Exceções embutidas

No Python, todas as exceções devem ser instâncias de uma classe derivada de `BaseException`. Em uma instrução `try` com uma cláusula `except` que menciona uma classe específica, essa cláusula também lida com qualquer classe de exceção derivada dessa classe (mas não com as classes de exceção a partir das quais *ela* é derivada). Duas classes de exceção que não são relacionadas por subclasse nunca são equivalentes, mesmo que tenham o mesmo nome.

As exceções embutidas listadas abaixo podem ser geradas pelo interpretador ou pelas funções embutidas. Exceto onde mencionado, eles têm um “valor associado” indicando a causa detalhada do erro. Pode ser uma sequência ou uma tupla de vários itens de informação (por exemplo, um código de erro e uma sequência que explica o código). O valor associado geralmente é passado como argumentos para o construtor da classe de exceção.

O código do usuário pode gerar exceções embutidas. Isso pode ser usado para testar um manipulador de exceções ou para relatar uma condição de erro “exatamente como” a situação na qual o interpretador gera a mesma exceção; mas lembre-se de que nada impede o código do usuário de gerar um erro inadequado.

As classes de exceções embutidas podem ser usadas como subclasses para definir novas exceções; Os programadores são incentivados a derivar novas exceções da classe `Exception` ou de uma de suas subclasses, e não de `BaseException`. Mais informações sobre a definição de exceções estão disponíveis no Tutorial do Python em Exceções definidas pelo usuário.

Contexto da exceção

Ao levantar uma nova exceção enquanto outra exceção já está sendo tratada, o atributo `__context__` da nova exceção é automaticamente definido para a exceção tratada. Uma exceção pode ser tratada quando uma cláusula `except` ou `finally`, ou uma instrução `with`, é usada.

Esse contexto implícito da exceção pode ser complementado com uma causa explícita usando `from` com `raise`:

```
raise new_exc from original_exc
```

A expressão a seguir `from` deve ser uma exceção ou `None`. Ela será definida como `__cause__` na exceção levantada. A definição de `__cause__` também define implicitamente o atributo `__suppress_context__` como `True`, de modo que o uso de `raise new_exc from None` substitui efetivamente a exceção antiga pela nova para fins de exibição (por exemplo, convertendo `KeyError` para `AttributeError`), deixando a exceção antiga disponível em `__context__` para introspecção durante a depuração.

O código de exibição padrão do traceback mostra essas exceções encadeadas, além do traceback da própria exceção. Uma exceção explicitamente encadeada em `__cause__` sempre é mostrada quando presente. Uma exceção implicitamente encadeada em `__context__` é mostrada apenas se `__cause__` for `None` e `__suppress_context__` for falso.

Em qualquer um dos casos, a exceção em si sempre é mostrada após todas as exceções encadeadas, de modo que a linha final do traceback sempre mostre a última exceção que foi levantada.

Herdando de exceções embutidas

atributo `args`, bem como devido a possíveis incompatibilidades de layout de memória.

Detalhes da implementação do CPython: A maioria das exceções embutidas são implementadas em C para eficiência, veja: Objects/exceptions.c. Algumas têm layouts de memória personalizados, o que impossibilita a criação de uma subclasse que herda de vários tipos de exceção. O layout de memória de um tipo é um detalhe de implementação e pode mudar entre as versões do Python, levando a novos conflitos no futuro. Portanto, é recomendável evitar criar subclasses de vários tipos de exceção.

Classes base

As seguintes exceções são usadas principalmente como classes base para outras exceções.

exception **BaseException**

A classe base para todas as exceções embutidas. Não é para ser herdada diretamente por classes definidas pelo usuário (para isso, use `Exception`). Se `str()` for chamado em uma instância desta classe, a representação do(s) argumento(s) para a instância será retornada ou a string vazia quando não houver argumentos.

args

A tupla de argumentos fornecidos ao construtor de exceções. Algumas exceções embutidas (como `OSError`) esperam um certo número de argumentos e atribuem um significado especial aos elementos dessa tupla, enquanto outras são normalmente chamadas apenas com uma única string que fornece uma mensagem de erro.

with_traceback(*tb*)

Este método define *tb* como o novo traceback (situação da pilha de execução) para a exceção e retorna o objeto exceção. Era mais comumente usado antes que os recursos de encadeamento de exceções de **PEP 3134** se tornassem disponíveis. O exemplo a seguir mostra como podemos converter uma instância de `SomeException` em uma instância de `OtherException` enquanto preservamos o traceback. Uma vez gerado, o quadro atual é empurrado para o traceback de `OtherException`, como teria acontecido com o traceback de `SomeException` original se tivéssemos permitido que ele se propagasse para o chamador.

```
try:
    ...
except SomeException:
    tb = sys.exc_info()[2]
    raise OtherException(...).with_traceback(tb)
```

add_note(*note*)

Adiciona a string `note` às notas da exceção que aparecem no traceback padrão após a string de exceção. Uma exceção `TypeError` é levantada se `note` não for uma string.

Novo na versão 3.11.

__notes__

Uma lista das notas desta exceção, que foram adicionadas com `add_note()`. Este atributo é criado quando `add_note()` é chamado.

Novo na versão 3.11.

exception **Exception**

exception **ArithmeticError**
A classe base para as exceções embutidas levantadas para vários erros aritméticos: `OverflowError`, `ZeroDivisionError`, `FloatingPointError`.

exception **BufferError**
Levantado quando uma operação relacionada a buffer não puder ser realizada.

exception **LookupError**
A classe base para as exceções levantadas quando uma chave ou índice usado em um mapeamento ou sequência é inválido: `IndexError`, `KeyError`. Isso pode ser levantado diretamente por `codecs.lookup()`.

Exceções concretas

As seguintes exceções são as que geralmente são levantados.

exception **AssertionError**
Levantado quando uma instrução `assert` falha.

exception **AttributeError**
Levantado quando uma referência de atributo (consulte Attribute references) ou atribuição falha. (Quando um objeto não oferece suporte a referências ou atribuições de atributos, `TypeError` é levantado.)

Os atributos `name` e `obj` podem ser configurados usando argumentos somente-nomeados para o construtor. Quando configurados, eles representam o nome do atributo que se tentou acessar e do objeto que foi acessado por esse atributo, respectivamente.

Alterado na versão 3.10: Adicionado os atributos `name` e `obj`.

exception **EOFError**
Levantado quando a função `input()` atinge uma condição de fim de arquivo (EOF) sem ler nenhum dado. (Note: os métodos `io.IOBase.read()` e `io.IOBase.readline()` retornam uma string vazia quando pressionam o EOF.)

exception **FloatingPointError**
Não usado atualmente.

exception **GeneratorExit**
Levantado quando um gerador ou uma corrotina está fechado(a); veja `generator.close()` e `coroutine.close()`. Herda diretamente de `BaseException` em vez de `Exception`, já que tecnicamente não é um erro.

exception **ImportError**
Levantada quando a instrução `import` tem problemas ao tentar carregar um módulo. Também é gerado quando o “from list” em `from ... import` tem um nome que não pode ser encontrado.

Os atributos `name` e `path` podem ser configurados usando argumentos somente-nomeados para o construtor. Quando configurados, eles representam o nome do módulo que foi tentado ser importado e o caminho para qualquer arquivo que acionou a exceção, respectivamente.

Alterado na versão 3.3: Adicionados os atributos `name` e `path`.

exception **ModuleNotFoundError**

Novo na versão 3.6.

exception **IndexError**
Levantada quando um índice de alguma sequência está fora do intervalo. (Índices de fatia são truncados silenciosamente para cair num intervalo permitido; se um índice não for um inteiro, `TypeError` é levantada.)

exception **KeyError**
Levantada quando uma chave de mapeamento (dicionário) não é encontrada no conjunto de chaves existentes.

exception **KeyboardInterrupt**
Levantada quando um usuário aperta a tecla de interrupção (normalmente `Control-C` ou `Delete`). Durante a execução, uma checagem de interrupção é feita regularmente. A exceção herda de `BaseException` para que não seja capturada acidentalmente por códigos que tratam `Exception` e assim evita que o interpretador saia.

> **Nota:** Capturar uma `KeyboardInterrupt` requer consideração especial. Como pode ser levantada em pontos imprevisíveis, pode, em algumas circunstâncias, deixar o programa em execução em um estado inconsistente. Geralmente é melhor permitir que `KeyboardInterrupt` termine o programa o mais rápido possível ou evitar levantá-la de todo. (Veja Note on Signal Handlers and Exceptions.)

exception **MemoryError**
Levantada quando uma operação fica sem memória mas a situação ainda pode ser recuperada (excluindo alguns objetos). O valor associado é uma string que indica o tipo de operação (interna) que ficou sem memória. Observe que, por causa da arquitetura de gerenciamento de memória subjacente (função `malloc()` do C), o interpretador pode não ser capaz de se recuperar completamente da situação; no entanto, levanta uma exceção para que um traceback possa ser impresso, no caso de um outro programa ser a causa.

exception **NameError**
Levantada quando um nome local ou global não é encontrado. Isso se aplica apenas a nomes não qualificados. O valor associado é uma mensagem de erro que inclui o nome que não pode ser encontrado.

O atributo `name` pode ser definido usando um argumento somente-nomeado para o construtor. Quando definido, representa o nome da variável que foi tentada ser acessada.

Alterado na versão 3.10: Adicionado o atributo `name`.

exception **NotImplementedError**
Essa exceção é derivada da `RuntimeError`. Em classes base, definidas pelo usuário, os métodos abstratos devem gerar essa exceção quando requerem que classes derivadas substituam o método, ou enquanto a classe está sendo desenvolvida, para indicar que a implementação real ainda precisa ser adicionada.

> **Nota:** Não deve ser usada para indicar que um operador ou método não será mais suportado – nesse caso deixe o operador / método indefinido ou, se é uma subclasse, defina-o como `None`.

> **Nota:** `NotImplementedError` e `NotImplemented` não são intercambiáveis, mesmo que tenham nomes e propósitos similares. Veja `NotImplemented` para detalhes e casos de uso.

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

Esta exceção é levantada quando uma função do sistema retorna um erro relacionado ao sistema, incluindo falhas do tipo E/S como “file not found” ou “disk full” (não para tipos de argumentos não permitidos ou outro erro acessório).

A segunda forma do construtor definir os atributos correspondentes, descritos abaixo. Os atributos usarão o valor padrão `None` se não forem especificados. Por compatibilidade com versões anteriores, se três argumentos são passados, o atributo `args` contêm somente uma tupla de 2 elementos, os dois primeiros argumentos do construtor.

O construtor geralmente retorna uma subclasse de `OSError`, como descrito abaixo em OS exceptions . A subclasse particular depende do valor final de `errno`. Este comportamento ocorre apenas durante a construção direta ou por meio de um apelido de `OSError`, e não é herdado na criação de subclasses.

errno

Um código de erro numérico da variável C `errno`.

winerror

No Windows, isso fornece o código de erro nativo do Windows. O atributo `errno` é então uma tradução aproximada, em termos POSIX, desse código de erro nativo.

No Windows, se o argumento de construtor *winerror* for um inteiro, o atributo `errno` é determinado a partir do código de erro do Windows e o argumento *errno* é ignorado. Em outras plataformas, o argumento *winerror* é ignorado e o atributo `winerror` não existe.

strerror

A mensagem de erro correspondente, conforme fornecida pelo sistema operacional. É formatada pelas funções C `perror()` no POSIX e `FormatMessage()` no Windows.

filename
filename2

Para exceções que envolvem um caminho do sistema de arquivos (como `open()` ou `os.unlink()`), `filename` é o nome do arquivo passado para a função. Para funções que envolvem dois caminhos de sistema de arquivos (como `os.rename()`), `filename2` corresponde ao segundo nome de arquivo passado para a função.

Alterado na versão 3.3: `EnvironmentError`, `IOError`, `WindowsError`, `socket.error`, `select.error` e `mmap.error` foram fundidos em `OSError`, e o construtor pode retornar uma subclasse.

Alterado na versão 3.4: O atributo `filename` agora é o nome do arquivo original passado para a função, ao invés do nome codificado ou decodificado da tratador de erros e codificação do sistema de arquivos. Além disso, o argumento e o atributo de construtor *filename2* foi adicionado.

exception **OverflowError**

Levantada quando o resultado de uma operação aritmética é muito grande para ser representada. Isso não pode ocorrer para inteiros (que prefere levantar `MemoryError` a desistir). No entanto, por motivos históricos, OverflowError às vezes é levantada para inteiros que estão fora de um intervalo obrigatório. Devido à falta de padronização do tratamento de exceção de ponto flutuante em C, a maioria das operações de ponto flutuante não são verificadas.

exception **RecursionError**

Novo na versão 3.5: Anteriormente, uma `RuntimeError` simples era levantada.

exception **ReferenceError**

Esta exceção é levantada quando um intermediário de referência fraca, criado pela função `weakref.proxy()`, é usado para acessar um atributo do referente após ter sido coletado como lixo. Para mais informações sobre referências fracas, veja o módulo `weakref`.

exception **RuntimeError**

Levantada quando um erro é detectado e não se encaixa em nenhuma das outras categorias. O valor associado é uma string indicando o que precisamente deu errado.

exception **StopIteration**

Levantada pela função embutida `next()` e o método `__next__()` de um iterador para sinalizar que não há mais itens produzidos pelo iterador.

O objeto exceção tem um único atributo `value`, que é fornecido como um argumento ao construir a exceção, e o padrão é `None`.

Quando uma função geradora ou corrotina retorna, uma nova instância `StopIteration` é levantada, e o valor retornado pela função é usado como o parâmetro `value` para o construtor da exceção.

Se um código gerador direta ou indiretamente levantar `StopIteration`, ele é convertido em uma `RuntimeError` (mantendo o `StopIteration` como a nova causa da exceção).

Alterado na versão 3.3: Adicionado o atributo `value` e a capacidade das funções geradoras de usá-lo para retornar um valor.

Alterado na versão 3.5: Introduzida a transformação RuntimeError via `from __future__ import generator_stop`, consulte **PEP 479**.

Alterado na versão 3.7: Habilita **PEP 479** para todo o código por padrão: um erro `StopIteration` levantado em um gerador é transformado em uma `RuntimeError`.

exception **StopAsyncIteration**

Deve ser levantada pelo método `__anext__()` de um objeto iterador assíncrono para parar a iteração.

Novo na versão 3.5.

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

Levantada quando o analisador encontra um erro de sintaxe. Isso pode ocorrer em uma instrução `import`, em uma chamada às funções embutidas `compile()`, `exec()` ou `eval()`, ou ao ler o script inicial ou entrada padrão (também interativamente).

A função `str()` da instância de exceção retorna apenas a mensagem de erro. Detalhes é uma tupla cujos membros também estão disponíveis como atributos separados.

filename

O nome do arquivo em que ocorreu o erro de sintaxe.

lineno

Em qual número de linha no arquivo o erro ocorreu. Este é indexado em 1: a primeira linha no arquivo tem um `lineno` de 1.

um `offset` de 1.

text

O texto do código-fonte envolvido no erro.

end_lineno

Em qual número de linha no arquivo o erro ocorrido termina. Este é indexado em 1: a primeira linha no arquivo tem um `lineno` de 1.

end_offset

A coluna da linha final em que erro ocorrido finaliza Este é indexado em 1: o primeiro caractere na linha tem um `offset` de 1.

Para erros em campos de f-string, a mensagem é prefixada por “f-string: ” e os “offsets” são deslocamentos em um texto construído a partir da expressão de substituição. Por exemplo, compilar o campo f’Bad {a b}’ resulta neste atributo de argumentos: (‘f-string: …’, (‘’, 1, 2, ‘(a b)n’, 1, 5)).

Alterado na versão 3.10: Adicionado os atributos `end_lineno` e `end_offset`.

exception **IndentationError**

Classe base para erros de sintaxe relacionados a indentação incorreta. Esta é uma subclasse de `SyntaxError`.

exception **TabError**

Levantada quando o indentação contém um uso inconsistente de tabulações e espaços. Esta é uma subclasse de `IndentationError`.

exception **SystemError**

Levantada quando o interpretador encontra um erro interno, mas a situação não parece tão grave para fazer com que perca todas as esperanças. O valor associado é uma string que indica o que deu errado (em termos de baixo nível).

Você deve relatar isso ao autor ou mantenedor do seu interpretador Python. Certifique-se de relatar a versão do interpretador Python (`sys.version`; também é impresso no início de uma sessão Python interativa), a mensagem de erro exata (o valor associado da exceção) e se possível a fonte do programa que acionou o erro.

exception **SystemExit**

Esta exceção é levantada pela função `sys.exit()`. Ele herda de `BaseException` em vez de `Exception` para que não seja acidentalmente capturado pelo código que captura `Exception`. Isso permite que a exceção se propague corretamente e faça com que o interpretador saia. Quando não é tratado, o interpretador Python sai; nenhum traceback (situação da pilha de execução) é impresso. O construtor aceita o mesmo argumento opcional passado para `sys.exit()`. Se o valor for um inteiro, ele especifica o status de saída do sistema (passado para a função C `exit()`); se for `None`, o status de saída é zero; se tiver outro tipo (como uma string), o valor do objeto é exibido e o status de saída é um.

Uma chamada para `sys.exit()` é traduzida em uma exceção para que os tratadores de limpeza (cláusulas `finally` das instruções `try`) possam ser executados, e para que um depurador possa executar um script sem correr o risco de perder o controle. A função `os._exit()` pode ser usada se for absolutamente necessário sair imediatamente (por exemplo, no processo filho após uma chamada para `os.fork()`).

exception **TypeError**

Levantada quando uma operação ou função é aplicada a um objeto de tipo inadequado. O valor associado é uma string que fornece detalhes sobre a incompatibilidade de tipo.

Essa exceção pode ser levantada pelo código do usuário para indicar que uma tentativa de operação em um objeto não é suportada e não deveria ser. Se um objeto deve ter suporte a uma dada operação, mas ainda não forneceu uma implementação, `NotImplementedError` é a exceção apropriada a ser levantada.

Passar argumentos do tipo errado (por exemplo, passar uma `list` quando um `int` é esperado) deve resultar em uma `TypeError`, mas passar argumentos com o valor errado (por exemplo, um número fora limites esperados) deve resultar em uma `ValueError`.

exception **UnboundLocalError**

Levantada quando uma referência é feita a uma variável local em uma função ou método, mas nenhum valor foi vinculado a essa variável. Esta é uma subclasse de `NameError`.

exception **UnicodeError**

Levantada quando ocorre um erro de codificação ou decodificação relacionado ao Unicode. É uma subclasse de `ValueError`.

`UnicodeError` possui atributos que descrevem o erro de codificação ou decodificação. Por exemplo, `err.object[err.start:err.end]` fornece a entrada inválida específica na qual o codec falhou.

encoding

O nome da codificação que levantou o erro.

reason

Uma string que descreve o erro de codec específico.

object

O objeto que o codec estava tentando codificar ou decodificar.

start

O primeiro índice de dados inválidos em `object`.

end

O índice após os últimos dados inválidos em `object`.

exception **UnicodeEncodeError**

Levantada quando ocorre um erro relacionado ao Unicode durante a codificação. É uma subclasse de `UnicodeError`.

exception **UnicodeDecodeError**

Levantada quando ocorre um erro relacionado ao Unicode durante a decodificação. É uma subclasse de `UnicodeError`.

exception **UnicodeTranslateError**

Levantada quando ocorre um erro relacionado ao Unicode durante a tradução. É uma subclasse de `UnicodeError`.

exception **ValueError**

exception **ZeroDivisionError**
Levantada quando o segundo argumento de uma divisão ou operação de módulo é zero. O valor associado é uma string que indica o tipo dos operandos e a operação.

As seguintes exceções são mantidas para compatibilidade com versões anteriores; a partir do Python 3.3, eles são apelidos de `OSError`.

exception **EnvironmentError**

exception **IOError**

exception **WindowsError**
Disponível apenas no Windows.

Exceções de sistema operacional

As seguintes exceções são subclasses de `OSError`, elas são levantadas dependendo do código de erro do sistema.

exception **BlockingIOError**
Levantada quando uma operação bloquearia em um objeto (por exemplo, soquete) definido para operação sem bloqueio. Corresponde a `EAGAIN`, `EALREADY`, `EWOULDBLOCK` e `EINPROGRESS` de `errno`.

Além daquelas de `OSError`, `BlockingIOError` pode ter mais um atributo:

characters_written
Um inteiro contendo o número de caracteres gravados no fluxo antes de ser bloqueado. Este atributo está disponível ao usar as classes de E/S em buffer do módulo `io`.

exception **ChildProcessError**
Levantada quando uma operação em um processo filho falhou. Corresponde a `ECHILD` de `errno`.

exception **ConnectionError**
Uma classe base para problemas relacionados à conexão.

Suas subclasses são `BrokenPipeError`, `ConnectionAbortedError`, `ConnectionRefusedError` e `ConnectionResetError`.

exception **BrokenPipeError**
Uma subclasse de `ConnectionError`, levantada ao tentar escrever em um encadeamento, ou *pipe*, enquanto a outra extremidade foi fechada, ou ao tentar escrever em um socket que foi desligado para escrita. Corresponde a `EPIPE` e `ESHUTDOWN` de `errno`.

exception **ConnectionAbortedError**
A subclass of `ConnectionError`, raised when a connection attempt is aborted by the peer. Corresponds to `errno` `ECONNABORTED`.

exception **ConnectionRefusedError**
A subclass of `ConnectionError`, raised when a connection attempt is refused by the peer. Corresponds to `errno` `ECONNREFUSED`.

exception **ConnectionResetError**

exception **FileExistsError**
Raised when trying to create a file or directory which already exists. Corresponds to `errno` `EEXIST`.

exception **FileNotFoundError**
Raised when a file or directory is requested but doesn't exist. Corresponds to `errno` `ENOENT`.

exception **InterruptedError**
Levantada quando uma chamada do sistema é interrompida por um sinal de entrada. Corresponde a `EINTR` de `errno`.

Alterado na versão 3.5: Python agora tenta novamente chamadas de sistema quando uma *syscall* é interrompida por um sinal, exceto se o tratador de sinal levanta uma exceção (veja **PEP 475** para a justificativa), em vez de levantar `InterruptedError`.

exception **IsADirectoryError**
Raised when a file operation (such as `os.remove()`) is requested on a directory. Corresponds to `errno` `EISDIR`.

exception **NotADirectoryError**
Raised when a directory operation (such as `os.listdir()`) is requested on something which is not a directory. On most POSIX platforms, it may also be raised if an operation attempts to open or traverse a non-directory file as if it were a directory. Corresponds to `errno` `ENOTDIR`.

exception **PermissionError**
Raised when trying to run an operation without the adequate access rights - for example filesystem permissions. Corresponds to `errno` `EACCES`, `EPERM`, and `ENOTCAPABLE`.

Alterado na versão 3.11.1: WASI's `ENOTCAPABLE` is now mapped to `PermissionError`.

exception **ProcessLookupError**
Raised when a given process doesn't exist. Corresponds to `errno` `ESRCH`.

exception **TimeoutError**
Raised when a system function timed out at the system level. Corresponds to `errno` `ETIMEDOUT`.

Novo na versão 3.3: Todas as subclasses de `OSError` acima foram adicionadas.

Ver também: **PEP 3151** - Reworking the OS and IO exception hierarchy

Avisos

As seguintes exceções são usadas como categorias de aviso; veja a documentação de Categorias de avisos para mais detalhes.

exception **Warning**
Classe base para categorias de aviso.

exception **UserWarning**
Classe base para avisos gerados pelo código do usuário.

exception **DeprecationWarning**

Ignorado pelos filtros de aviso padrão, exceto no módulo `__main__` (**PEP 565**). Habilitar o Modo de Desenvolvimento do Python mostra este aviso.

The deprecation policy is described in **PEP 387**.

exception **PendingDeprecationWarning**
Classe base para avisos sobre recursos que foram descontinuados e devem ser descontinuados no futuro, mas não foram descontinuados ainda.

Esta classe raramente é usada para emitir um aviso sobre uma possível descontinuação futura, é incomum, e `DeprecationWarning` é preferível para descontinuações já ativas.

Ignorado pelos filtros de aviso padrão. Habilitar o Modo de Desenvolvimento do Python mostra este aviso.

The deprecation policy is described in **PEP 387**.

exception **SyntaxWarning**
Classe base para avisos sobre sintaxe duvidosa.

exception **RuntimeWarning**
Classe base para avisos sobre comportamento duvidoso de tempo de execução.

exception **FutureWarning**
Classe base para avisos sobre recursos descontinuados quando esses avisos se destinam a usuários finais de aplicações escritas em Python.

exception **ImportWarning**
Classe base para avisos sobre prováveis erros na importação de módulos.

Ignorado pelos filtros de aviso padrão. Habilitar o Modo de Desenvolvimento do Python mostra este aviso.

exception **UnicodeWarning**
Classe base para avisos relacionados a Unicode.

exception **EncodingWarning**
Classe base para avisos relacionados a codificações.

Veja Opt-in EncodingWarning para detalhes.

Novo na versão 3.10.

exception **BytesWarning**
Classe base para avisos relacionados a `bytes` e `bytearray`.

exception **ResourceWarning**
Classe base para avisos relacionados a uso de recursos.

Ignorado pelos filtros de aviso padrão. Habilitar o Modo de Desenvolvimento do Python mostra este aviso.

Novo na versão 3.2.

The following are used when it is necessary to raise multiple unrelated exceptions. They are part of the exception hierarchy so they can be handled with `except` like all other exceptions. In addition, they are recognised by `except*`, which matches their subgroups based on the types of the contained exceptions.

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

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

Both of these exception types wrap the exceptions in the sequence `excs`. The `msg` parameter must be a string. The difference between the two classes is that `BaseExceptionGroup` extends `BaseException` and it can wrap any exception, while `ExceptionGroup` extends `Exception` and it can only wrap subclasses of `Exception`. This design is so that `except Exception` catches an `ExceptionGroup` but not `BaseExceptionGroup`.

The `BaseExceptionGroup` constructor returns an `ExceptionGroup` rather than a `BaseExceptionGroup` if all contained exceptions are `Exception` instances, so it can be used to make the selection automatic. The `ExceptionGroup` constructor, on the other hand, raises a `TypeError` if any contained exception is not an `Exception` subclass.

message

The `msg` argument to the constructor. This is a read-only attribute.

exceptions

A tuple of the exceptions in the `excs` sequence given to the constructor. This is a read-only attribute.

subgroup(*condition*)

Returns an exception group that contains only the exceptions from the current group that match *condition*, or `None` if the result is empty.

The condition can be either a function that accepts an exception and returns true for those that should be in the subgroup, or it can be an exception type or a tuple of exception types, which is used to check for a match using the same check that is used in an `except` clause.

The nesting structure of the current exception is preserved in the result, as are the values of its `message`, `__traceback__`, `__cause__`, `__context__` and `__notes__` fields. Empty nested groups are omitted from the result.

The condition is checked for all exceptions in the nested exception group, including the top-level and any nested exception groups. If the condition is true for such an exception group, it is included in the result in full.

split(*condition*)

Like `subgroup()`, but returns the pair `(match, rest)` where `match` is `subgroup(condition)` and `rest` is the remaining non-matching part.

derive(*excs*)

Returns an exception group with the same `message`, but which wraps the exceptions in `excs`.

This method is used by `subgroup()` and `split()`. A subclass needs to override it in order to make `subgroup()` and `split()` return instances of the subclass rather than `ExceptionGroup`.

`subgroup()` and `split()` copy the `__traceback__`, `__cause__`, `__context__` and `__notes__` fields from the original exception group to the one returned by `derive()`, so these fields do not need

```
>>> class MyGroup(ExceptionGroup):
...     def derive(self, exc):
...         return MyGroup(self.message, exc)
...
>>> 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'), Exception('cau
>>> match, match.__context__, match.__cause__, match.__notes__
(MyGroup('eg', [ValueError(1)]), Exception('context'), Exception('cause'), ['a note
>>> rest, rest.__context__, rest.__cause__, rest.__notes__
(MyGroup('eg', [TypeError(2)]), Exception('context'), Exception('cause'), ['a note'
>>> exc.__traceback__ is match.__traceback__ is rest.__traceback__
True
```

Note that `BaseExceptionGroup` defines `__new__()`, so subclasses that need a different constructor signature need to override that rather than `__init__()`. For example, the following defines an exception group subclass which accepts an exit_code and and constructs the group's message from it.

```
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)
```

Like `ExceptionGroup`, any subclass of `BaseExceptionGroup` which is also a subclass of `Exception` can only wrap instances of `Exception`.

Novo na versão 3.11.

Hierarquia das exceções

A hierarquia de classes para exceções embutidas é:

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
```

```
    ├── 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
    │   └── 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
```