Built-in Exceptions

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.

The built-in exceptions listed below 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 User-defined Exceptions.

When raising (or re-raising) an exception in an `except` or `finally` clause `__context__` is automatically set to the last exception caught; if the new exception is not handled the traceback that is eventually displayed will include the originating exception(s) and the final exception.

When raising a new exception (rather than using a bare `raise` to re-raise the exception currently being handled), the 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` 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.

Base classes

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

exception **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 **PEP 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.exc_info()[2]
    raise OtherException(...).with_traceback(tb)
```

exception **Exception**

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

exception **ArithmeticError**

exception `BufferError`

> Raised w hen a buffer related operation cannot be performed.

exception `LookupError`

> The base class for the exceptions that are raised w hen 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 follow ing exceptions are the exceptions that are usually raised.

exception `AssertionError`

> Raised w hen an `assert` statement fails.

exception `AttributeError`

> Raised w hen 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 `name` and `obj` attributes can be set using keyw ord-only arguments to the constructor. When set they represent the name of the attribute that w as attempted to be accessed and the object that w as accessed for said attribute, respectively.
>
> *Changed in version 3.10:* Added the `name` and `obj` attributes.

exception `EOFError`

> Raised w hen the `input()` function hits an end-of-file condition (EOF) w ithout reading any data. (N.B.: the `io.IOBase.read()` and `io.IOBase.readline()` methods return an empty string w hen they hit EOF.)

exception `FloatingPointError`

> Not currently used.

exception `GeneratorExit`

> Raised w hen 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.

exception `ImportError`

> Raised w hen the `import` statement has troubles trying to load a module. Also raised w hen the "from list" in `from ... import` has a name that cannot be found.
>
> The `name` and `path` attributes can be set using keyw ord-only arguments to the constructor. When set they represent the name of the module that w as attempted to be imported and the path to any file w hich triggered the exception, respectively.
>
> *Changed in version 3.3:* Added the `name` and `path` attributes.

exception `ModuleNotFoundError`

> A subclass of `ImportError` w hich is raised by `import` w hen a module could not be located. It is also raised w hen `None` is found in `sys.modules`.
>
> *New in version 3.6.*

exception `IndexError`

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

exception `KeyError`

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

exception `KeyboardInterrupt`

> Raised w hen 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.

exception `MemoryError`

> Raised w hen an operation runs out of memory but the situation may still be rescued (by deleting some objects). The associated value is a string indicating w hat 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 alw ays 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-aw ay program w as the cause.

includes the name that could not be found.

The `name` attribute can be set using a keyword-only argument to the constructor. When set it represent the name of the variable that was attempted to be accessed.

Changed in version 3.10: Added the `name` attribute.

exception **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`.

> **Note:**   `NotImplementedError` and `NotImplemented` are not interchangeable, even though they have similar names and purposes. See `NotImplemented` for details on when to use it.

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

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 `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. It is formatted by the C functions `perror()` under POSIX, and `FormatMessage()` under Windows.

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.

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

Changed in version 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.

exception **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.

exception **RecursionError**

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

New in version 3.5: Previously, a plain `RuntimeError` was raised.

referent after it has been garbage collected. For more information on weak references, see the weakref module.

exception **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.

exception **StopIteration**

> Raised by built-in function next() and an iterator's __next__() method to signal that there are no further items produced by the iterator.
>
> 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).
>
> *Changed in version 3.3:* Added value attribute and the ability for generator functions to use it to return a value.
>
> *Changed in version 3.5:* Introduced the RuntimeError transformation via from __future__ import generator_stop, see **PEP 479**.
>
> *Changed in version 3.7:* Enable **PEP 479** for all code by default: a StopIteration error raised in a generator is transformed into a RuntimeError.

exception **StopAsyncIteration**

> Must be raised by __anext__() method of an asynchronous iterator object to stop the iteration.
>
> *New in version 3.5.*

exception **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.
>
> For errors in f-string fields, the message is prefixed by "f-string: " and the offsets are offsets in a text constructed from the replacement expression. For example, compiling f'Bad {a b} field' results in this args attribute: ('f-string: …', (", 1, 2, '(a b)n', 1, 5)).
>
> *Changed in version 3.10:* Added the end_lineno and end_offset attributes.

exception **IndentationError**

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

exception **TabError**

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

exception **SystemError**

    3.10.0 ▾   🔍   Go

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.

exception **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`.)

exception **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`.

exception **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`.

exception **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`.
> >
> > **end**
> >
> > > The index after the last invalid data in `object`.

exception **UnicodeEncodeError**

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

exception **UnicodeDecodeError**

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

exception **UnicodeTranslateError**

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

exception **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`.

operands and the operation.

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

exception **EnvironmentError**

exception **IOError**

exception **WindowsError**

> Only available on Windows.

OS exceptions

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

exception **BlockingIOError**

> Raised when an operation would block on an object (e.g. socket) set for non-blocking operation. Corresponds to `errno` `EAGAIN`, `EALREADY`, `EWOULDBLOCK` and `EINPROGRESS`.
>
> In addition to those of `OSError`, `BlockingIOError` can have one more attribute:
>
> **characters_written**
>
> > An integer containing the number of characters written to the stream before it blocked. This attribute is available when using the buffered I/O classes from the `io` module.

exception **ChildProcessError**

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

exception **ConnectionError**

> A base class for connection-related issues.
>
> Subclasses are `BrokenPipeError`, `ConnectionAbortedError`, `ConnectionRefusedError` and `ConnectionResetError`.

exception **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` `EPIPE` and `ESHUTDOWN`.

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**

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

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**

> Raised when a system call is interrupted by an incoming signal. Corresponds to `errno` `EINTR`.
>
> *Changed in version 3.5:* Python now retries system calls when a syscall is interrupted by a signal, except if the signal handler raises an exception (see **PEP 475** for the rationale), instead of raising `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`.

EACCES and EPERM.

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`.

New in version 3.3: All the above `OSError` subclasses were added.

> **See also:** **PEP 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.

exception `Warning`

> Base class for warning categories.

exception `UserWarning`

> Base class for warnings generated by user code.

exception `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 (**PEP 565**). Enabling the Python Development Mode shows this warning.
>
> The deprecation policy is described in **PEP 387**.

exception `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 shows this warning.
>
> The deprecation policy is described in **PEP 387**.

exception `SyntaxWarning`

> Base class for warnings about dubious syntax.

exception `RuntimeWarning`

> Base class for warnings about dubious runtime behavior.

exception `FutureWarning`

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

exception `ImportWarning`

> Base class for warnings about probable mistakes in module imports.
>
> Ignored by the default warning filters. Enabling the Python Development Mode shows this warning.

exception `UnicodeWarning`

> Base class for warnings related to Unicode.

exception `EncodingWarning`

> Base class for warnings related to encodings.
>
> See Opt-in EncodingWarning for details.
>
> *New in version 3.10.*

exception `BytesWarning`

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

Ignored by the default warning filters. Enabling the Python Development Mode shows this warning.

New in version 3.2.

Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- 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
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- EncodingWarning
           +-- ResourceWarning
```