Operating System Utilities

PyObject* **PyOS_FSPath**(PyObject *path)

Return value: New reference.

Return the file system representation for *path*. If the object is a `str` or `bytes` object, then its reference count is incremented. If the object implements the `os.PathLike` interface, then `__fspath__()` is returned as long as it is a `str` or `bytes` object. Otherwise `TypeError` is raised and `NULL` is returned.

New in version 3.6.

int **Py_FdIsInteractive**(FILE *fp, const char *filename)

Return true (nonzero) if the standard I/O file *fp* with name *filename* is deemed interactive. This is the case for files for which `isatty(fileno(fp))` is true. If the global flag `Py_InteractiveFlag` is true, this function also returns true if the *filename* pointer is *NULL* or if the name is equal to one of the strings `'<stdin>'` or `'???'`.

void **PyOS_AfterFork**()

Function to update some internal state after a process fork; this should be called in the new process if the Python interpreter will continue to be used. If a new executable is loaded into the new process, this function does not need to be called.

int **PyOS_CheckStack**()

Return true when the interpreter runs out of stack space. This is a reliable check, but is only available when `USE_STACKCHECK` is defined (currently on Windows using the Microsoft Visual C++ compiler). `USE_STACKCHECK` will be defined automatically; you should never change the definition in your own code.

PyOS_sighandler_t **PyOS_getsig**(int *i*)

Return the current signal handler for signal *i*. This is a thin wrapper around either `sigaction()` or `signal()`. Do not call those functions directly! `PyOS_sighandler_t` is a typedef alias for `void (*)(int)`.

PyOS_sighandler_t **PyOS_setsig**(int *i*, PyOS_sighandler_t *h*)

Set the signal handler for signal *i* to be *h*; return the old signal handler. This is a thin wrapper around either `sigaction()` or `signal()`. Do not call those functions directly! `PyOS_sighandler_t` is a typedef alias for `void (*)(int)`.

wchar_t* **Py_DecodeLocale**(const char* *arg*, size_t *size)

Decode a byte string from the locale encoding with the surrogateescape error handler: undecodable bytes are decoded as characters in range

U+DC80..U+DCFF. If a byte sequence can be decoded as a surrogate character, escape the bytes using the surrogateescape error handler instead of decoding them.

Encoding, highest priority to lowest priority:

- `UTF-8` on macOS and Android;
- `ASCII` if the `LC_CTYPE` locale is `"C"`, `nl_langinfo(CODESET)` returns the `ASCII` encoding (or an alias), and `mbstowcs()` and `wcstombs()` functions use the `ISO-8859-1` encoding.
- the current locale encoding (`LC_CTYPE` locale).

Return a pointer to a newly allocated wide character string, use `PyMem_RawFree()` to free the memory. If size is not `NULL`, write the number of wide characters excluding the null character into `*size`.

Return `NULL` on decoding error or memory allocation error. If *size* is not `NULL`, `*size` is set to `(size_t)-1` on memory error or set to `(size_t)-2` on decoding error.

Decoding errors should never happen, unless there is a bug in the C library.

Use the `Py_EncodeLocale()` function to encode the character string back to a byte string.

> **See also:** The `PyUnicode_DecodeFSDefaultAndSize()` and `PyUnicode_DecodeLocaleAndSize()` functions.

New in version 3.5.

char* **Py_EncodeLocale**(const wchar_t **text*, size_t **error_pos*)

Encode a wide character string to the locale encoding with the surrogateescape error handler: surrogate characters in the range U+DC80..U+DCFF are converted to bytes 0x80..0xFF.

Encoding, highest priority to lowest priority:

- `UTF-8` on macOS and Android;
- `ASCII` if the `LC_CTYPE` locale is `"C"`, `nl_langinfo(CODESET)` returns the `ASCII` encoding (or an alias), and `mbstowcs()` and `wcstombs()` functions uses the `ISO-8859-1` encoding.
- the current locale encoding.

Return a pointer to a newly allocated byte string, use `PyMem_Free()` to free the memory. Return `NULL` on encoding error or memory allocation error

If error_pos is not `NULL`, `*error_pos` is set to the index of the invalid character on encoding error, or set to `(size_t)-1` otherwise.

Use the `Py_DecodeLocale()` function to decode the bytes string back to a wide character string.

> **See also:** The `PyUnicode_EncodeFSDefault()` and `PyUnicode_EncodeLocale()` functions.

New in version 3.5.

System Functions

These are utility functions that make functionality from the `sys` module accessible to C code. They all work with the current interpreter thread's `sys` module's dict, which is contained in the internal thread state structure.

PyObject ***PySys_GetObject**(const char **name*)

Return value: Borrowed reference.

Return the object *name* from the `sys` module or *NULL* if it does not exist, without setting an exception.

int **PySys_SetObject**(const char **name*, PyObject **v*)

Set *name* in the `sys` module to *v* unless *v* is *NULL*, in which case *name* is deleted from the sys module. Returns `0` on success, `-1` on error.

void **PySys_ResetWarnOptions**()

Reset `sys.warnoptions` to an empty list.

void **PySys_AddWarnOption**(wchar_t **s*)

Append *s* to `sys.warnoptions`.

void **PySys_AddWarnOptionUnicode**(PyObject **unicode*)

Append *unicode* to `sys.warnoptions`.

void **PySys_SetPath**(wchar_t **path*)

Set `sys.path` to a list object of paths found in *path* which should be a list of paths separated with the platform's search path delimiter (`:` on Unix, `;` on Windows).

void **PySys_WriteStdout**(const char **format*, ...)

Write the output string described by *format* to `sys.stdout`. No exceptions are raised, even if truncation occurs (see below).

format should limit the total size of the formatted output string to 1000 bytes or less – after 1000 bytes, the output string is truncated. In particular, this means that no unrestricted “%s” formats should occur; these should be limited using “%.<N>s” where <N> is a decimal number calculated so that <N> plus the maximum size of other formatted text does not exceed 1000 bytes. Also watch out for “%f”, which can print hundreds of digits for very large numbers.

If a problem occurs, or `sys.stdout` is unset, the formatted message is written to the real (C level) *stdout*.

void **PySys_WriteStderr**(const char **format*, ...)

As `PySys_WriteStdout()`, but write to `sys.stderr` or *stderr* instead.

void **PySys_FormatStdout**(const char **format*, ...)

Function similar to PySys_WriteStdout() but format the message using `PyUnicode_FromFormatV()` and don’t truncate the message to an arbitrary length.

New in version 3.2.

void **PySys_FormatStderr**(const char **format*, ...)

As `PySys_FormatStdout()`, but write to `sys.stderr` or *stderr* instead.

New in version 3.2.

void **PySys_AddXOption**(const wchar_t **s*)

Parse *s* as a set of `-X` options and add them to the current options mapping as returned by `PySys_GetXOptions()`.

New in version 3.2.

PyObject ***PySys_GetXOptions**()

Return value: Borrowed reference.

Return the current dictionary of `-X` options, similarly to `sys._xoptions`. On error, *NULL* is returned and an exception is set.

New in version 3.2.

Process Control

void **Py_FatalError**(const char **message*)

Print a fatal error message and kill the process. No cleanup is performed. This function should only be invoked when a condition is detected that would make it dangerous to continue using the Python interpreter; e.g., when the object ad-

ministration appears to be corrupted. On Unix, the standard C library function `abort()` is called which will attempt to produce a `core` file.

void **Py_Exit**(int *status*)

Exit the current process. This calls `Py_FinalizeEx()` and then calls the standard C library function `exit(status)`. If `Py_FinalizeEx()` indicates an error, the exit status is set to 120.

Changed in version 3.6: Errors from finalization no longer ignored.

int **Py_AtExit**(void (**func*)())

Register a cleanup function to be called by `Py_FinalizeEx()`. The cleanup function will be called with no arguments and should return no value. At most 32 cleanup functions can be registered. When the registration is successful, `Py_AtExit()` returns `0`; on failure, it returns `-1`. The cleanup function registered last is called first. Each cleanup function will be called at most once. Since Python's internal finalization will have completed before the cleanup function, no Python APIs should be called by *func*.