Built-in Exceptions

- Exception context
- Inheriting from built-in exceptions
- Base classes
- Concrete exceptions
  - OS exceptions
- Warnings
- Exception groups
- Exception hierarchy

pair: statement; try pair: statement; except

In Python, all exceptions must be instances of a class that derives from `BaseException`. In a `try` statement with an `except` clause that mentions a particular class, that clause also handles any exception classes derived from that class (but not exception classes from which *it* is derived). Two exception classes that are not related via subclassing are never equivalent, even if they have the same name.

pair: statement; raise

The built-in exceptions listed in this chapter can be generated by the interpreter or built-in functions. Except where mentioned, they have an "associated value" indicating the detailed cause of the error. This may be a string or a tuple of several items of information (e.g., an error code and a string explaining the code). The associated value is usually passed as arguments to the exception class's constructor.

User code can raise built-in exceptions. This can be used to test an exception handler or to report an error condition "just like" the situation in which the interpreter raises the same exception; but beware that there is nothing to prevent user code from raising an inappropriate error.

The built-in exception classes can be subclassed to define new exceptions; programmers are encouraged to derive new exceptions from the `Exception` class or one of its subclasses, and not from `BaseException`. More information on defining exceptions is available in the Python Tutorial under `tut-userexceptions`.

Exception context

pair: exception; chaining __cause__ (exception attribute) __context__ (exception attribute) __suppress_context__ (exception attribute)

Three attributes on exception objects provide information about the context in which the exception was raised:

BaseException.__context__ BaseException.__cause__ BaseException.__suppress_context__

When raising a new exception while another exception is already being handled, the new exception's `!__context__` attribute is automatically set to the handled exception. An exception may be handled when an `except` or `finally` clause, or a `with` statement, is used.

This implicit exception context can be supplemented with an explicit cause by using `!from` with `raise`:

```
raise new_exc from original_exc
```

The expression following `from<raise>` must be an exception or `None`. It will be set as `!__cause__` on the raised exception. Setting `!__cause__` also implicitly sets the `!__suppress_context__` attribute to `True`, so that using `raise new_exc from None` effectively replaces the old exception with the new one for display purposes (e.g. converting `KeyError` to `AttributeError`), while leaving the old exception available in `!__context__` for introspection when debugging.

The default traceback display code shows these chained exceptions in addition to the traceback for the exception itself. An explicitly chained exception in `!__cause__` is always shown when present. An implicitly chained exception in `!__context__` is shown only if `!__cause__` is `None` and `!__suppress_context__` is false.

In either case, the exception itself is always shown after any chained exceptions so that the final line of the traceback always shows the last exception that was raised.

Inheriting from built-in exceptions

User code can create subclasses that inherit from an exception type. It's recommended to only subclass one exception type at a time to avoid any possible conflicts between how the bases handle the `args` attribute, as well as due to possible memory layout incompatibilities.

Most built-in exceptions are implemented in C for efficiency, see: `Objects/exceptions.c`. Some have custom memory layouts which makes it impossible to create a subclass that inherits from multiple

exception types. The memory layout of a type is an implementation detail and might change between Python versions, leading to new conflicts in the future. Therefore, it's recommended to avoid subclassing multiple exception types altogether.

Base classes

The following exceptions are used mostly as base classes for other exceptions.

BaseException

The base class for all built-in exceptions. It is not meant to be directly inherited by user-defined classes (for that, use `Exception`). If `str` is called on an instance of this class, the representation of the argument(s) to the instance are returned, or the empty string when there were no arguments.

args

The tuple of arguments given to the exception constructor. Some built-in exceptions (like `OSError`) expect a certain number of arguments and assign a special meaning to the elements of this tuple, while others are usually called only with a single string giving an error message.

with_traceback(tb)

This method sets *tb* as the new traceback for the exception and returns the exception object. It was more commonly used before the exception chaining features of `3134` became available. The following example shows how we can convert an instance of `SomeException` into an instance of `OtherException` while preserving the traceback. Once raised, the current frame is pushed onto the traceback of the `OtherException`, as would have happened to the traceback of the original `SomeException` had we allowed it to propagate to the caller. :

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

__traceback__

A writable field that holds the `traceback object <traceback-objects>` associated with this exception. See also: `raise`.

add_note(note)

Add the string `note` to the exception's notes which appear in the standard traceback after the exception string. A `TypeError` is raised if `note` is not a string.

3.11

__notes__

A list of the notes of this exception, which were added with `add_note`. This attribute is created when `add_note` is called.

3.11

Exception

All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.

ArithmeticError

The base class for those built-in exceptions that are raised for various arithmetic errors: `OverflowError`, `ZeroDivisionError`, `FloatingPointError`.

BufferError

Raised when a `buffer <bufferobjects>` related operation cannot be performed.

LookupError

The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: `IndexError`, `KeyError`. This can be raised directly by `codecs.lookup`.

Concrete exceptions

The following exceptions are the exceptions that are usually raised.

AssertionError

pair: statement; assert

Raised when an `assert` statement fails.

AttributeError

Raised when an attribute reference (see `attribute-references`) or assignment fails. (When an object does not support attribute references or attribute assignments at all, `TypeError` is raised.)

The optional *name* and *obj* keyword-only arguments set the corresponding attributes:

name

The name of the attribute that was attempted to be accessed.

obj

The object that was accessed for the named attribute.

When possible, `name` and `obj` are set automatically.

3.10 Added the `name` and `obj` attributes.

EOFError

Raised when the `input` function hits an end-of-file condition (EOF) without reading any data. (Note: the `io.TextIOBase.read` and `io.IOBase.readline` methods return an empty string when they hit EOF.)

FloatingPointError

Not currently used.

GeneratorExit

Raised when a `generator` or `coroutine` is closed; see `generator.close` and `coroutine.close`. It directly inherits from `BaseException` instead of `Exception` since it is technically not an error.

ImportError

Raised when the `import` statement has troubles trying to load a module. Also raised when the "from list" in `from ... import` has a name that cannot be found.

The optional *name* and *path* keyword-only arguments set the corresponding attributes:

name

The name of the module that was attempted to be imported.

path

The path to any file which triggered the exception.

3.3 Added the `name` and `path` attributes.

ModuleNotFoundError

A subclass of `ImportError` which is raised by `import` when a module could not be located. It is also raised when `None` is found in `sys.modules`.

3.6

ImportCycleError

A subclass of `ImportError` which is raised when a lazy import fails because it (directly or indirectly) tries to import itself.

3.15

IndexError

Raised when a sequence subscript is out of range. (Slice indices are silently truncated to fall in the allowed range; if an index is not an integer, `TypeError` is raised.)

KeyError

Raised when a mapping (dictionary) key is not found in the set of existing keys.

KeyboardInterrupt

Raised when the user hits the interrupt key (normally `Control-C` or `Delete`). During execution, a check for interrupts is made regularly. The exception inherits from `BaseException` so as to not be accidentally caught by code that catches `Exception` and thus prevent the interpreter from exiting.

> [!NOTE] Catching a `KeyboardInterrupt` requires special consideration. Because it can be raised at unpredictable points, it may, in some circumstances, leave the running program in an inconsistent state. It is generally best to allow `KeyboardInterrupt` to end the program as quickly as possible or avoid raising it entirely. (See `handlers-and-exceptions`.)

MemoryError

Raised when an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating what kind of (internal) operation ran out of memory. Note that because of the underlying memory management architecture (C's `malloc` function), the interpreter may not always be able to completely recover from this situation; it

nevertheless raises an exception so that a stack traceback can be printed, in case a run-away program was the cause.

NameError

Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.

The optional *name* keyword-only argument sets the attribute:

name

The name of the variable that was attempted to be accessed.

3.10 Added the `name` attribute.

NotImplementedError

This exception is derived from `RuntimeError`. In user defined base classes, abstract methods should raise this exception when they require derived classes to override the method, or while the class is being developed to indicate that the real implementation still needs to be added.

> [!NOTE] It should not be used to indicate that an operator or method is not meant to be supported at all -- in that case either leave the operator / method undefined or, if a subclass, set it to `None`.

> [!CAUTION] `!NotImplementedError` and `!NotImplemented` are not interchangeable. This exception should only be used as described above; see `NotImplemented` for details on correct usage of the built-in constant.

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

pair: module; errno

This exception is raised when a system function returns a system-related error, including I/O failures such as "file not found" or "disk full" (not for illegal argument types or other incidental errors).

The second form of the constructor sets the corresponding attributes, described below. The attributes default to `None` if not specified. For backwards compatibility, if three arguments are passed, the `~BaseException.args` attribute contains only a 2-tuple of the first two constructor arguments.

The constructor often actually returns a subclass of `OSError`, as described in OS exceptions below. The particular subclass depends on the final `.errno` value. This behaviour only occurs when constructing `OSError` directly or via an alias, and is not inherited when subclassing.

errno

A numeric error code from the C variable `errno`.

winerror

Under Windows, this gives you the native Windows error code. The `.errno` attribute is then an approximate translation, in POSIX terms, of that native error code.

Under Windows, if the *winerror* constructor argument is an integer, the `.errno` attribute is determined from the Windows error code, and the *errno* argument is ignored. On other platforms, the *winerror* argument is ignored, and the `winerror` attribute does not exist.

strerror

The corresponding error message, as provided by the operating system. Under POSIX, it is obtained using the C functions `!strerror` or `!strerror_r`; under Windows, `!FormatMessage` is used.

filename filename2

For exceptions that involve a file system path (such as `open` or `os.unlink`), `filename` is the file name passed to the function. For functions that involve two file system paths (such as `os.rename`), `filename2` corresponds to the second file name passed to the function.

3.3 `EnvironmentError`, `IOError`, `WindowsError`, `socket.error`, `select.error` and `!mmap.error` have been merged into `OSError`, and the constructor may return a subclass.

3.4 The `filename` attribute is now the original file name passed to the function, instead of the name encoded to or decoded from the `filesystem encoding and error handler`. Also, the *filename2* constructor argument and attribute was added.

OverflowError

Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for integers (which would rather raise `MemoryError` than give up). However, for historical reasons, OverflowError is sometimes raised for integers that are outside a required range. Because of the lack of standardization of floating-point exception handling in C, most floating-point operations are not checked.

PythonFinalizationError

This exception is derived from `RuntimeError`. It is raised when an operation is blocked during interpreter shutdown also known as `Python finalization <interpreter shutdown>`.

Examples of operations which can be blocked with a `PythonFinalizationError` during the Python finalization:

- Creating a new Python thread.
- `Joining <threading.Thread.join>` a running daemon thread.
- `os.fork`,
- acquiring a lock such as `threading.Lock`, when it is known that the operation would otherwise deadlock.

See also the `sys.is_finalizing` function.

3.13 Previously, a plain `RuntimeError` was raised.

3.14

`threading.Thread.join` can now raise this exception.

3.15

This exception may be raised when acquiring `threading.Lock` or `threading.RLock`.

RecursionError

This exception is derived from `RuntimeError`. It is raised when the interpreter detects that the maximum recursion depth (see `sys.getrecursionlimit`) is exceeded.

3.5 Previously, a plain `RuntimeError` was raised.

ReferenceError

This exception is raised when a weak reference proxy, created by the `weakref.proxy` function, is used to access an attribute of the referent after it has been garbage collected. For more information on weak references, see the `weakref` module.

RuntimeError

Raised when an error is detected that doesn't fall in any of the other categories. The associated value is a string indicating what precisely went wrong.

StopIteration

Raised by built-in function `next` and an `iterator`'s `~iterator.__next__` method to signal that there are no further items produced by the iterator.

StopIteration.value

The exception object has a single attribute `!value`, which is given as an argument when constructing the exception, and defaults to `None`.

When a `generator` or `coroutine` function returns, a new `StopIteration` instance is raised, and the value returned by the function is used as the `value` parameter to the constructor of the exception.

If a generator code directly or indirectly raises `StopIteration`, it is converted into a `RuntimeError` (retaining the `StopIteration` as the new exception's cause).

3.3 Added `value` attribute and the ability for generator functions to use it to return a value.

3.5 Introduced the RuntimeError transformation via `from __future__ import generator_stop`, see `479`.

3.7 Enable `479` for all code by default: a `StopIteration` error raised in a generator is transformed into a `RuntimeError`.

StopAsyncIteration

Must be raised by `~object.__anext__` method of an `asynchronous iterator` object to stop the iteration.

3.5

SyntaxError(message, details)

Raised when the parser encounters a syntax error. This may occur in an `import` statement, in a call to the built-in functions `compile`, `exec`, or `eval`, or when reading the initial script or standard input (also interactively).

The `str` of the exception instance returns only the error message. Details is a tuple whose members are also available as separate attributes.

filename

The name of the file the syntax error occurred in.

lineno

Which line number in the file the error occurred in. This is 1-indexed: the first line in the file has a `lineno` of 1.

offset

The column in the line where the error occurred. This is 1-indexed: the first character in the line has an `offset` of 1.

text

The source code text involved in the error.

end_lineno

Which line number in the file the error occurred ends in. This is 1-indexed: the first line in the file has a `lineno` of 1.

end_offset

The column in the end line where the error occurred finishes. This is 1-indexed: the first character in the line has an `offset` of 1.

3.10 Added the `end_lineno` and `end_offset` attributes.

IndentationError

Base class for syntax errors related to incorrect indentation. This is a subclass of `SyntaxError`.

TabError

Raised when indentation contains an inconsistent use of tabs and spaces. This is a subclass of `IndentationError`.

SystemError

Raised when the interpreter finds an internal error, but the situation does not look so serious to cause it to abandon all hope. The associated value is a string indicating what went wrong (in low-level terms). In `CPython`, this could be raised by incorrectly using Python's C API, such as returning a `NULL` value without an exception set.

If you're confident that this exception wasn't your fault, or the fault of a package you're using, you should report this to the author or maintainer of your Python interpreter. Be sure to report the version of the Python interpreter (`sys.version`; it is also printed at the start of an interactive Python session), the exact error message (the exception's associated value) and if possible the source of the program that triggered the error.

SystemExit

This exception is raised by the `sys.exit` function. It inherits from `BaseException` instead of `Exception` so that it is not accidentally caught by code that catches `Exception`. This allows the exception to properly propagate up and cause the interpreter to exit. When it is not handled, the

Python interpreter exits; no stack traceback is printed. The constructor accepts the same optional argument passed to `sys.exit`. If the value is an integer, it specifies the system exit status (passed to C's `!exit` function); if it is `None`, the exit status is zero; if it has another type (such as a string), the object's value is printed and the exit status is one.

A call to `sys.exit` is translated into an exception so that clean-up handlers (`finally` clauses of `try` statements) can be executed, and so that a debugger can execute a script without running the risk of losing control. The `os._exit` function can be used if it is absolutely positively necessary to exit immediately (for example, in the child process after a call to `os.fork`).

code

The exit status or error message that is passed to the constructor. (Defaults to `None`.)

TypeError

Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

This exception may be raised by user code to indicate that an attempted operation on an object is not supported, and is not meant to be. If an object is meant to support a given operation but has not yet provided an implementation, `NotImplementedError` is the proper exception to raise.

Passing arguments of the wrong type (e.g. passing a `list` when an `int` is expected) should result in a `TypeError`, but passing arguments with the wrong value (e.g. a number outside expected boundaries) should result in a `ValueError`.

UnboundLocalError

Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of `NameError`.

UnicodeError

Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of `ValueError`.

`UnicodeError` has attributes that describe the encoding or decoding error. For example, `err.object[err.start:err.end]` gives the particular invalid input that the codec failed on.

encoding

The name of the encoding that raised the error.

reason

A string describing the specific codec error.

object

The object the codec was attempting to encode or decode.

start

The first index of invalid data in `object`.

This value should not be negative as it is interpreted as an absolute offset but this constraint is not enforced at runtime.

end

The index after the last invalid data in `object`.

This value should not be negative as it is interpreted as an absolute offset but this constraint is not enforced at runtime.

UnicodeEncodeError

Raised when a Unicode-related error occurs during encoding. It is a subclass of `UnicodeError`.

UnicodeDecodeError

Raised when a Unicode-related error occurs during decoding. It is a subclass of `UnicodeError`.

UnicodeTranslateError

Raised when a Unicode-related error occurs during translating. It is a subclass of `UnicodeError`.

ValueError

Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError`.

ZeroDivisionError

Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

The following exceptions are kept for compatibility with previous versions; starting from Python 3.3, they are aliases of `OSError`.

EnvironmentError

IOError

WindowsError

Only available on Windows.

OS exceptions

The following exceptions are subclasses of `OSError`, they get raised depending on the system error code.

BlockingIOError

Raised when an operation would block on an object (e.g. socket) set for non-blocking operation. Corresponds to `errno` `~errno.EAGAIN`, `~errno.EALREADY`, `~errno.EWOULDBLOCK` and `~errno.EINPROGRESS`.

In addition to those of `OSError`, `BlockingIOError` can have one more attribute:

characters_written

An integer containing the number of **bytes** written to the stream before it blocked. This attribute is available when using the buffered I/O classes from the `io` module.

ChildProcessError

Raised when an operation on a child process failed. Corresponds to `errno` `~errno.ECHILD`.

ConnectionError

A base class for connection-related issues.

Subclasses are `BrokenPipeError`, `ConnectionAbortedError`, `ConnectionRefusedError` and `ConnectionResetError`.

BrokenPipeError

A subclass of `ConnectionError`, raised when trying to write on a pipe while the other end has been closed, or trying to write on a socket which has been shutdown for writing. Corresponds to `errno` `~errno.EPIPE` and `~errno.ESHUTDOWN`.

ConnectionAbortedError

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

ConnectionRefusedError

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

ConnectionResetError

A subclass of `ConnectionError`, raised when a connection is reset by the peer. Corresponds to `errno` `~errno.ECONNRESET`.

FileExistsError

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

FileNotFoundError

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

InterruptedError

Raised when a system call is interrupted by an incoming signal. Corresponds to `errno` `~errno.EINTR`.

3.5 Python now retries system calls when a syscall is interrupted by a signal, except if the signal handler raises an exception (see `475` for the rationale), instead of raising `InterruptedError`.

IsADirectoryError

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

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` `~errno.ENOTDIR`.

PermissionError

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

3.11.1 WASI's `~errno.ENOTCAPABLE` is now mapped to `PermissionError`.

ProcessLookupError

Raised when a given process doesn't exist. Corresponds to `errno` `~errno.ESRCH`.

TimeoutError

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

3.3 All the above `OSError` subclasses were added.

`3151` - Reworking the OS and IO exception hierarchy

Warnings

The following exceptions are used as warning categories; see the `warning-categories` documentation for more details.

Warning

Base class for warning categories.

UserWarning

Base class for warnings generated by user code.

DeprecationWarning

Base class for warnings about deprecated features when those warnings are intended for other Python developers.

Ignored by the default warning filters, except in the `__main__` module (`565`). Enabling the `Python Development Mode <devmode>` shows this warning.

The deprecation policy is described in `387`.

PendingDeprecationWarning

Base class for warnings about features which are obsolete and expected to be deprecated in the future, but are not deprecated at the moment.

This class is rarely used as emitting a warning about a possible upcoming deprecation is unusual, and `DeprecationWarning` is preferred for already active deprecations.

Ignored by the default warning filters. Enabling the `Python Development Mode <devmode>` shows this warning.

The deprecation policy is described in `387`.

SyntaxWarning

Base class for warnings about dubious syntax.

This warning is typically emitted when compiling Python source code, and usually won't be reported when running already compiled code.

RuntimeWarning

Base class for warnings about dubious runtime behavior.

FutureWarning

Base class for warnings about deprecated features when those warnings are intended for end users of applications that are written in Python.

ImportWarning

Base class for warnings about probable mistakes in module imports.

Ignored by the default warning filters. Enabling the `Python Development Mode <devmode>` shows this warning.

UnicodeWarning

Base class for warnings related to Unicode.

EncodingWarning

Base class for warnings related to encodings.

See `io-encoding-warning` for details.

3.10

BytesWarning

Base class for warnings related to `bytes` and `bytearray`.

ResourceWarning

Base class for warnings related to resource usage.

Ignored by the default warning filters. Enabling the `Python Development Mode <devmode>` shows this warning.

3.2

Exception groups

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*<except_star>`, which matches their subgroups based on the types of the contained exceptions.

ExceptionGroup(msg, excs)

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.

Exception groups are `generic <generics>` over the type of their contained exceptions.

The `excs` parameter may be any sequence, but lists and tuples are specifically processed more efficiently here. For optimal performance, pass a tuple as `excs`.

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 an exception type or tuple of exception types, in which case each exception is checked for a match using the same check that is used in an `except` clause. The condition can also be a callable (other than a type object) that accepts an exception as its single argument and returns true for the exceptions that should be in the subgroup.

The nesting structure of the current exception is preserved in the result, as are the values of its `message`, `~BaseException.__traceback__`, `~BaseException.__cause__`, `~BaseException.__context__` and `~BaseException.__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.

3.13 `condition` can be any callable which is not a type object.

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`, which are used in various contexts to break up an exception group. 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 `~BaseException.__traceback__`, `~BaseException.__cause__`, `~BaseException.__context__` and `~BaseException.__notes__` fields from the original exception group to the one returned by `derive`, so these fields do not need to be updated by `derive`.

>>> 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'), Exception('cause'), ['a note']) >>> 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 `~object.__new__`, so subclasses that need a different constructor signature need to override that rather than `~object.__init__`. For example, the following defines an exception group subclass which accepts an exit_code 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`.

3.11

Exception hierarchy

The class hierarchy for built-in exceptions is:

../../Lib/test/exception_hierarchy.txt

□□□ #2
□ □□□ □□□ 2026-10-31 16:48:14 UTC
□ □□□ □□□ 2026-10-31 16:48:22 UTC