6. Built-in Exceptions

Exceptions should be class objects. The exceptions are defined in the module `exceptions`. This module never needs to be imported explicitly: the exceptions are provided in the built-in namespace as well as the `exceptions` module.

For class exceptions, 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 containing several items of information (e.g., an error code and a string explaining the code). The associated value is the second argument to the `raise` statement. If the exception class is derived from the standard root class `BaseException`, the associated value is present as the exception instance's `args` attribute.

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.

The following exceptions are only used 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()` or `unicode()` 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.

New in version 2.5.

`args`

The tuple of arguments given to the exception constructor. Some built-in exceptions (like `IOError`) 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.

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.

Changed in version 2.5: Changed to inherit from `BaseException`.

exception **StandardError**
The base class for all built-in exceptions except `StopIteration`, `GeneratorExit`, `KeyboardInterrupt` and `SystemExit`. `StandardError` itself is derived from `Exception`.

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

exception **BufferError**
Raised when a buffer related operation cannot be performed.

exception **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()`.

exception **EnvironmentError**
The base class for exceptions that can occur outside the Python system: `IOError`, `OSError`. When exceptions of this type are created with a 2-tuple, the first item is available on the instance's `errno` attribute (it is assumed to be an error number), and the second item is available on the `strerror` attribute (it is usually the associated error message). The tuple itself is also available on the `args` attribute.

New in version 1.5.2.

When an `EnvironmentError` exception is instantiated with a 3-tuple, the first two items are available as above, while the third item is available on the `filename` attribute. However, for backwards compatibility, the `args` attribute contains only a 2-tuple of the first two constructor arguments.

The `filename` attribute is `None` when this exception is created with other than 3 arguments. The `errno` and `strerror` attributes are also `None` when the instance was created with other than 2 or 3 arguments. In this last case, `args` contains the verbatim constructor arguments as a tuple.

The following exceptions are the exceptions that are actually raised.

exception **AssertionError**
Raised when an `assert` statement fails.

exception **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.)

exception **EOFError**
Raised when one of the built-in functions (`input()` or `raw_input()`) hits an end-of-file condition (EOF) without reading any data. (N.B.: the `file.read()` and `file.readline()` methods return an empty string when they hit EOF.)

exception **FloatingPointError**

Raised when a floating point operation fails. This exception is always defined, but can only be raised when Python is configured with the `--with-fpectl` option, or the **`WANT_SIGFPE_HANDLER`** symbol is defined in the `pyconfig.h` file.

exception **`GeneratorExit`**
Raised when a generator's **`close()`** method is called. It directly inherits from **`BaseException`** instead of **`StandardError`** since it is technically not an error.

New in version 2.5.

Changed in version 2.6: Changed to inherit from **`BaseException`**.

exception **`IOError`**
Raised when an I/O operation (such as a **`print`** statement, the built-in **`open()`** function or a method of a file object) fails for an I/O-related reason, e.g., "file not found" or "disk full".

This class is derived from **`EnvironmentError`**. See the discussion above for more information on exception instance attributes.

Changed in version 2.6: Changed **`socket.error`** to use this as a base class.

exception **`ImportError`**
Raised when an **`import`** statement fails to find the module definition or when a `from ... import` fails to find a name that is to be imported.

exception **`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 a plain integer, **`TypeError`** is raised.)

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

exception **`KeyboardInterrupt`**
Raised when the user hits the interrupt key (normally `Control-C` or `Delete`). During execution, a check for interrupts is made regularly. Interrupts typed when a built-in function **`input()`** or **`raw_input()`** is waiting for input also raise this exception. The exception inherits from **`BaseException`** so as to not be accidentally caught by code that catches **`Exception`** and thus prevent the interpreter from exiting.

Changed in version 2.5: Changed to inherit from **`BaseException`**.

exception **`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.

exception **`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.

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.

New in version 1.5.2.

exception **OSError**
This exception is derived from `EnvironmentError`. It is raised when a function returns a system-related error (not for illegal argument types or other incidental errors). The `errno` attribute is a numeric error code from `errno`, and the `strerror` attribute is the corresponding string, as would be printed by the C function `perror()`. See the module `errno`, which contains names for the error codes defined by the underlying operating system.

For exceptions that involve a file system path (such as `chdir()` or `unlink()`), the exception instance will contain a third attribute, `filename`, which is the file name passed to the function.

New in version 1.5.2.

exception **OverflowError**
Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for long integers (which would rather raise `MemoryError` than give up) and for most operations with plain integers, which return a long integer instead. Because of the lack of standardization of floating point exception handling in C, most floating point operations also aren't checked.

exception **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.

New in version 2.2: Previously known as the `weakref.ReferenceError` exception.

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 an iterator's `next()` method to signal that there are no further values. This is derived from `Exception` rather than `StandardError`, since this is not considered an error in its normal application.

New in version 2.2.

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

Instances of this class have attributes `filename`, `lineno`, `offset` and `text` for easier access to the details. `str()` of the exception instance returns only the message.

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**
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).

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. When it is not handled, the Python interpreter exits; no stack traceback is printed. If the associated value is a plain 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.

Instances have an attribute `code` which is set to the proposed exit status or error message (defaulting to `None`). Also, this exception derives directly from `BaseException` and not `StandardError`, since it is not technically an error.

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()`).

The exception inherits from `BaseException` instead of `StandardError` or `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.

Changed in version 2.5: Changed to inherit from `BaseException`.

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.

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`.

New in version 2.0.

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`.

New in version 2.0.

exception **UnicodeEncodeError**

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

New in version 2.3.

exception **UnicodeDecodeError**

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

New in version 2.3.

exception **UnicodeTranslateError**

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

New in version 2.3.

exception **ValueError**

Raised when a built-in 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`.

exception **VMSError**

Only available on VMS. Raised when a VMS-specific error occurs.

exception **WindowsError**

Raised when a Windows-specific error occurs or when the error number does not correspond to an `errno` value. The `winerror` and `strerror` values are created from the return values of the `GetLastError()` and `FormatMessage()` functions from the Windows Platform API. The `errno` value maps the `winerror` value to corresponding `errno.h` values. This is a subclass of `OSError`.

New in version 2.0.

Changed in version 2.5: Previous versions put the `GetLastError()` codes into `errno`.

exception **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 used as warning categories; see the `warnings` module for more information.

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.

exception **PendingDeprecationWarning**
Base class for warnings about features which will be deprecated in the future.

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 constructs that will change semantically in the future.

exception **ImportWarning**
Base class for warnings about probable mistakes in module imports.

New in version 2.5.

exception **UnicodeWarning**
Base class for warnings related to Unicode.

New in version 2.5.

6.1. Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
```