Exception Handling

The functions described in this chapter will let you handle and raise Python exceptions. It is important to understand some of the basics of Python exception handling. It works somewhat like the POSIX `errno` variable: there is a global indicator (per thread) of the last error that occurred. Most C API functions don't clear this on success, but will set it to indicate the cause of the error on failure. Most C API functions also return an error indicator, usually *NULL* if they are supposed to return a pointer, or `-1` if they return an integer (exception: the `PyArg_*()` functions return `1` for success and `0` for failure).

Concretely, the error indicator consists of three object pointers: the exception's type, the exception's value, and the traceback object. Any of those pointers can be NULL if non-set (although some combinations are forbidden, for example you can't have a non-NULL traceback if the exception type is NULL).

When a function must fail because some function it called failed, it generally doesn't set the error indicator; the function it called already set it. It is responsible for either handling the error and clearing the exception or returning after cleaning up any resources it holds (such as object references or memory allocations); it should *not* continue normally if it is not prepared to handle the error. If returning due to an error, it is important to indicate to the caller that an error has been set. If the error is not handled or carefully propagated, additional calls into the Python/C API may not behave as intended and may fail in mysterious ways.

Note: The error indicator is **not** the result of `sys.exc_info()`. The former corresponds to an exception that is not yet caught (and is therefore still propagating), while the latter returns an exception after it is caught (and has therefore stopped propagating).

Printing and clearing

void **`PyErr_Clear`**()

Clear the error indicator. If the error indicator is not set, there is no effect.

void **`PyErr_PrintEx`**(int *set_sys_last_vars*)

Print a standard traceback to `sys.stderr` and clear the error indicator. Call this function only when the error indicator is set. (Otherwise it will cause a fatal error!)

If *set_sys_last_vars* is nonzero, the variables `sys.last_type`, `sys.last_value` and `sys.last_traceback` will be set to the type, value and traceback of the printed exception, respectively.

void **`PyErr_Print`**()

Alias for `PyErr_PrintEx(1)`.

void **`PyErr_WriteUnraisable`**(PyObject **obj*)

This utility function prints a warning message to `sys.stderr` when an exception has been set but it is impossible for the interpreter to actually raise the exception. It is used, for example, when an exception occurs in an `__del__()` method.

The function is called with a single argument *obj* that identifies the context in which the unraisable exception occurred. If possible, the repr of *obj* will be printed in the warning message.

Raising exceptions

These functions help you set the current thread's error indicator. For convenience, some of these functions will always return a NULL pointer for use in a `return` statement.

void **`PyErr_SetString`**(PyObject **type*, const char **message*)

This is the most common way to set the error indicator. The first argument specifies the exception type; it is normally one of the standard exceptions, e.g. `PyExc_RuntimeError`. You need not increment its reference count. The second argument is an error message; it is decoded from `'utf-8`'.

void **`PyErr_SetObject`**(PyObject **type*, PyObject **value*)

This function is similar to `PyErr_SetString()` but lets you specify an arbitrary Python object for the "value" of the exception.

PyObject* **`PyErr_Format`**(PyObject **exception*, const char **format*, ...)

Return value: Always NULL.

This function sets the error indicator and returns *NULL*. *exception* should be a Python exception class. The *format* and subsequent parameters help format the error message; they have the same meaning and values as in `PyUnicode_FromFormat()`. *format* is an ASCII-encoded string.

PyObject* **`PyErr_FormatV`**(PyObject **exception*, const char **format*, va_list *vargs*)

Return value: Always NULL.

Same as PyErr_Format(), but taking a va_list argument rather than a variable number of arguments.

New in version 3.5.

void **PyErr_SetNone**(PyObject **type*)

This is a shorthand for PyErr_SetObject(type, Py_None).

int **PyErr_BadArgument**()

This is a shorthand for PyErr_SetString(PyExc_TypeError, message), where *message* indicates that a built-in operation was invoked with an illegal argument. It is mostly for internal use.

PyObject* **PyErr_NoMemory**()

Return value: Always NULL.

This is a shorthand for PyErr_SetNone(PyExc_MemoryError); it returns *NULL* so an object allocation function can write return PyErr_NoMemory(); when it runs out of memory.

PyObject* **PyErr_SetFromErrno**(PyObject **type*)

Return value: Always NULL.

This is a convenience function to raise an exception when a C library function has returned an error and set the C variable errno. It constructs a tuple object whose first item is the integer errno value and whose second item is the corresponding error message (gotten from strerror()), and then calls PyErr_SetObject(type, object). On Unix, when the errno value is EINTR, indicating an interrupted system call, this calls PyErr_CheckSignals(), and if that set the error indicator, leaves it set to that. The function always returns *NULL*, so a wrapper function around a system call can write return PyErr_SetFromErrno(type); when the system call returns an error.

PyObject* **PyErr_SetFromErrnoWithFilenameObject**(PyObject **type*, PyObject **filenameObject*)

Similar to PyErr_SetFromErrno(), with the additional behavior that if *filenameObject* is not *NULL*, it is passed to the constructor of *type* as a third parameter. In the case of OSError exception, this is used to define the filename attribute of the exception instance.

PyObject* **PyErr_SetFromErrnoWithFilenameObjects**(PyObject **type*, PyObject **filenameObject*, PyObject **filenameObject2*)

Similar to PyErr_SetFromErrnoWithFilenameObject(), but takes a second filename object, for raising errors when a function that takes two filenames fails.

New in version 3.4.

PyObject* **PyErr_SetFromErrnoWithFilename**(PyObject **type*, const char **filename*)

Return value: Always NULL.

Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but the filename is given as a C string. *filename* is decoded from the filesystem encoding (`os.fsdecode()`).

PyObject* **PyErr_SetFromWindowsErr**(int *ierr*)

Return value: Always NULL.

This is a convenience function to raise `WindowsError`. If called with *ierr* of `0`, the error code returned by a call to `GetLastError()` is used instead. It calls the Win32 function `FormatMessage()` to retrieve the Windows description of error code given by *ierr* or `GetLastError()`, then it constructs a tuple object whose first item is the *ierr* value and whose second item is the corresponding error message (gotten from `FormatMessage()`), and then calls `PyErr_SetObject (PyExc_WindowsError, object)`. This function always returns *NULL*. Availability: Windows.

PyObject* **PyErr_SetExcFromWindowsErr**(PyObject **type*, int *ierr*)

Return value: Always NULL.

Similar to `PyErr_SetFromWindowsErr()`, with an additional parameter specifying the exception type to be raised. Availability: Windows.

PyObject* **PyErr_SetFromWindowsErrWithFilename**(int *ierr*, const char **filename*)

Return value: Always NULL.

Similar to `PyErr_SetFromWindowsErrWithFilenameObject()`, but the filename is given as a C string. *filename* is decoded from the filesystem encoding (`os.fsdecode()`). Availability: Windows.

PyObject* **PyErr_SetExcFromWindowsErrWithFilenameObject**(PyObject **type*, int *ierr*, PyObject **filename*)

Similar to `PyErr_SetFromWindowsErrWithFilenameObject()`, with an additional parameter specifying the exception type to be raised. Availability: Windows.

PyObject* **PyErr_SetExcFromWindowsErrWithFilenameObjects**(PyObject **type*, int *ierr*, PyObject **filename*, PyObject **filename2*)

Similar to `PyErr_SetExcFromWindowsErrWithFilenameObject()`, but accepts a second filename object. Availability: Windows.

New in version 3.4.

PyObject* **PyErr_SetExcFromWindowsErrWithFilename**(PyObject **type*, int *ierr*, const char **filename*)

Return value: Always NULL.

Similar to `PyErr_SetFromWindowsErrWithFilename()`, with an additional parameter specifying the exception type to be raised. Availability: Windows.

PyObject* **`PyErr_SetImportError`**(PyObject **msg*, PyObject **name*, PyObject **path*)

This is a convenience function to raise `ImportError`. *msg* will be set as the exception's message string. *name* and *path*, both of which can be `NULL`, will be set as the `ImportError`'s respective `name` and `path` attributes.

New in version 3.3.

void **`PyErr_SyntaxLocationObject`**(PyObject **filename*, int *lineno*, int *col_offset*)

Set file, line, and offset information for the current exception. If the current exception is not a `SyntaxError`, then it sets additional attributes, which make the exception printing subsystem think the exception is a `SyntaxError`.

New in version 3.4.

void **`PyErr_SyntaxLocationEx`**(const char **filename*, int *lineno*, int *col_offset*)

Like `PyErr_SyntaxLocationObject()`, but *filename* is a byte string decoded from the filesystem encoding (`os.fsdecode()`).

New in version 3.2.

void **`PyErr_SyntaxLocation`**(const char **filename*, int *lineno*)

Like `PyErr_SyntaxLocationEx()`, but the col_offset parameter is omitted.

void **`PyErr_BadInternalCall`**()

This is a shorthand for `PyErr_SetString(PyExc_SystemError, message)`, where *message* indicates that an internal operation (e.g. a Python/C API function) was invoked with an illegal argument. It is mostly for internal use.

Issuing warnings

Use these functions to issue warnings from C code. They mirror similar functions exported by the Python `warnings` module. They normally print a warning message to *sys.stderr*; however, it is also possible that the user has specified that warnings are to be turned into errors, and in that case they will raise an exception. It is also possible that the functions raise an exception because of a problem with the warning machinery. The return value is `0` if no exception is raised, or `-1` if an exception is raised. (It is not possible to determine whether a warning message is actually printed, nor what the reason is for the exception; this is intentional.) If an exception is raised, the caller should do its normal exception handling (for example, `Py_DECREF()` owned references and return an error value).

int **`PyErr_WarnEx`**(PyObject **category*, const char **message*, Py_ssize_t *stack_level*)

Issue a warning message. The *category* argument is a warning category (see below) or *NULL*; the *message* argument is a UTF-8 encoded string. *stack_level* is a positive number giving a number of stack frames; the warning will be issued from the currently executing line of code in that stack frame. A *stack_level* of 1 is the function calling `PyErr_WarnEx()`, 2 is the function above that, and so forth.

Warning categories must be subclasses of `PyExc_Warning`; `PyExc_Warning` is a subclass of `PyExc_Exception`; the default warning category is `PyExc_RuntimeWarning`. The standard Python warning categories are available as global variables whose names are enumerated at Standard Warning Categories.

For information about warning control, see the documentation for the `warnings` module and the `-W` option in the command line documentation. There is no C API for warning control.

PyObject* **`PyErr_SetImportErrorSubclass`**(PyObject **msg*, PyObject **name*, PyObject **path*)

Much like `PyErr_SetImportError()` but this function allows for specifying a subclass of `ImportError` to raise.

New in version 3.6.

int **`PyErr_WarnExplicitObject`**(PyObject **category*, PyObject **message*, PyObject **filename*, int *lineno*, PyObject **module*, PyObject **registry*)

Issue a warning message with explicit control over all warning attributes. This is a straightforward wrapper around the Python function `warnings.warn_explicit()`, see there for more information. The *module* and *registry* arguments may be set to *NULL* to get the default effect described there.

New in version 3.4.

int **`PyErr_WarnExplicit`**(PyObject **category*, const char **message*, const char **filename*, int *lineno*, const char **module*, PyObject **registry*)

Similar to `PyErr_WarnExplicitObject()` except that *message* and *module* are UTF-8 encoded strings, and *filename* is decoded from the filesystem encoding (`os.fsdecode()`).

int **PyErr_WarnFormat**(PyObject **category*, Py_ssize_t *stack_level*, const char **format*, ...)

Function similar to `PyErr_WarnEx()`, but use `PyUnicode_FromFormat()` to format the warning message. *format* is an ASCII-encoded string.

New in version 3.2.

int **PyErr_ResourceWarning**(PyObject **source*, Py_ssize_t *stack_level*, const char **format*, ...)

Function similar to `PyErr_WarnFormat()`, but *category* is `ResourceWarning` and pass *source* to `warnings.WarningMessage()`.

New in version 3.6.

Querying the error indicator

PyObject* **PyErr_Occurred**()

Return value: Borrowed reference.

Test whether the error indicator is set. If set, return the exception *type* (the first argument to the last call to one of the `PyErr_Set*()` functions or to `PyErr_Restore()`). If not set, return *NULL*. You do not own a reference to the return value, so you do not need to `Py_DECREF()` it.

> **Note:** Do not compare the return value to a specific exception; use `PyErr_ExceptionMatches()` instead, shown below. (The comparison could easily fail since the exception may be an instance instead of a class, in the case of a class exception, or it may be a subclass of the expected exception.)

int **PyErr_ExceptionMatches**(PyObject **exc*)

Equivalent to `PyErr_GivenExceptionMatches(PyErr_Occurred(), exc)`. This should only be called when an exception is actually set; a memory access violation will occur if no exception has been raised.

int **PyErr_GivenExceptionMatches**(PyObject **given*, PyObject **exc*)

Return true if the *given* exception matches the exception type in *exc*. If *exc* is a class object, this also returns true when *given* is an instance of a subclass. If *exc* is a tuple, all exception types in the tuple (and recursively in subtuples) are searched for a match.

void **PyErr_Fetch**(PyObject ***ptype*, PyObject ***pvalue*, PyObject ***ptraceback*)

Retrieve the error indicator into three variables whose addresses are passed. If the error indicator is not set, set all three variables to *NULL*. If it is set, it will be cleared and you own a reference to each object retrieved. The value and traceback object may be *NULL* even when the type object is not.

Note: This function is normally only used by code that needs to catch exceptions or by code that needs to save and restore the error indicator temporarily, e.g.:

```
{
   PyObject *type, *value, *traceback;
   PyErr_Fetch(&type, &value, &traceback);

   /* ... code that might produce other errors ... */

   PyErr_Restore(type, value, traceback);
}
```

void **PyErr_Restore**(PyObject *type, PyObject *value, PyObject *traceback)

Set the error indicator from the three objects. If the error indicator is already set, it is cleared first. If the objects are *NULL*, the error indicator is cleared. Do not pass a *NULL* type and non-*NULL* value or traceback. The exception type should be a class. Do not pass an invalid exception type or value. (Violating these rules will cause subtle problems later.) This call takes away a reference to each object: you must own a reference to each object before the call and after the call you no longer own these references. (If you don't understand this, don't use this function. I warned you.)

Note: This function is normally only used by code that needs to save and restore the error indicator temporarily. Use `PyErr_Fetch()` to save the current error indicator.

void **PyErr_NormalizeException**(PyObject**exc, PyObject**val, PyObject**tb)

Under certain circumstances, the values returned by `PyErr_Fetch()` below can be "unnormalized", meaning that `*exc` is a class object but `*val` is not an instance of the same class. This function can be used to instantiate the class in that case. If the values are already normalized, nothing happens. The delayed normalization is implemented to improve performance.

Note: This function *does not* implicitly set the `__traceback__` attribute on the exception value. If setting the traceback appropriately is desired, the following additional snippet is needed:

```
if (tb != NULL) {
  PyException_SetTraceback(val, tb);
}
```

void **PyErr_GetExcInfo**(PyObject **ptype*, PyObject **pvalue*, PyObject **ptraceback*)

Retrieve the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. Returns new references for the three objects, any of which may be *NULL*. Does not modify the exception info state.

> **Note:** This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use `PyErr_SetExcInfo()` to restore or clear the exception state.

New in version 3.3.

void **PyErr_SetExcInfo**(PyObject **type*, PyObject **value*, PyObject **traceback*)

Set the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. This function steals the references of the arguments. To clear the exception state, pass *NULL* for all three arguments. For general rules about the three arguments, see `PyErr_Restore()`.

> **Note:** This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use `PyErr_GetExcInfo()` to read the exception state.

New in version 3.3.

Signal Handling

int **PyErr_CheckSignals**()

This function interacts with Python's signal handling. It checks whether a signal has been sent to the processes and if so, invokes the corresponding signal handler. If the `signal` module is supported, this can invoke a signal handler written in Python. In all cases, the default effect for `SIGINT` is to raise the `KeyboardInterrupt` exception. If an exception is raised the error indicator is set and the function returns `-1`; otherwise the function returns `0`. The error indicator may or may not be cleared if it was previously set.

void **`PyErr_SetInterrupt`**()

This function simulates the effect of a `SIGINT` signal arriving — the next time `PyErr_CheckSignals()` is called, `KeyboardInterrupt` will be raised. It may be called without holding the interpreter lock.

int **`PySignal_SetWakeupFd`**(int *fd*)

This utility function specifies a file descriptor to which the signal number is written as a single byte whenever a signal is received. *fd* must be non-blocking. It returns the previous such file descriptor.

The value `-1` disables the feature; this is the initial state. This is equivalent to `signal.set_wakeup_fd()` in Python, but without any error checking. *fd* should be a valid file descriptor. The function should only be called from the main thread.

Changed in version 3.5: On Windows, the function now also supports socket handles.

Exception Classes

PyObject* **`PyErr_NewException`**(const char **name*, PyObject **base*, PyObject **dict*)

Return value: New reference.

This utility function creates and returns a new exception class. The *name* argument must be the name of the new exception, a C string of the form `module.classname`. The *base* and *dict* arguments are normally *NULL*. This creates a class object derived from `Exception` (accessible in C as `PyExc_Exception`).

The `__module__` attribute of the new class is set to the first part (up to the last dot) of the *name* argument, and the class name is set to the last part (after the last dot). The *base* argument can be used to specify alternate base classes; it can either be only one class or a tuple of classes. The *dict* argument can be used to specify a dictionary of class variables and methods.

PyObject* **`PyErr_NewExceptionWithDoc`**(const char **name*, const char **doc*, PyObject **base*, PyObject **dict*)

Return value: New reference.

Same as `PyErr_NewException()`, except that the new exception class can easily be given a docstring: If *doc* is non-*NULL*, it will be used as the docstring for the exception class.

New in version 3.2.

Exception Objects

PyObject* **PyException_GetTraceback**(PyObject *ex*)

Return value: New reference.

Return the traceback associated with the exception as a new reference, as accessible from Python through `__traceback__`. If there is no traceback associated, this returns *NULL*.

int **PyException_SetTraceback**(PyObject *ex*, PyObject *tb*)

Set the traceback associated with the exception to *tb*. Use `Py_None` to clear it.

PyObject* **PyException_GetContext**(PyObject *ex*)

Return the context (another exception instance during whose handling *ex* was raised) associated with the exception as a new reference, as accessible from Python through `__context__`. If there is no context associated, this returns *NULL*.

void **PyException_SetContext**(PyObject *ex*, PyObject *ctx*)

Set the context associated with the exception to *ctx*. Use *NULL* to clear it. There is no type check to make sure that *ctx* is an exception instance. This steals a reference to *ctx*.

PyObject* **PyException_GetCause**(PyObject *ex*)

Return the cause (either an exception instance, or `None`, set by `raise ... from ...`) associated with the exception as a new reference, as accessible from Python through `__cause__`.

void **PyException_SetCause**(PyObject *ex*, PyObject *cause*)

Set the cause associated with the exception to *cause*. Use *NULL* to clear it. There is no type check to make sure that *cause* is either an exception instance or `None`. This steals a reference to *cause*.

`__suppress_context__` is implicitly set to `True` by this function.

Unicode Exception Objects

The following functions are used to create and modify Unicode exceptions from C.

PyObject* **PyUnicodeDecodeError_Create**(const char *encoding*, const char *object*, Py_ssize_t *length*, Py_ssize_t *start*, Py_ssize_t *end*, const char *reason*)

Create a `UnicodeDecodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

PyObject* **PyUnicodeEncodeError_Create**(const char **encoding*, const Py_UNICODE **object*, Py_ssize_t *length*, Py_ssize_t *start*, Py_ssize_t *end*, const char **reason*)

Create a `UnicodeEncodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

PyObject* **PyUnicodeTranslateError_Create**(const Py_UNICODE **object*, Py_ssize_t *length*, Py_ssize_t *start*, Py_ssize_t *end*, const char **reason*)

Create a `UnicodeTranslateError` object with the attributes *object*, *length*, *start*, *end* and *reason*. *reason* is a UTF-8 encoded string.

PyObject* **PyUnicodeDecodeError_GetEncoding**(PyObject **exc*)
PyObject* **PyUnicodeEncodeError_GetEncoding**(PyObject **exc*)

Return the *encoding* attribute of the given exception object.

PyObject* **PyUnicodeDecodeError_GetObject**(PyObject **exc*)
PyObject* **PyUnicodeEncodeError_GetObject**(PyObject **exc*)
PyObject* **PyUnicodeTranslateError_GetObject**(PyObject **exc*)

Return the *object* attribute of the given exception object.

int **PyUnicodeDecodeError_GetStart**(PyObject **exc*, Py_ssize_t **start*)
int **PyUnicodeEncodeError_GetStart**(PyObject **exc*, Py_ssize_t **start*)
int **PyUnicodeTranslateError_GetStart**(PyObject **exc*, Py_ssize_t **start*)

Get the *start* attribute of the given exception object and place it into **start*. *start* must not be *NULL*. Return `0` on success, `-1` on failure.

int **PyUnicodeDecodeError_SetStart**(PyObject **exc*, Py_ssize_t *start*)
int **PyUnicodeEncodeError_SetStart**(PyObject **exc*, Py_ssize_t *start*)
int **PyUnicodeTranslateError_SetStart**(PyObject **exc*, Py_ssize_t *start*)

Set the *start* attribute of the given exception object to *start*. Return `0` on success, `-1` on failure.

int **PyUnicodeDecodeError_GetEnd**(PyObject **exc*, Py_ssize_t **end*)
int **PyUnicodeEncodeError_GetEnd**(PyObject **exc*, Py_ssize_t **end*)
int **PyUnicodeTranslateError_GetEnd**(PyObject **exc*, Py_ssize_t **end*)

Get the *end* attribute of the given exception object and place it into **end*. *end* must not be *NULL*. Return `0` on success, `-1` on failure.

int **PyUnicodeDecodeError_SetEnd**(PyObject **exc*, Py_ssize_t *end*)

int **PyUnicodeEncodeError_SetEnd**(PyObject *exc*, Py_ssize_t *end*)
int **PyUnicodeTranslateError_SetEnd**(PyObject *exc*, Py_ssize_t *end*)

Set the *end* attribute of the given exception object to *end*. Return `0` on success, `-1` on failure.

PyObject* **PyUnicodeDecodeError_GetReason**(PyObject *exc*)
PyObject* **PyUnicodeEncodeError_GetReason**(PyObject *exc*)
PyObject* **PyUnicodeTranslateError_GetReason**(PyObject *exc*)

Return the *reason* attribute of the given exception object.

int **PyUnicodeDecodeError_SetReason**(PyObject *exc*, const char *reason*)

int **PyUnicodeEncodeError_SetReason**(PyObject *exc*, const char *reason*)

int **PyUnicodeTranslateError_SetReason**(PyObject *exc*, const char *reason*)

Set the *reason* attribute of the given exception object to *reason*. Return `0` on success, `-1` on failure.

Recursion Control

These two functions provide a way to perform safe recursive calls at the C level, both in the core and in extension modules. They are needed if the recursive code does not necessarily invoke Python code (which tracks its recursion depth automatically).

int **Py_EnterRecursiveCall**(const char *where*)

Marks a point where a recursive C-level call is about to be performed.

If `USE_STACKCHECK` is defined, this function checks if the OS stack overflowed using `PyOS_CheckStack()`. In this is the case, it sets a `MemoryError` and returns a nonzero value.

The function then checks if the recursion limit is reached. If this is the case, a `RecursionError` is set and a nonzero value is returned. Otherwise, zero is returned.

where should be a string such as `" in instance check"` to be concatenated to the `RecursionError` message caused by the recursion depth limit.

void **Py_LeaveRecursiveCall**()

Ends a `Py_EnterRecursiveCall()`. Must be called once for each *successful* invocation of `Py_EnterRecursiveCall()`.

Properly implementing `tp_repr` for container types requires special recursion handling. In addition to protecting the stack, `tp_repr` also needs to track objects to prevent cycles. The following two functions facilitate this functionality. Effectively, these are the C equivalent to `reprlib.recursive_repr()`.

int **`Py_ReprEnter`**(PyObject **object*)

Called at the beginning of the `tp_repr` implementation to detect cycles.

If the object has already been processed, the function returns a positive integer. In that case the `tp_repr` implementation should return a string object indicating a cycle. As examples, `dict` objects return `{...}` and `list` objects return `[...]`.

The function will return a negative integer if the recursion limit is reached. In that case the `tp_repr` implementation should typically return `NULL`.

Otherwise, the function returns zero and the `tp_repr` implementation can continue normally.

void **`Py_ReprLeave`**(PyObject **object*)

Ends a `Py_ReprEnter()`. Must be called once for each invocation of `Py_ReprEnter()` that returns zero.

Standard Exceptions

All standard Python exceptions are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type `PyObject*`; they are all class objects. For completeness, here are all the variables:

C Name	Python Name	Notes
`PyExc_BaseException`	`BaseException`	(1)
`PyExc_Exception`	`Exception`	(1)
`PyExc_ArithmeticError`	`ArithmeticError`	(1)
`PyExc_AssertionError`	`AssertionError`	
`PyExc_AttributeError`	`AttributeError`	
`PyExc_BlockingIOError`	`BlockingIOError`	
`PyExc_BrokenPipeError`	`BrokenPipeError`	
`PyExc_BufferError`	`BufferError`	
`PyExc_ChildProcessError`	`ChildProcessError`	
`PyExc_ConnectionAbortedError`	`ConnectionAbortedError`	

C Name	Python Name	Notes
PyExc_ConnectionError	ConnectionError	
PyExc_ConnectionRefusedError	ConnectionRefusedError	
PyExc_ConnectionResetError	ConnectionResetError	
PyExc_EOFError	EOFError	
PyExc_FileExistsError	FileExistsError	
PyExc_FileNotFoundError	FileNotFoundError	
PyExc_FloatingPointError	FloatingPointError	
PyExc_GeneratorExit	GeneratorExit	
PyExc_ImportError	ImportError	
PyExc_IndentationError	IndentationError	
PyExc_IndexError	IndexError	
PyExc_InterruptedError	InterruptedError	
PyExc_IsADirectoryError	IsADirectoryError	
PyExc_KeyError	KeyError	
PyExc_KeyboardInterrupt	KeyboardInterrupt	
PyExc_LookupError	LookupError	(1)
PyExc_MemoryError	MemoryError	
PyExc_ModuleNotFoundError	ModuleNotFoundError	
PyExc_NameError	NameError	
PyExc_NotADirectoryError	NotADirectoryError	
PyExc_NotImplementedError	NotImplementedError	
PyExc_OSError	OSError	(1)
PyExc_OverflowError	OverflowError	
PyExc_PermissionError	PermissionError	
PyExc_ProcessLookupError	ProcessLookupError	
PyExc_RecursionError	RecursionError	
PyExc_ReferenceError	ReferenceError	(2)
PyExc_RuntimeError	RuntimeError	
PyExc_StopAsyncIteration	StopAsyncIteration	
PyExc_StopIteration	StopIteration	
PyExc_SyntaxError	SyntaxError	
PyExc_SystemError	SystemError	
PyExc_SystemExit	SystemExit	

C Name	Python Name	Notes
PyExc_TabError	TabError	
PyExc_TimeoutError	TimeoutError	
PyExc_TypeError	TypeError	
PyExc_UnboundLocalError	UnboundLocalError	
PyExc_UnicodeDecodeError	UnicodeDecodeError	
PyExc_UnicodeEncodeError	UnicodeEncodeError	
PyExc_UnicodeError	UnicodeError	
PyExc_UnicodeTranslateError	UnicodeTranslateError	
PyExc_ValueError	ValueError	
PyExc_ZeroDivisionError	ZeroDivisionError	

New in version 3.3: PyExc_BlockingIOError, PyExc_BrokenPipeError, PyExc_ChildProcessError, PyExc_ConnectionError, PyExc_ConnectionAbortedError, PyExc_ConnectionRefusedError, PyExc_ConnectionResetError, PyExc_FileExistsError, PyExc_FileNotFoundError, PyExc_InterruptedError, PyExc_IsADirectoryError, PyExc_NotADirectoryError, PyExc_PermissionError, PyExc_ProcessLookupError and PyExc_TimeoutError were introduced following **PEP 3151**.

New in version 3.5: PyExc_StopAsyncIteration and PyExc_RecursionError.

New in version 3.6: PyExc_ModuleNotFoundError.

These are compatibility aliases to PyExc_OSError:

C Name	Notes
PyExc_EnvironmentError	
PyExc_IOError	
PyExc_WindowsError	(3)

Changed in version 3.3: These aliases used to be separate exception types.

Notes:

1. This is a base class for other standard exceptions.
2. This is the same as weakref.ReferenceError.
3. Only defined on Windows; protect code that uses this by testing that the preprocessor macro MS_WINDOWS is defined.

Standard Warning Categories

All standard Python warning categories are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type `PyObject*`; they are all class objects. For completeness, here are all the variables:

C Name	Python Name	Notes
`PyExc_Warning`	`Warning`	(1)
`PyExc_BytesWarning`	`BytesWarning`	
`PyExc_DeprecationWarning`	`DeprecationWarning`	
`PyExc_FutureWarning`	`FutureWarning`	
`PyExc_ImportWarning`	`ImportWarning`	
`PyExc_PendingDeprecationWarning`	`PendingDeprecationWarning`	
`PyExc_ResourceWarning`	`ResourceWarning`	
`PyExc_RuntimeWarning`	`RuntimeWarning`	
`PyExc_SyntaxWarning`	`SyntaxWarning`	
`PyExc_UnicodeWarning`	`UnicodeWarning`	
`PyExc_UserWarning`	`UserWarning`	

New in version 3.2: `PyExc_ResourceWarning`.

Notes:

1. This is a base class for other standard warning categories.