` 的调用，或是读取原始脚本或标准输入（也包括交互模式）的时候。

异常实例的 `str()` 只返回错误消息。 错误详情为一个元组，其成员也可在单独的属性中分别获取。

filename

发生语法错误所在文件的名称。

lineno

发生错误所在文件中的行号。 行号索引从 1 开始：文件中首行的 `lineno` 为 1。

offset

发生错误所在文件中的列号。 列号索引从 1 开始：行中首个字符的 `offset` 为 1。

text

错误所涉及的源代码文本。

end_lineno

发生的错误在文件中的末尾行号。 这个索引是从 1 开始的：文件中首行的 `lineno` 为 1。

end_offset

发生的错误在文件中的末尾列号。 这个索引是从 1 开始：行中首个字符的 `offset` 为 1。

对于 f-字符串字段中的错误，消息会带有 "f-string: " 前缀并且其位置是基于替换表达式构建的文本中的位置。 例如，编译 f'Bad {a b} field' 将产生这样的 args 属性: ('f-string: ...', ('', 1, 2, '(a b)n', 1, 5))。

> *在 3.10 版本发生变更:* 增加了 `end_lineno` 和 `end_offset` 属性。

exception **IndentationError**

与不正确的缩进相关的语法错误的基类。 这是 `SyntaxError` 的一个子类。

exception **TabError**

当缩进包含对制表符和空格符不一致的使用时将被引发。 这是 `IndentationError` 的一个子类。

exception **SystemError**

当解释器发现内部错误，但情况看起来尚未严重到要放弃所有希望时将被引发。 关联的值是一个指明发生了什么问题的字符串（使用低层级的表示形式）。 在 CPython 中，这可能会因不正确地使用 Python 的 C API 而引发，例如返回 `NULL` 值而不设置一个异常。

如果你确信此异常不是你的问题，或你所使用的软件包的问题，你应当将此问题报告给你所用 Python 解释器的作者或维护者。 请确保报告 Python 解释器的版本 (`sys.version`；它也会在交互式 Python 会话开始时被打印出来)，具体的错误消息 (异常所关联的值) 以及可能触发该错误的程序的源代码。

exception **SystemExit**

此异常由 `sys.exit()` 函数引发。 它继承自 `BaseException` 而不是 `Exception` 这样它就不会被捕获 `Exception` 的代码所意外捕获。 这允许此异常正确地向上传播并导致解释器退出。 当它未被处理时，Python 解释器将会退出；不会有栈回溯信息被打印。 构造器接受与传给 `sys.exit()` 的相同的参数。 如果参数值是一个整数，它表示系统退出状态（将被传给 C 的 `exit()` 函数）；如果是 `None`，则退出状态值为零；如果它具有其他类型（如字符串），对象的值将被打印而退出状态值为一。

对 `sys.exit()` 的调用会被转换为一个异常以便能执行清理处理程序 (`try` 语句的 `finally` 子句)，并且使得调试器可以执行一段脚本而不必冒失去控制的风险。 如果绝对确实地需要立即退出（例如在调用 `os.fork()` 之后的子进程中）则可使用 `os._exit()`.

code

传给构造器的退出状态码或错误信息（默认为 `None`。）

exception **TypeError**

当一个操作或函数被应用于类型不适当的对象时将被引发。 关联的值是一个字符串，给出有关类型不匹配的详情。

此异常可以由用户代码引发，以表明尝试对某个对象进行的操作不受支持也不应当受支持。 如果某个对象应当支持给定的操作但尚未提供相应的实现，所要引发的适当异常应为 `NotImplementedError`。

传入参数的类型错误 (例如在要求 `int` 时却传入了 `list`) 应当导致 `TypeError`，但传入参数的值错误 (例如传入要求范围之外的数值) 则应当导致 `ValueError`。

exception **UnboundLocalError**

当在函数或方法中对某个局部变量进行引用，但该变量并未绑定任何值时将被引发。 此异常是 `NameError` 的一个子类。

exception **UnicodeError**

当发生与 Unicode 相关的编码或解码错误时将被引发。 此异常是 `ValueError` 的一个子类。

`UnicodeError` 具有一些描述编码或解码错误的属性。 例如 `err.object[err.start:err.end]` 会给出导致编解码器失败的特定无效输入。

encoding

引发错误的编码名称。

reason

描述特定编解码器错误的字符串。

object

编解码器试图要编码或解码的对象。

start

`object` 中无效数据的开始位置索引。

该值将被解读为绝对偏移量所以它不应为负数但是该项约束不会在运行时强制执行。

end

`object` 中无效数据的末尾位置索引（不含）。

该值将被解读为绝对偏移量所以它不应为负数但是该项约束不会在运行时强制执行。

exception **UnicodeEncodeError**

当在编码过程中发生与 Unicode 相关的错误时将被引发。 此异常是 `UnicodeError` 的一个子类。

exception **UnicodeDecodeError**

当在解码过程中发生与 Unicode 相关的错误时将被引发。 此异常是 `UnicodeError` 的一个子类。

exception **UnicodeTranslateError**
在转写过程中发生与 Unicode 相关的错误时将被引发。 此异常是 UnicodeError 的一个子类。

exception **ValueError**
当操作或函数接收到具有正确类型但值不适合的参数，并且情况不能用更精确的异常例如 IndexError 来描述时将被引发。

exception **ZeroDivisionError**
当除法或取余运算的第二个参数为零时将被引发。 关联的值是一个字符串，指明操作数和运算的类型。

下列异常被保留以与之前的版本相兼容；从 Python 3.3 开始，它们都是 OSError 的别名。

exception **EnvironmentError**

exception **IOError**

exception **WindowsError**
限在 Windows 中可用。

OS 异常

下列异常均为 OSError 的子类，它们将根据系统错误代码被引发。

exception **BlockingIOError**
当一个操作将在设置为非阻塞操作的对象（例如套接字）上发生阻塞时将被引发。 对应于 errno EAGAIN, EALREADY, EWOULDBLOCK 和 EINPROGRESS。

除了 OSError 已有的属性，BlockingIOError 还有一个额外属性：

characters_written
一个整数，表示在被阻塞前已写入到流的字符数。 当使用来自 io 模块的带缓冲 I/O 类时此属性可用。

exception **ChildProcessError**
当一个子进程上的操作失败时将被引发。 对应于 errno ECHILD。

exception **ConnectionError**
与连接相关问题的基类。

其子类有 BrokenPipeError, ConnectionAbortedError, ConnectionRefusedError 和 ConnectionResetError。

exception **BrokenPipeError**
ConnectionError 的子类，当试图写入一个管道而其另一端已关闭，或者试图写入一个套接字而其已关闭写入时将被引发。 对应于 errno EPIPE 和 ESHUTDOWN。

exception **ConnectionAbortedError**

`ConnectionError` 的子类，当一个连接尝试被对端中止时将被引发。 对应于 errno `ECONNABORTED`。

exception **ConnectionRefusedError**

`ConnectionError` 的子类，当一个连接尝试被对端拒绝时将被引发。 对应于 errno `ECONNREFUSED`。

exception **ConnectionResetError**

`ConnectionError` 的子类，当一个连接尝试被对端重置时将被引发。 对应于 errno `ECONNRESET`。

exception **FileExistsError**

当试图创建一个已存在的文件或目录时将被引发。 对应于 errno `EEXIST`。

exception **FileNotFoundError**

当所请求的文件或目录不存在时将被引发。 对应于 errno `ENOENT`。

exception **InterruptedError**

当一个系统调用被传入的信号中断时将被引发。 对应于 errno `EINTR`。

> *在 3.5 版本发生变更:* 当系统调用被某个信号中断时，Python 现在会重试系统调用，除非该信号的处理程序引发了其它异常 (原理参见 **PEP 475**) 而不是引发 `InterruptedError`。

exception **IsADirectoryError**

当请求对一个目录执行文件操作 (如 `os.remove()`) 时将被引发。 对应于 errno `EISDIR`。

exception **NotADirectoryError**

当请求对一个非目录执行目录操作 (如 `os.listdir()`) 时将被引发。 在大多数 POSIX 平台上，它还可能在某个操作试图将一个非目录作为目录打开或遍历时被引发。 对应于 errno `ENOTDIR`。

exception **PermissionError**

当在没有足够访问权限的情况下试图运行某个操作时将被引发 —— 例如文件系统权限。 对应于 errno `EACCES`, `EPERM` 和 `ENOTCAPABLE`。

> *在 3.11.1 版本发生变更:* WASI 的 `ENOTCAPABLE` 现在被映射至 `PermissionError`。

exception **ProcessLookupError**

当给定的进程不存在时将被引发。 对应于 errno `ESRCH`。

exception **TimeoutError**

当一个系统函数在系统层级发生超时的情况下将被引发。 对应于 errno `ETIMEDOUT`。

> *Added in version 3.3:* 添加了以上所有 `OSError` 的子类。

参见: **PEP 3151** - 重写 OS 和 IO 异常的层次结构

警告

下列异常被用作警告类别；请参阅 警告类别 文档了解详情。

exception **Warning**
警告类别的基类。

exception **UserWarning**
用户代码所产生警告的基类。

exception **DeprecationWarning**
如果所发出的警告是针对其他 Python 开发者的，则以此作为与已弃用特性相关警告的基类。

会被默认警告过滤器忽略，在 `__main__` 模块中的情况除外 (**PEP 565**)。 启用 Python 开发模式 时会显示此警告。

这个弃用政策是在 **PEP 387** 中描述的。

exception **PendingDeprecationWarning**
对于已过时并预计在未来弃用，但目前尚未弃用的特性相关警告的基类。

这个类很少被使用，因为针对未来可能的弃用发出警告的做法并不常见，而针对当前已有的弃用则推荐使用 `DeprecationWarning`。

会被默认警告过滤器忽略。 启用 Python 开发模式 时会显示此警告。

这个弃用政策是在 **PEP 387** 中描述的。

exception **SyntaxWarning**
与模糊的语法相关的警告的基类。

该警告通常会在编译 Python 源代码时发出，而在运行已编译的代码时通常不会报告。

exception **RuntimeWarning**
与模糊的运行时行为相关的警告的基类。

exception **FutureWarning**
如果所发出的警告是针对以 Python 所编写应用的最终用户的，则以此作为与已弃用特性相关警告的基类。

exception **ImportWarning**
与在模块导入中可能的错误相关的警告的基类。

会被默认警告过滤器忽略。 启用 Python 开发模式 时会显示此警告。

exception **UnicodeWarning**
与 Unicode 相关的警告的基类。

exception **EncodingWarning**
与编码格式相关的警告的基类。

请参阅 选择性的 EncodingWarning 来了解详情。

> *Added in version 3.10.*

exception **BytesWarning**

与 bytes 和 bytearray 相关的警告的基类。

exception **ResourceWarning**

资源使用相关警告的基类。

会被默认警告过滤器忽略。 启用 Python 开发模式 时会显示此警告。

> *Added in version 3.2.*

异常组

下列异常是在有必要引发多个不相关联的异常时使用的。 它们是异常层级结构的一部分因此它们可以像所有其他异常一样通过 except 来处理。 此外，它们还可被 except* 所识别，此语法将基于所包含异常的类型来匹配其子分组。

exception **ExceptionGroup**(*msg*, *excs*)

exception **BaseExceptionGroup**(*msg*, *excs*)

这两个异常类型都将多个异常包装在序列 `excs` 中。 `msg` 形参必须为一个字符串。 这两个类之间的区别在于 BaseExceptionGroup 扩展了 BaseException 并且它可以包装任何异常，而 ExceptionGroup 则扩展了 Exception 并且它只能包装 Exception 的子类。 这样的设计是为了使得 `except Exception` 只捕获 ExceptionGroup 而不捕获 BaseExceptionGroup。

BaseExceptionGroup 构造器返回一个 ExceptionGroup 而不是 BaseExceptionGroup，如果所包含的全部异常都是 Exception 的实例的话，因此它可以被用来制造自动化的选择。 在另一方面，ExceptionGroup 构造器则会引发 TypeError，如果所包含的任何异常不是 Exception 的子类的话。

message

传给构造器的 `msg` 参数。 这是一个只读属性。

exceptions

传给构造器的 `excs` 序列中的由异常组成的元组。 这是一个只读属性。

subgroup(*condition*)

返回一个只包含来自当前组的匹配 *condition* 的异常的异常组，或者如果结果为空则返回 `None`。

该条件可以是一个异常类型或由异常类型组成的元组，在后一种情况中将对每个异常使用在 `except` 子句中所使用的相同检测方式来检测是否匹配。 该条件也可以是一个可调用对象（而非类型对象），它接受一个异常作为其唯一参数并会针对应当属于特定子分组的异常返回真值。

当前异常的嵌套结构会在结果中保留，就如其 message, __traceback__, __cause__, __context__ 和 __notes__ 字段的值一样。 空的嵌套组会在结果中被略去。

条件检测会针对嵌套异常组中的所有异常执行，包括最高层级的和任何嵌套的异常组。 如果针对此类异常组的条件为真值，它将被完整包括在结果中。

> *Added in version 3.13:* `condition` 可以是任意不为类型对象的可调用对象。

split(*condition*)

类似于 subgroup()，但将返回 `(match, rest)` 对，其中 `match` 为 `subgroup(condition)` 而 `rest` 为剩余的非匹配部分。

derive(*excs*)

返回一个具有相同 message 的异常组，但会将异常包装在 `excs` 中。

此方法是由 subgroup() 和 split() 使用的，它们被用于在各种上下文中拆分异常组。 子类需要重写它以便让 subgroup() 和 split() 返回子类的实例而不是 ExceptionGroup。

subgroup() 和 split() 会从原始异常组拷贝 __traceback__, __cause__, __context__ 和 __notes__ 字段到 derive() 所返回的异常组，这样这些字段就不需要被 derive() 更新。

```
>>> class MyGroup(ExceptionGroup):
...     def derive(self, excs):
...         return MyGroup(self.message, excs)
...
>>> e = MyGroup("eg", [ValueError(1), TypeError(2)])
>>> e.add_note("a note")
>>> e.__context__ = Exception("context")
>>> e.__cause__ = Exception("cause")
>>> try:
...     raise e
... except Exception as e:
...     exc = e
...
>>> match, rest = exc.split(ValueError)
>>> exc, exc.__context__, exc.__cause__, exc.__notes__
(MyGroup('eg', [ValueError(1), TypeError(2)]), Exception('context'), Excep
>>> match, match.__context__, match.__cause__, match.__notes__
(MyGroup('eg', [ValueError(1)]), Exception('context'), Exception('cause'),
>>> rest, rest.__context__, rest.__cause__, rest.__notes__
(MyGroup('eg', [TypeError(2)]), Exception('context'), Exception('cause'),
>>> exc.__traceback__ is match.__traceback__ is rest.__traceback__
True
```

请注意 BaseExceptionGroup 定义了 __new__()，因此需要不同构造器签名的子类必须重写该方法而不是 __init__()。 例如，下面定义了一个接受 exit_code 并根据它来构造分组消息的异常组子类。

```
class Errors(ExceptionGroup):
   def __new__(cls, errors, exit_code):
      self = super().__new__(Errors, f"exit code: {exit_code}", errors)
      self.exit_code = exit_code
      return self

   def derive(self, excs):
      return Errors(excs, self.exit_code)
```

类似于 `ExceptionGroup`，任何 `BaseExceptionGroup` 的子类也是 `Exception` 的子类，只能包装 `Exception` 的实例。

> *Added in version 3.11.*

异常层次结构

内置异常的类层级结构如下：

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
```

```
        ├── ReferenceError
        ├── RuntimeError
        │    ├── NotImplementedError
        │    ├── PythonFinalizationError
        │    └── RecursionError
        ├── StopAsyncIteration
        ├── StopIteration
        ├── SyntaxError
        │    └── IndentationError
        │         └── TabError
        ├── SystemError
        ├── TypeError
        ├── ValueError
        │    └── UnicodeError
        │         ├── UnicodeDecodeError
        │         ├── UnicodeEncodeError
        │         └── UnicodeTranslateError
        └── Warning
             ├── BytesWarning
             ├── DeprecationWarning
             ├── EncodingWarning
             ├── FutureWarning
             ├── ImportWarning
             ├── PendingDeprecationWarning
             ├── ResourceWarning
             ├── RuntimeWarning
             ├── SyntaxWarning
             ├── UnicodeWarning
             └── UserWarning
```