

sys — System-specific parameters and functions

This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available. Unless explicitly noted otherwise, all variables are read-only.

sys.abiflags

On POSIX systems where Python was built with the standard `configure` script, this contains the ABI flags as specified by [PEP 3149](#).

Added in version 3.2.

Changed in version 3.8: Default flags became an empty string (m flag for pymalloc has been removed).

Availability: Unix.

sys.addaudithook(*hook*)

Append the callable *hook* to the list of active auditing hooks for the current (sub)interpreter.

When an auditing event is raised through the [`sys.audit\(\)`](#) function, each hook will be called in the order it was added with the event name and the tuple of arguments. Native hooks added by [`PySys_AddAuditHook\(\)`](#) are called first, followed by hooks added in the current (sub)interpreter. Hooks can then log the event, raise an exception to abort the operation, or terminate the process entirely.

Note that audit hooks are primarily for collecting information about internal or otherwise unobservable actions, whether by Python or libraries written in Python. They are not suitable for implementing a “sandbox”. In particular, malicious code can trivially disable or bypass hooks added using this function. At a minimum, any security-sensitive hooks must be added using the C API [`PySys_AddAuditHook\(\)`](#) before initialising the runtime, and any modules allowing arbitrary memory modification (such as [`ctypes`](#)) should be completely removed or closely monitored.

Calling [`sys.addaudithook\(\)`](#) will itself raise an auditing event named `sys.addaudithook` with no arguments. If any existing hooks raise an exception derived from [`RuntimeError`](#), the new hook will not be added and the exception suppressed. As a result, callers cannot assume that their hook has been added unless they control all existing hooks.

See the [audit events table](#) for all events raised by CPython, and [PEP 578](#) for the original design discussion.

Added in version 3.8.

Changed in version 3.8.1: Exceptions derived from [`Exception`](#) but not [`RuntimeError`](#) are no longer suppressed.

CPython implementation detail: When tracing is enabled (see [`settrace\(\)`](#)), Python hooks are only traced if the callable has a `__cantrace__` member that is set to a true value. Otherwise, trace functions will skip the hook.

sys.argv

The list of command line arguments passed to a Python script. `argv[0]` is the script name (it is operating system dependent whether this is a full pathname or not). If the command was executed using the `-c` command line option to the interpreter, `argv[0]` is set to the string '`-c`'. If no script name was passed to the Python interpreter, `argv[0]` is the empty string.

To loop over the standard input, or the list of files given on the command line, see the [fileinput](#) module.

See also [sys.orig_argv](#).

Note: On Unix, command line arguments are passed by bytes from OS. Python decodes them with filesystem encoding and "surrogateescape" error handler. When you need original bytes, you can get it by `[os.fsendcode(arg) for arg in sys.argv]`.

sys.audit(event, *args)

Raise an auditing event and trigger any active auditing hooks. `event` is a string identifying the event, and `args` may contain optional arguments with more information about the event. The number and types of arguments for a given event are considered a public and stable API and should not be modified between releases.

For example, one auditing event is named `os.chdir`. This event has one argument called `path` that will contain the requested new working directory.

[sys.audit\(\)](#) will call the existing auditing hooks, passing the event name and arguments, and will re-raise the first exception from any hook. In general, if an exception is raised, it should not be handled and the process should be terminated as quickly as possible. This allows hook implementations to decide how to respond to particular events: they can merely log the event or abort the operation by raising an exception.

Hooks are added using the [sys.addaudithook\(\)](#) or [PySys_AddAuditHook\(\)](#) functions.

The native equivalent of this function is [PySys_Audit\(\)](#). Using the native function is preferred when possible.

See the [audit events table](#) for all events raised by CPython.

Added in version 3.8.

sys.base_exec_prefix

Equivalent to [exec_prefix](#), but referring to the base Python installation.

When running under [Virtual Environments](#), [exec_prefix](#) gets overwritten to the virtual environment prefix. [base_exec_prefix](#), conversely, does not change, and always points to the base Python installation. Refer to [Virtual Environments](#) for more information.

Added in version 3.3.

sys.base_prefix

Equivalent to [prefix](#), but referring to the base Python installation.

When running under [virtual environment](#), `prefix` gets overwritten to the virtual environment prefix. `base_prefix`, conversely, does not change, and always points to the base Python installation. Refer to [Virtual Environments](#) for more information.

Added in version 3.3.

`sys.byteorder`

An indicator of the native byte order. This will have the value 'big' on big-endian (most-significant byte first) platforms, and 'little' on little-endian (least-significant byte first) platforms.

`sys.builtin_module_names`

A tuple of strings containing the names of all modules that are compiled into this Python interpreter. (This information is not available in any other way — `modules.keys()` only lists the imported modules.)

See also the [sys.stdlib_module_names](#) list.

`sys.call_tracing(func, args)`

Call `func(*args)`, while tracing is enabled. The tracing state is saved, and restored afterwards. This is intended to be called from a debugger from a checkpoint, to recursively debug or profile some other code.

Tracing is suspended while calling a tracing function set by [settrace\(\)](#) or [setprofile\(\)](#) to avoid infinite recursion. `call_tracing()` enables explicit recursion of the tracing function.

`sys.copyright`

A string containing the copyright pertaining to the Python interpreter.

`sys._clear_type_cache()`

Clear the internal type cache. The type cache is used to speed up attribute and method lookups. Use the function *only* to drop unnecessary references during reference leak debugging.

This function should be used for internal and specialized purposes only.

Deprecated since version 3.13: Use the more general [_clear_internal_caches\(\)](#) function instead.

`sys._clear_internal_caches()`

Clear all internal performance-related caches. Use this function *only* to release unnecessary references and memory blocks when hunting for leaks.

Added in version 3.13.

`sys._current_frames()`

Return a dictionary mapping each thread's identifier to the topmost stack frame currently active in that thread at the time the function is called. Note that functions in the [traceback](#) module can build the call stack given such a frame.

This is most useful for debugging deadlock: this function does not require the deadlocked threads' cooperation, and such threads' call stacks are frozen for as long as they remain deadlocked. The frame returned for a non-deadlocked thread may bear no relationship to that thread's current activity by the time calling code examines the frame.

This function should be used for internal and specialized purposes only.

Raises an [auditing event](#) `sys._current_frames` with no arguments.

`sys._current_exceptions()`

Return a dictionary mapping each thread's identifier to the topmost exception currently active in that thread at the time the function is called. If a thread is not currently handling an exception, it is not included in the result dictionary.

This is most useful for statistical profiling.

This function should be used for internal and specialized purposes only.

Raises an [auditing event](#) `sys._current_exceptions` with no arguments.

Changed in version 3.12: Each value in the dictionary is now a single exception instance, rather than a 3-tuple as returned from `sys.exc_info()`.

`sys.breakpointhook()`

This hook function is called by built-in [`breakpoint\(\)`](#). By default, it drops you into the [`pdb`](#) debugger, but it can be set to any other function so that you can choose which debugger gets used.

The signature of this function is dependent on what it calls. For example, the default binding (e.g. `pdb.set_trace()`) expects no arguments, but you might bind it to a function that expects additional arguments (positional and/or keyword). The built-in `breakpoint()` function passes its `*args` and `**kws` straight through. Whatever `breakpointhooks()` returns is returned from `breakpoint()`.

The default implementation first consults the environment variable [`PYTHONBREAKPOINT`](#). If that is set to "0" then this function returns immediately; i.e. it is a no-op. If the environment variable is not set, or is set to the empty string, `pdb.set_trace()` is called. Otherwise this variable should name a function to run, using Python's dotted-import nomenclature, e.g. `package.subpackage.module.function`. In this case, `package.subpackage.module` would be imported and the resulting module must have a callable named `function()`. This is run, passing in `*args` and `**kws`, and whatever `function()` returns, `sys.breakpointhook()` returns to the built-in [`breakpoint\(\)`](#) function.

Note that if anything goes wrong while importing the callable named by [`PYTHONBREAKPOINT`](#), a [`RuntimeWarning`](#) is reported and the breakpoint is ignored.

Also note that if `sys.breakpointhook()` is overridden programmatically, [`PYTHONBREAKPOINT`](#) is not consulted.

Added in version 3.7.

`sys._debugmallocstats()`

Print low-level information to `stderr` about the state of CPython's memory allocator.

If Python is [`built in debug mode`](#) (`configure --with-pydebug` option), it also performs some expensive internal consistency checks.

Added in version 3.3.

CPython implementation detail: This function is specific to CPython. The exact output format is not defined here, and may change.

sys.dllhandle

Integer specifying the handle of the Python DLL.

Availability: Windows.

sys.displayhook(*value*)

If *value* is not None, this function prints `repr(value)` to `sys.stdout`, and saves *value* in `builtins._`. If `repr(value)` is not encodable to `sys.stdout.encoding` with `sys.stdout.errors` error handler (which is probably 'strict'), encode it to `sys.stdout.encoding` with 'backslashreplace' error handler.

`sys.displayhook` is called on the result of evaluating an expression entered in an interactive Python session. The display of these values can be customized by assigning another one-argument function to `sys.displayhook`.

Pseudo-code:

```
def displayhook(value):
    if value is None:
        return
    # Set '_' to None to avoid recursion
    builtins._ = None
    text = repr(value)
    try:
        sys.stdout.write(text)
    except UnicodeEncodeError:
        bytes = text.encode(sys.stdout.encoding, 'backslashreplace')
        if hasattr(sys.stdout, 'buffer'):
            sys.stdout.buffer.write(bytes)
        else:
            text = bytes.decode(sys.stdout.encoding, 'strict')
            sys.stdout.write(text)
    sys.stdout.write("\n")
    builtins._ = value
```

Changed in version 3.2: Use 'backslashreplace' error handler on UnicodeEncodeError.

sys.dont_write_bytecode

If this is true, Python won't try to write .pyc files on the import of source modules. This value is initially set to True or False depending on the `-B` command line option and the `PYTHONDONTWRITEBYTECODE` environment variable, but you can set it yourself to control bytecode file generation.

sys._emscripten_info

A named tuple holding information about the environment on the *wasm32-emscripten* platform. The named tuple is provisional and may change in the future.

_emscripten_info.emscripten_version

Emscripten version as tuple of ints (major, minor, micro), e.g. (3, 1, 8).

_emscripten_info.runtime

Runtime string, e.g. browser user agent, 'Node.js v14.18.2', or 'UNKNOWN'.

_emscripten_info.pthreads

True if Python is compiled with Emscripten pthreads support.

`_emscripten_info.shared_memory`

True if Python is compiled with shared memory support.

Availability: Emscripten.

Added in version 3.11.

`sys.pycache_prefix`

If this is set (not `None`), Python will write bytecode-cache `.pyc` files to (and read them from) a parallel directory tree rooted at this directory, rather than from `__pycache__` directories in the source code tree. Any `__pycache__` directories in the source code tree will be ignored and new `.pyc` files written within the pycache prefix. Thus if you use `compileall` as a pre-build step, you must ensure you run it with the same pycache prefix (if any) that you will use at runtime.

A relative path is interpreted relative to the current working directory.

This value is initially set based on the value of the `-X pycache_prefix=PATH` command-line option or the `PYTHONPYCACHEPREFIX` environment variable (command-line takes precedence). If neither are set, it is `None`.

Added in version 3.8.

`sys.excepthook(type, value, traceback)`

This function prints out a given traceback and exception to `sys.stderr`.

When an exception other than `SystemExit` is raised and uncaught, the interpreter calls `sys.excepthook` with three arguments, the exception class, exception instance, and a traceback object. In an interactive session this happens just before control is returned to the prompt; in a Python program this happens just before the program exits. The handling of such top-level exceptions can be customized by assigning another three-argument function to `sys.excepthook`.

Raise an auditing event `sys.excepthook` with arguments `hook`, `type`, `value`, `traceback` when an uncaught exception occurs. If no hook has been set, `hook` may be `None`. If any hook raises an exception derived from `RuntimeError` the call to the hook will be suppressed. Otherwise, the audit hook exception will be reported as unraisable and `sys.excepthook` will be called.

See also: The `sys.unraisablehook()` function handles unraisable exceptions and the `threading.excepthook()` function handles exception raised by `threading.Thread.run()`.

`sys.__breakpointhook__` `sys.__displayhook__` `sys.__excepthook__` `sys.__unraisablehook__`

These objects contain the original values of `breakpointhook`, `displayhook`, `excepthook`, and `unraisablehook` at the start of the program. They are saved so that `breakpointhook`, `displayhook` and `excepthook`, `unraisablehook` can be restored in case they happen to get replaced with broken or alternative objects.

Added in version 3.7: `__breakpointhook__`

Added in version 3.8: `__unraisablehook__`

sys.exception()

This function, when called while an exception handler is executing (such as an `except` or `except*` clause), returns the exception instance that was caught by this handler. When exception handlers are nested within one another, only the exception handled by the innermost handler is accessible.

If no exception handler is executing, this function returns `None`.

Added in version 3.11.

sys.exc_info()

This function returns the old-style representation of the handled exception. If an exception `e` is currently handled (so `exception()` would return `e`), `exc_info()` returns the tuple `(type(e), e, e.__traceback__)`. That is, a tuple containing the type of the exception (a subclass of `BaseException`), the exception itself, and a `traceback object` which typically encapsulates the call stack at the point where the exception last occurred.

If no exception is being handled anywhere on the stack, this function return a tuple containing three `None` values.

Changed in version 3.11: The `type` and `traceback` fields are now derived from the `value` (the exception instance), so when an exception is modified while it is being handled, the changes are reflected in the results of subsequent calls to `exc_info()`.

sys.exec_prefix

A string giving the site-specific directory prefix where the platform-dependent Python files are installed; by default, this is also '`/usr/local`'. This can be set at build time with the `--exec-prefix` argument to the `configure` script. Specifically, all configuration files (e.g. the `pyconfig.h` header file) are installed in the directory `exec_prefix/lib/pythonX.Y/config`, and shared library modules are installed in `exec_prefix/lib/pythonX.Y/lib-dynload`, where `X.Y` is the version number of Python, for example 3.2.

Note: If a `virtual environment` is in effect, this `exec_prefix` will point to the virtual environment. The value for the Python installation will still be available, via `base_exec_prefix`. Refer to [Virtual Environments](#) for more information.

Changed in version 3.14: When running under a `virtual environment`, `prefix` and `exec_prefix` are now set to the virtual environment prefix by the `path initialization`, instead of `site`. This means that `prefix` and `exec_prefix` always point to the virtual environment, even when `site` is disabled (`-S`).

sys.executable

A string giving the absolute path of the executable binary for the Python interpreter, on systems where this makes sense. If Python is unable to retrieve the real path to its executable, `sys.executable` will be an empty string or `None`.

sys.exit([arg])

Raise a `SystemExit` exception, signaling an intention to exit the interpreter.

The optional argument `arg` can be an integer giving the exit status (defaulting to zero), or another type of object. If it is an integer, zero is considered "successful termination" and any nonzero value is considered "abnormal termination" by shells and the like. Most systems require it to be in the range 0–127, and pro-

duce undefined results otherwise. Some systems have a convention for assigning specific meanings to specific exit codes, but these are generally underdeveloped; Unix programs generally use 2 for command line syntax errors and 1 for all other kind of errors. If another type of object is passed, `None` is equivalent to passing zero, and any other object is printed to `stderr` and results in an exit code of 1. In particular, `sys.exit("some error message")` is a quick way to exit a program when an error occurs.

Since `exit()` ultimately “only” raises an exception, it will only exit the process when called from the main thread, and the exception is not intercepted. Cleanup actions specified by finally clauses of `try` statements are honored, and it is possible to intercept the exit attempt at an outer level.

Changed in version 3.6: If an error occurs in the cleanup after the Python interpreter has caught `SystemExit` (such as an error flushing buffered data in the standard streams), the exit status is changed to 120.

sys.flags

The `named tuple` `flags` exposes the status of command line flags. Flags should only be accessed only by name and not by index. The attributes are read only.

<code>flags.debug</code>	<code>-d</code>
<code>flags.inspect</code>	<code>-i</code>
<code>flags.interactive</code>	<code>-i</code>
<code>flags.isolated</code>	<code>-I</code>
<code>flags.optimize</code>	<code>-O</code> or <code>-OO</code>
<code>flags.dont_write_bytecode</code>	<code>-B</code>
<code>flags.no_user_site</code>	<code>-s</code>
<code>flags.no_site</code>	<code>-S</code>
<code>flags.ignore_environment</code>	<code>-E</code>
<code>flags.verbose</code>	<code>-v</code>
<code>flags.bytes_warning</code>	<code>-b</code>
<code>flags.quiet</code>	<code>-q</code>
<code>flags.hash_randomization</code>	<code>-R</code>
<code>flags.dev_mode</code>	<code>-X dev</code> (Python Development Mode)
<code>flags.utf8_mode</code>	<code>-X utf8</code>
<code>flags.safe_path</code>	<code>-P</code>
<code>flags.int_max_str_digits</code>	<code>-X int_max_str_digits</code> (integer string conversion length limitation)

<code>flags.warn_default_encoding</code>	<code>-X warn_default_encoding</code>
<code>flags.gil</code>	<code>-X gil</code> and <code>PYTHON_GIL</code>
<code>flags.thread_inherit_context</code>	<code>-X thread_inherit_context</code> and <code>PYTHON_THREAD_INHERIT_CONTEXT</code>
<code>flags.context_aware_warnings</code>	<code>-X context_aware_warnings</code> and <code>PYTHON_CONTEXT_AWARE_WARNINGS</code>

Changed in version 3.2: Added `quiet` attribute for the new `-q` flag.

Added in version 3.2.3: The `hash_randomization` attribute.

Changed in version 3.3: Removed obsolete `division_warning` attribute.

Changed in version 3.4: Added `isolated` attribute for `-I` isolated flag.

Changed in version 3.7: Added the `dev_mode` attribute for the new [Python Development Mode](#) and the `utf8_mode` attribute for the new `-X utf8` flag.

Changed in version 3.10: Added `warn_default_encoding` attribute for `-X warn_default_encoding` flag.

Changed in version 3.11: Added the `safe_path` attribute for `-P` option.

Changed in version 3.11: Added the `int_max_str_digits` attribute.

Changed in version 3.13: Added the `gil` attribute.

Changed in version 3.14: Added the `thread_inherit_context` attribute.

Changed in version 3.14: Added the `context_aware_warnings` attribute.

sys.float_info

A [named tuple](#) holding information about the float type. It contains low level information about the precision and internal representation. The values correspond to the various floating-point constants defined in the standard header file `float.h` for the ‘C’ programming language; see section 5.2.4.2.2 of the 1999 ISO/IEC C standard [\[C99\]](#), ‘Characteristics of floating types’, for details.

Attributes of the `float_info` [named tuple](#)

attribute	float.h macro	explanation
<code>float_info.epsilon</code>	<code>DBL_EPSILON</code>	difference between 1.0 and the least value greater than 1.0 that is representable as a float. See also math.ulp() .
<code>float_info.dig</code>	<code>DBL_DIG</code>	The maximum number of decimal digits that can be faithfully represented in a float; see below.
<code>float_info.mant_dig</code>	<code>DBL_MANT_DIG</code>	Float precision: the number of base-radix digits in the significand of a float.

attribute	float.h macro	explanation
<code>float_info.max</code>	DBL_MAX	The maximum representable positive finite float.
<code>float_info.max_exp</code>	DBL_MAX_EXP	The maximum integer e such that radix^{e-1} is a representable finite float.
<code>float_info.max_10_exp</code>	DBL_MAX_10_EXP	The maximum integer e such that 10^e is in the range of representable finite floats.
<code>float_info.min</code>	DBL_MIN	The minimum representable positive <i>normalized</i> float. Use <code>math.ulp(0.0)</code> to get the smallest positive <i>denormalized</i> representable float.
<code>float_info.min_exp</code>	DBL_MIN_EXP	The minimum integer e such that radix^{e-1} is a normalized float.
<code>float_info.min_10_exp</code>	DBL_MIN_10_EXP	The minimum integer e such that 10^e is a normalized float.
<code>float_info.radix</code>	FLT_RADIX	The radix of exponent representation.
<code>float_info.rounds</code>	FLT_ROUNDS	An integer representing the rounding mode for floating-point arithmetic. This reflects the value of the system <code>FLT_ROUNDS</code> macro at interpreter startup time: <ul style="list-style-type: none"> • -1: indeterminable • 0: toward zero • 1: to nearest • 2: toward positive infinity • 3: toward negative infinity All other values for <code>FLT_ROUNDS</code> characterize implementation-defined rounding behavior.

The attribute `sys.float_info.dig` needs further explanation. If `s` is any string representing a decimal number with at most `sys.float_info.dig` significant digits, then converting `s` to a float and back again will recover a string representing the same decimal value:

```
>>> import sys
>>> sys.float_info.dig
15
>>> s = '3.14159265358979'      # decimal string with 15 significant digits
>>> format(float(s), '.15g')    # convert to float and back -> same value
'3.14159265358979'
```

But for strings with more than `sys.float_info.dig` significant digits, this isn't always true:

```
>>> s = '9876543211234567'      # 16 significant digits is too many!
>>> format(float(s), '.16g')    # conversion changes value
```

'9876543211234568'

sys.**float_repr_style**

A string indicating how the `repr()` function behaves for floats. If the string has value 'short' then for a finite float `x`, `repr(x)` aims to produce a short string with the property that `float(repr(x)) == x`. This is the usual behaviour in Python 3.1 and later. Otherwise, `float_repr_style` has value 'legacy' and `repr(x)` behaves in the same way as it did in versions of Python prior to 3.1.

Added in version 3.1.

sys.**getallocatedblocks()**

Return the number of memory blocks currently allocated by the interpreter, regardless of their size. This function is mainly useful for tracking and debugging memory leaks. Because of the interpreter's internal caches, the result can vary from call to call; you may have to call `clear_internal_caches()` and `gc.collect()` to get more predictable results.

If a Python build or implementation cannot reasonably compute this information, `getallocatedblocks()` is allowed to return 0 instead.

Added in version 3.4.

sys.**getunicodeinternedsize()**

Return the number of unicode objects that have been interned.

Added in version 3.12.

sys.**getandroidapilevel()**

Return the build-time API level of Android as an integer. This represents the minimum version of Android this build of Python can run on. For runtime version information, see `platform.android_ver()`.

Availability: Android.

Added in version 3.7.

sys.**getdefaultencoding()**

Return 'utf-8'. This is the name of the default string encoding, used in methods like `str.encode()`.

sys.**getdlopenflags()**

Return the current value of the flags that are used for `dlopen()` calls. Symbolic names for the flag values can be found in the `os` module (RTLD_XXX constants, e.g. `os.RTLD_LAZY`).

Availability: Unix.

sys.**getfilesystemencoding()**

Get the `filesystem encoding`: the encoding used with the `filesystem error handler` to convert between Unicode filenames and bytes filenames. The filesystem error handler is returned from `getfilesystemencodeerrors()`.

For best compatibility, `str` should be used for filenames in all cases, although representing filenames as bytes is also supported. Functions accepting or returning filenames should support either `str` or `bytes` and internally convert to the system's preferred representation.

`os.fsencode()` and `os.fsdecode()` should be used to ensure that the correct encoding and errors mode are used.

The [filesystem encoding and error handler](#) are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

Changed in version 3.2: `getfilesystemencoding()` result cannot be `None` anymore.

Changed in version 3.6: Windows is no longer guaranteed to return '`mbcs`'. See [PEP 529](#) and `enablelegacywindowsfsencoding()` for more information.

Changed in version 3.7: Return '`utf-8`' if the [Python UTF-8 Mode](#) is enabled.

`sys.getfilesystemencodeerrors()`

Get the [filesystem error handler](#): the error handler used with the [filesystem encoding](#) to convert between Unicode filenames and bytes filenames. The filesystem encoding is returned from `getfilesystemencoding()`.

`os.fsencode()` and `os.fsdecode()` should be used to ensure that the correct encoding and errors mode are used.

The [filesystem encoding and error handler](#) are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

Added in version 3.6.

`sys.get_int_max_str_digits()`

Returns the current value for the [integer string conversion length limitation](#). See also `set_int_max_str_digits()`.

Added in version 3.11.

`sys.getrefcount(object)`

Return the reference count of the `object`. The count returned is generally one higher than you might expect, because it includes the (temporary) reference as an argument to `getrefcount()`.

Note that the returned value may not actually reflect how many references to the object are actually held. For example, some objects are [immortal](#) and have a very high refcount that does not reflect the actual number of references. Consequently, do not rely on the returned value to be accurate, other than a value of 0 or 1.

CPython implementation detail: [Immortal](#) objects with a large reference count can be identified via `is_immortal()`.

Changed in version 3.12: Immortal objects have very large refcounts that do not match the actual number of references to the object.

`sys.getrecursionlimit()`

Return the current value of the recursion limit, the maximum depth of the Python interpreter stack. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python. It can be set by `setrecursionlimit()`.

sys.getsizeof(object[, default])

Return the size of an object in bytes. The object can be any type of object. All built-in objects will return correct results, but this does not have to hold true for third-party extensions as it is implementation specific.

Only the memory consumption directly attributed to the object is accounted for, not the memory consumption of objects it refers to.

If given, *default* will be returned if the object does not provide means to retrieve the size. Otherwise a [TypeError](#) will be raised.

`getsizeof()` calls the object's `__sizeof__` method and adds an additional garbage collector overhead if the object is managed by the garbage collector.

See [recursive sizeof recipe](#) for an example of using `getsizeof()` recursively to find the size of containers and all their contents.

sys.getswitchinterval()

Return the interpreter's "thread switch interval" in seconds; see [setswitchinterval\(\)](#).

Added in version 3.2.

sys._getframe([depth])

Return a frame object from the call stack. If optional integer *depth* is given, return the frame object that many calls below the top of the stack. If that is deeper than the call stack, [ValueError](#) is raised. The default for *depth* is zero, returning the frame at the top of the call stack.

Raises an [auditing event](#) `sys._getframe` with argument `frame`.

CPython implementation detail: This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.

sys._getframemodulename([depth])

Return the name of a module from the call stack. If optional integer *depth* is given, return the module that many calls below the top of the stack. If that is deeper than the call stack, or if the module is unidentifiable, `None` is returned. The default for *depth* is zero, returning the module at the top of the call stack.

Raises an [auditing event](#) `sys._getframemodulename` with argument `depth`.

CPython implementation detail: This function should be used for internal and specialized purposes only. It is not guaranteed to exist in all implementations of Python.

Added in version 3.12.

sys.getobjects(limit[, type])

This function only exists if CPython was built using the specialized configure option [--with-trace-refs](#). It is intended only for debugging garbage-collection issues.

Return a list of up to *limit* dynamically allocated Python objects. If *type* is given, only objects of that exact type (not subtypes) are included.

Objects from the list are not safe to use. Specifically, the result will include objects from all interpreters that share their object allocator state (that is, ones created with `PyInterpreterConfig.use_main_omalloc` set to 1 or using `Py_NewInterpreter()`, and the [main interpreter](#)). Mixing objects from different interpreters may lead to crashes or other unexpected behavior.

CPython implementation detail: This function should be used for specialized purposes only. It is not guaranteed to exist in all implementations of Python.

Changed in version 3.14: The result may include objects from other interpreters.

`sys.getprofile()`

Get the profiler function as set by [setprofile\(\)](#).

`sys.gettrace()`

Get the trace function as set by [settrace\(\)](#).

CPython implementation detail: The `gettrace()` function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

`sys.getwindowsversion()`

Return a named tuple describing the Windows version currently running. The named elements are `major`, `minor`, `build`, `platform`, `service_pack`, `service_pack_minor`, `service_pack_major`, `suite_mask`, `product_type` and `platform_version`. `service_pack` contains a string, `platform_version` a 3-tuple and all other values are integers. The components can also be accessed by name, so `sys.getwindowsversion()[0]` is equivalent to `sys.getwindowsversion().major`. For compatibility with prior versions, only the first 5 elements are retrievable by indexing.

`platform` will be 2 (VER_PLATFORM_WIN32_NT).

`product_type` may be one of the following values:

Constant	Meaning
1 (VER_NT_WORKSTATION)	The system is a workstation.
2 (VER_NT_DOMAIN_CONTROLLER)	The system is a domain controller.
3 (VER_NT_SERVER)	The system is a server, but not a domain controller.

This function wraps the Win32 `GetVersionEx()` function; see the Microsoft documentation on `OSVERSIONINFOEX()` for more information about these fields.

`platform_version` returns the major version, minor version and build number of the current operating system, rather than the version that is being emulated for the process. It is intended for use in logging rather than for feature detection.

Note: `platform_version` derives the version from `kernel32.dll` which can be of a different version than the OS version. Please use [platform](#) module for achieving accurate OS version.

Availability: Windows.

Changed in version 3.2: Changed to a named tuple and added `service_pack_minor`, `service_pack_major`, `suite_mask`, and `product_type`.

Changed in version 3.6: Added `platform_version`

`sys.get_asyncgen_hooks()`

Returns an `asyncgen_hooks` object, which is similar to a [namedtuple](#) of the form `(firstiter, finalizer)`, where `firstiter` and `finalizer` are expected to be either `None` or functions which take an [asynchronous generator iterator](#) as an argument, and are used to schedule finalization of an asynchronous generator by an event loop.

Added in version 3.6: See [PEP 525](#) for more details.

Note: This function has been added on a provisional basis (see [PEP 411](#) for details.)

`sys.getCoroutine_origin_tracking_depth()`

Get the current coroutine origin tracking depth, as set by [set_coroutine_origin_tracking_depth\(\)](#).

Added in version 3.7.

Note: This function has been added on a provisional basis (see [PEP 411](#) for details.) Use it only for debugging purposes.

`sys.hash_info`

A [named tuple](#) giving parameters of the numeric hash implementation. For more details about hashing of numeric types, see [Hashing of numeric types](#).

`hash_info.width`

The width in bits used for hash values

`hash_info.modulus`

The prime modulus P used for numeric hash scheme

`hash_info.inf`

The hash value returned for a positive infinity

`hash_info.nan`

(This attribute is no longer used)

`hash_info.imag`

The multiplier used for the imaginary part of a complex number

`hash_info.algorithm`

The name of the algorithm for hashing of str, bytes, and memoryview

`hash_info.hash_bits`

The internal output size of the hash algorithm

`hash_info.seed_bits`

The size of the seed key of the hash algorithm

Added in version 3.2.

Changed in version 3.4: Added `algorithm`, `hash_bits` and `seed_bits`

sys.hexversion

The version number encoded as a single integer. This is guaranteed to increase with each version, including proper support for non-production releases. For example, to test that the Python interpreter is at least version 1.5.2, use:

```
if sys.hexversion >= 0x010502F0:  
    # use some advanced feature  
    ...  
else:  
    # use an alternative implementation or warn the user  
    ...
```

This is called `hexversion` since it only really looks meaningful when viewed as the result of passing it to the built-in `hex()` function. The [named tuple](#) `sys.version_info` may be used for a more human-friendly encoding of the same information.

More details of `hexversion` can be found at [API and ABI Versioning](#).

sys.implementation

An object containing information about the implementation of the currently running Python interpreter. The following attributes are required to exist in all Python implementations.

`name` is the implementation's identifier, e.g. '`cpython`'. The actual string is defined by the Python implementation, but it is guaranteed to be lower case.

`version` is a named tuple, in the same format as `sys.version_info`. It represents the version of the Python *implementation*. This has a distinct meaning from the specific version of the Python *language* to which the currently running interpreter conforms, which `sys.version_info` represents. For example, for PyPy 1.8 `sys.implementation.version` might be `sys.version_info(1, 8, 0, 'final', 0)`, whereas `sys.version_info` would be `sys.version_info(2, 7, 2, 'final', 0)`. For CPython they are the same value, since it is the reference implementation.

`hexversion` is the implementation version in hexadecimal format, like `sys.hexversion`.

`cache_tag` is the tag used by the import machinery in the filenames of cached modules. By convention, it would be a composite of the implementation's name and version, like '`cpython-33`'. However, a Python implementation may use some other value if appropriate. If `cache_tag` is set to `None`, it indicates that module caching should be disabled.

`supports_isolated_interpreters` is a boolean value, whether this implementation supports multiple isolated interpreters. It is `True` for CPython on most platforms. Platforms with this support implement the low-level `_interpreters` module.

See also: [PEP 684](#), [PEP 734](#), and [concurrent.interpreters](#).

`sys.implementation` may contain additional attributes specific to the Python implementation. These non-standard attributes must start with an underscore, and are not described here. Regardless of its contents, `sys.implementation` will not change during a run of the interpreter, nor between implementation versions. (It may change between Python language versions, however.) See [PEP 421](#) for more information.

Added in version 3.3.

Changed in version 3.14: Added `supports_isolated_interpreters` field.

Note: The addition of new required attributes must go through the normal PEP process. See [PEP 421](#) for more information.

sys.int_info

A [named tuple](#) that holds information about Python's internal representation of integers. The attributes are read only.

int_info.bits_per_digit

The number of bits held in each digit. Python integers are stored internally in base $2^{*\text{int_info.bits_per_digit}}$.

int_info.sizeof_digit

The size in bytes of the C type used to represent a digit.

int_info.default_max_str_digits

The default value for `sys.get_int_max_str_digits()` when it is not otherwise explicitly configured.

int_info.str_digits_check_threshold

The minimum non-zero value for `sys.set_int_max_str_digits()`, `PYTHONINTMAXSTRDIGITS`, or `_X_int_max_str_digits`.

Added in version 3.1.

Changed in version 3.11: Added `default_max_str_digits` and `str_digits_check_threshold`.

sys.__interactivehook__

When this attribute exists, its value is automatically called (with no arguments) when the interpreter is launched in [interactive mode](#). This is done after the `PYTHONSTARTUP` file is read, so that you can set this hook there. The `site` module [sets this](#).

Raises an [auditing event](#) `cpython.run_interactivehook` with the hook object as the argument when the hook is called on startup.

Added in version 3.4.

sys.intern(string)

Enter `string` in the table of "interned" strings and return the interned string – which is `string` itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

Interned strings are not [immortal](#); you must keep a reference to the return value of [`intern\(\)`](#) around to benefit from it.

`sys._is_gil_enabled()`

Return `True` if the [GIL](#) is enabled and `False` if it is disabled.

Added in version 3.13.

CPython implementation detail: It is not guaranteed to exist in all implementations of Python.

`sys.is_finalizing()`

Return `True` if the main Python interpreter is [shutting down](#). Return `False` otherwise.

See also the [PythonFinalizationError](#) exception.

Added in version 3.5.

`sys._jit`

Utilities for observing just-in-time compilation.

CPython implementation detail: JIT compilation is an *experimental implementation detail* of CPython.

`sys._jit` is not guaranteed to exist or behave the same way in all Python implementations, versions, or build configurations.

Added in version 3.14.

`_jit.is_available()`

Return `True` if the current Python executable supports JIT compilation, and `False` otherwise. This can be controlled by building CPython with the `--experimental-jit` option on Windows, and the `--enable-experimental-jit` option on all other platforms.

`_jit.is_enabled()`

Return `True` if JIT compilation is enabled for the current Python process (implies `sys._jit.is_available()`), and `False` otherwise. If JIT compilation is available, this can be controlled by setting the `PYTHON_JIT` environment variable to 0 (disabled) or 1 (enabled) at interpreter startup.

`_jit.is_active()`

Return `True` if the topmost Python frame is currently executing JIT code (implies `sys._jit.is_enabled()`), and `False` otherwise.

Note: This function is intended for testing and debugging the JIT itself. It should be avoided for any other purpose.

Note: Due to the nature of tracing JIT compilers, repeated calls to this function may give surprising results. For example, branching on its return value will likely lead to unexpected behavior (if doing so causes JIT code to be entered or exited):

```
>>> for warmup in range(BIG_NUMBER):
...     # This line is "hot", and is eventually JIT-compiled:
```

```

...
    if sys._jit.is_active():
...
        # This Line is "cold", and is run in the interpreter:
...
        assert sys._jit.is_active()
...
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
    assert sys._jit.is_active()
    ~~~~~^
AssertionError

```

sys.last_exc

This variable is not always defined; it is set to the exception instance when an exception is not handled and the interpreter prints an error message and a stack traceback. Its intended use is to allow an interactive user to import a debugger module and engage in post-mortem debugging without having to re-execute the command that caused the error. (Typical use is `import pdb; pdb.pm()` to enter the post-mortem debugger; see [pdb](#) module for more information.)

Added in version 3.12.

sys._is_immortal(op)

Return `True` if the given object is [immortal](#), `False` otherwise.

Note: Objects that are immortal (and thus return `True` upon being passed to this function) are not guaranteed to be immortal in future versions, and vice versa for mortal objects.

Added in version 3.14.

CPython implementation detail: This function should be used for specialized purposes only. It is not guaranteed to exist in all implementations of Python.

sys._is_interned(string)

Return `True` if the given string is “interned”, `False` otherwise.

Added in version 3.13.

CPython implementation detail: It is not guaranteed to exist in all implementations of Python.

sys.last_type sys.last_value sys.last_traceback

These three variables are deprecated; use [sys.last_exc](#) instead. They hold the legacy representation of [sys.last_exc](#), as returned from [exc_info\(\)](#) above.

sys.maxsize

An integer giving the maximum value a variable of type [Py_ssize_t](#) can take. It's usually $2^{31} - 1$ on a 32-bit platform and $2^{63} - 1$ on a 64-bit platform.

sys.maxunicode

An integer giving the value of the largest Unicode code point, i.e. 1114111 (0x10FFFF in hexadecimal).

Changed in version 3.3: Before [PEP 393](#), `sys.maxunicode` used to be either 0xFFFF or 0x10FFFF, depending on the configuration option that specified whether Unicode characters were stored as UCS-2 or UCS-4.

sys.meta_path

A list of [meta_path finder](#) objects that have their [find_spec\(\)](#) methods called to see if one of the objects can find the module to be imported. By default, it holds entries that implement Python's default import semantics. The [find_spec\(\)](#) method is called with at least the absolute name of the module being imported. If the module to be imported is contained in a package, then the parent package's [__path__](#) attribute is passed in as a second argument. The method returns a [module spec](#), or None if the module cannot be found.

See also:

[importlib.abc.MetaPathFinder](#)

The abstract base class defining the interface of finder objects on [meta_path](#).

[importlib.machinery.ModuleSpec](#)

The concrete class which [find_spec\(\)](#) should return instances of.

Changed in version 3.4: [Module specs](#) were introduced in Python 3.4, by [PEP 451](#).

Changed in version 3.12: Removed the fallback that looked for a [find_module\(\)](#) method if a [meta_path](#) entry didn't have a [find_spec\(\)](#) method.

sys.modules

This is a dictionary that maps module names to modules which have already been loaded. This can be manipulated to force reloading of modules and other tricks. However, replacing the dictionary will not necessarily work as expected and deleting essential items from the dictionary may cause Python to fail. If you want to iterate over this global dictionary always use [sys.modules.copy\(\)](#) or [tuple\(sys.modules\)](#) to avoid exceptions as its size may change during iteration as a side effect of code or activity in other threads.

sys.orig_argv

The list of the original command line arguments passed to the Python executable.

The elements of [sys.orig_argv](#) are the arguments to the Python interpreter, while the elements of [sys.argv](#) are the arguments to the user's program. Arguments consumed by the interpreter itself will be present in [sys.orig_argv](#) and missing from [sys.argv](#).

Added in version 3.10.

sys.path

A list of strings that specifies the search path for modules. Initialized from the environment variable [PYTHONPATH](#), plus an installation-dependent default.

By default, as initialized upon program startup, a potentially unsafe path is prepended to [sys.path](#) (*before* the entries inserted as a result of [PYTHONPATH](#)):

- `python -m` module command line: prepend the current working directory.
- `python script.py` command line: prepend the script's directory. If it's a symbolic link, resolve symbolic links.
- `python -c` code and `python` (REPL) command lines: prepend an empty string, which means the current working directory.

To not prepend this potentially unsafe path, use the `-P` command line option or the `PYTHONSAFEPEPATH` environment variable.

A program is free to modify this list for its own purposes. Only strings should be added to `sys.path`; all other data types are ignored during import.

See also:

- Module `site` This describes how to use .pth files to extend `sys.path`.

sys.path_hooks

A list of callables that take a path argument to try to create a `finder` for the path. If a finder can be created, it is to be returned by the callable, else raise `ImportError`.

Originally specified in [PEP 302](#).

sys.path_importer_cache

A dictionary acting as a cache for `finder` objects. The keys are paths that have been passed to `sys.path_hooks` and the values are the finders that are found. If a path is a valid file system path but no finder is found on `sys.path_hooks` then `None` is stored.

Originally specified in [PEP 302](#).

sys.platform

A string containing a platform identifier. Known values are:

System	platform value
AIX	'aix'
Android	'android'
Emscripten	'emscripten'
FreeBSD	'freebsd'
iOS	'ios'
Linux	'linux'
macOS	'darwin'
Windows	'win32'
Windows/Cygwin	'cygwin'
WASI	'wasi'

On Unix systems not listed in the table, the value is the lowercased OS name as returned by `uname -s`, with the first part of the version as returned by `uname -r` appended, e.g. 'sunos5', at the time when *Python was built*. Unless you want to test for a specific system version, it is therefore recommended to use the following idiom:

```
if sys.platform.startswith('sunos'):
    # SunOS-specific code here...
```

Changed in version 3.3: On Linux, `sys.platform` doesn't contain the major version anymore. It is always 'linux', instead of 'linux2' or 'linux3'.

Changed in version 3.8: On AIX, `sys.platform` doesn't contain the major version anymore. It is always 'aix', instead of 'aix5' or 'aix7'.

Changed in version 3.13: On Android, `sys.platform` now returns 'android' rather than 'linux'.

Changed in version 3.14: On FreeBSD, `sys.platform` doesn't contain the major version anymore. It is always 'freebsd', instead of 'freebsd13' or 'freebsd14'.

See also: `os.name` has a coarser granularity. `os.uname()` gives system-dependent version information.

The `platform` module provides detailed checks for the system's identity.

sys.platlibdir

Name of the platform-specific library directory. It is used to build the path of standard library and the paths of installed extension modules.

It is equal to "lib" on most platforms. On Fedora and SuSE, it is equal to "lib64" on 64-bit platforms which gives the following `sys.path` paths (where X.Y is the Python `major.minor` version):

- `/usr/lib64/pythonX.Y/`: Standard library (like `os.py` of the `os` module)
- `/usr/lib64/pythonX.Y/lib-dynload/`: C extension modules of the standard library (like the `errno` module, the exact filename is platform specific)
- `/usr/lib/pythonX.Y/site-packages/` (always use `lib`, not `sys.platlibdir`): Third-party modules
- `/usr/lib64/pythonX.Y/site-packages/`: C extension modules of third-party packages

Added in version 3.9.

sys.prefix

A string giving the site-specific directory prefix where the platform independent Python files are installed; on Unix, the default is `/usr/local`. This can be set at build time with the `--prefix` argument to the `configure` script. See [Installation paths](#) for derived paths.

Note: If a [virtual environment](#) is in effect, this `prefix` will point to the virtual environment. The value for the Python installation will still be available, via `base_prefix`. Refer to [Virtual Environments](#) for more information.

Changed in version 3.14: When running under a [virtual environment](#), `prefix` and `exec_prefix` are now set to the virtual environment prefix by the [path initialization](#), instead of `site`. This means that `prefix` and `exec_prefix` always point to the virtual environment, even when `site` is disabled (`-S`).

sys.ps1

sys.ps2

Strings specifying the primary and secondary prompt of the interpreter. These are only defined if the interpreter is in interactive mode. Their initial values in this case are '`>>>`' and '`...` '. If a non-string object is assigned to either variable, its `str()` is re-evaluated each time the interpreter prepares to read a new interactive command; this can be used to implement a dynamic prompt.

sys.setdlopenflags(*n*)

Set the flags used by the interpreter for `dlopen()` calls, such as when the interpreter loads extension modules. Among other things, this will enable a lazy resolving of symbols when importing a module, if called as `sys.setdlopenflags(0)`. To share symbols across extension modules, call as `sys.setdlopenflags(os.RTLD_GLOBAL)`. Symbolic names for the flag values can be found in the `os` module (`RTLD_xxx` constants, e.g. `os.RTLD_LAZY`).

Availability: Unix.

sys.set_int_max_str_digits(*maxdigits*)

Set the integer string conversion length limitation used by this interpreter. See also `get_int_max_str_digits()`.

Added in version 3.11.

sys.setprofile(*profilefunc*)

Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter [The Python Profilers](#) for more information on the Python profiler. The system's profile function is called similarly to the system's trace function (see `settrace()`), but it is called with different events, for example it isn't called for each executed line of code (only on call and return, but the return event is reported even when an exception has been set). The function is thread-specific, but there is no way for the profiler to know about context switches between threads, so it does not make sense to use this in the presence of multiple threads. Also, its return value is not used, so it can simply return `None`. Error in the profile function will cause itself unset.

Note: The same tracing mechanism is used for `setprofile()` as `settrace()`. To trace calls with `setprofile()` inside a tracing function (e.g. in a debugger breakpoint), see `call_tracing()`.

Profile functions should have three arguments: `frame`, `event`, and `arg`. `frame` is the current stack frame. `event` is a string: '`call`', '`return`', '`c_call`', '`c_return`', or '`c_exception`'. `arg` depends on the event type.

The events have the following meaning:

`'call'`

A function is called (or some other code block entered). The profile function is called; `arg` is `None`.

`'return'`

A function (or other code block) is about to return. The profile function is called; `arg` is the value that will be returned, or `None` if the event is caused by an exception being raised.

`'c_call'`

A C function is about to be called. This may be an extension function or a built-in. `arg` is the C function object.

`'c_return'`

A C function has returned. `arg` is the C function object.

`'c_exception'`

A C function has raised an exception. *arg* is the C function object.

Raises an [auditing_event](#) `sys.setprofile` with no arguments.

`sys.setrecursionlimit(limit)`

Set the maximum depth of the Python interpreter stack to *limit*. This limit prevents infinite recursion from causing an overflow of the C stack and crashing Python.

The highest possible limit is platform-dependent. A user may need to set the limit higher when they have a program that requires deep recursion and a platform that supports a higher limit. This should be done with care, because a too-high limit can lead to a crash.

If the new limit is too low at the current recursion depth, a [RecursionError](#) exception is raised.

Changed in version 3.5.1: A [RecursionError](#) exception is now raised if the new limit is too low at the current recursion depth.

`sys.setswitchinterval(interval)`

Set the interpreter's thread switch interval (in seconds). This floating-point value determines the ideal duration of the "timeslices" allocated to concurrently running Python threads. Please note that the actual value can be higher, especially if long-running internal functions or methods are used. Also, which thread becomes scheduled at the end of the interval is the operating system's decision. The interpreter doesn't have its own scheduler.

Added in version 3.2.

`sys.settrace(tracefunc)`

Set the system's trace function, which allows you to implement a Python source code debugger in Python. The function is thread-specific; for a debugger to support multiple threads, it must register a trace function using `settrace()` for each thread being debugged or use `threading.settrace()`.

Trace functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the [current stack frame](#). *event* is a string: 'call', 'line', 'return', 'exception' or 'opcode'. *arg* depends on the event type.

The trace function is invoked (with *event* set to 'call') whenever a new local scope is entered; it should return a reference to a local trace function to be used for the new scope, or `None` if the scope shouldn't be traced.

The local trace function should return a reference to itself, or to another function which would then be used as the local trace function for the scope.

If there is any error occurred in the trace function, it will be unset, just like `settrace(None)` is called.

Note: Tracing is disabled while calling the trace function (e.g. a function set by `settrace()`). For recursive tracing see [call_tracing\(\)](#).

The events have the following meaning:

'call'

A function is called (or some other code block entered). The global trace function is called; *arg* is `None`; the return value specifies the local trace function.

'line'

The interpreter is about to execute a new line of code or re-execute the condition of a loop. The local trace function is called; *arg* is `None`; the return value specifies the new local trace function. See `Objects/lnotab_notes.txt` for a detailed explanation of how this works. Per-line events may be disabled for a frame by setting `f_trace_lines` to `False` on that `frame`.

'return'

A function (or other code block) is about to return. The local trace function is called; *arg* is the value that will be returned, or `None` if the event is caused by an exception being raised. The trace function's return value is ignored.

'exception'

An exception has occurred. The local trace function is called; *arg* is a tuple (`exception`, `value`, `traceback`); the return value specifies the new local trace function.

'opcode'

The interpreter is about to execute a new opcode (see `dis` for opcode details). The local trace function is called; *arg* is `None`; the return value specifies the new local trace function. Per-opcode events are not emitted by default: they must be explicitly requested by setting `f_trace_opcodes` to `True` on the `frame`.

Note that as an exception is propagated down the chain of callers, an 'exception' event is generated at each level.

For more fine-grained usage, it's possible to set a trace function by assigning `frame.f_trace = tracefunc` explicitly, rather than relying on it being set indirectly via the return value from an already installed trace function. This is also required for activating the trace function on the current frame, which `settrace()` doesn't do. Note that in order for this to work, a global tracing function must have been installed with `settrace()` in order to enable the runtime tracing machinery, but it doesn't need to be the same tracing function (e.g. it could be a low overhead tracing function that simply returns `None` to disable itself immediately on each frame).

For more information on code and frame objects, refer to [The standard type hierarchy](#).

Raises an [auditing event](#) `sys.settrace` with no arguments.

CPython implementation detail: The `settrace()` function is intended only for implementing debuggers, profilers, coverage tools and the like. Its behavior is part of the implementation platform, rather than part of the language definition, and thus may not be available in all Python implementations.

Changed in version 3.7: 'opcode' event type added; `f_trace_lines` and `f_trace_opcodes` attributes added to frames

`sys.set_asyncgen_hooks([firstiter] [, finalizer])`

Accepts two optional keyword arguments which are callables that accept an [asynchronous generator iterator](#) as an argument. The `firstiter` callable will be called when an asynchronous generator is iterated for the first time. The `finalizer` will be called when an asynchronous generator is about to be garbage collected.

Raises an [auditing event](#) `sys.set_asyncgen_hooks_firstiter` with no arguments.

Raises an [auditing event](#) `sys.set_asasyncgen_hooks_finalizer` with no arguments.

Two auditing events are raised because the underlying API consists of two calls, each of which must raise its own event.

Added in version 3.6: See [PEP 525](#) for more details, and for a reference example of a *finalizer* method see the implementation of `asyncio.Loop.shutdown_asyncgens` in [Lib/asyncio/base_events.py](#).

Note: This function has been added on a provisional basis (see [PEP 411](#) for details.)

`sys.set_coroutine_origin_tracking_depth(depth)`

Allows enabling or disabling coroutine origin tracking. When enabled, the `cr_origin` attribute on coroutine objects will contain a tuple of (filename, line number, function name) tuples describing the traceback where the coroutine object was created, with the most recent call first. When disabled, `cr_origin` will be `None`.

To enable, pass a *depth* value greater than zero; this sets the number of frames whose information will be captured. To disable, set *depth* to zero.

This setting is thread-specific.

Added in version 3.7.

Note: This function has been added on a provisional basis (see [PEP 411](#) for details.) Use it only for debugging purposes.

`sys.activate_stack_trampoline(backend, ...)`

Activate the stack profiler trampoline *backend*. The only supported backend is "perf".

Stack trampolines cannot be activated if the JIT is active.

Availability: Linux.

Added in version 3.12.

See also:

- [Python support for the Linux perf profiler](#)
- <https://perf.wiki.kernel.org>

`sys.deactivate_stack_trampoline()`

Deactivate the current stack profiler trampoline backend.

If no stack profiler is activated, this function has no effect.

Availability: Linux.

Added in version 3.12.

`sys.is_stack_trampoline_active()`

Return True if a stack profiler trampoline is active.

Availability: Linux.

Added in version 3.12.

`sys.remote_exec(pid, script)`

Executes *script*, a file containing Python code in the remote process with the given *pid*.

This function returns immediately, and the code will be executed by the target process's main thread at the next available opportunity, similarly to how signals are handled. There is no interface to determine when the code has been executed. The caller is responsible for making sure that the file still exists whenever the remote process tries to read it and that it hasn't been overwritten.

The remote process must be running a CPython interpreter of the same major and minor version as the local process. If either the local or remote interpreter is pre-release (alpha, beta, or release candidate) then the local and remote interpreters must be the same exact version.

When the code is executed in the remote process, an auditing event `sys.remote_exec` is raised with the *pid* and the path to the script file. This event is raised in the process that called `sys.remote_exec()`.

When the script is executed in the remote process, an auditing event `cpython.remote_debugger_script` is raised with the path in the remote process. This event is raised in the remote process, not the one that called `sys.remote_exec()`.

Availability: Unix, Windows.

Added in version 3.14.

`sys._enablelegacywindowsfsencoding()`

Changes the filesystem encoding and error handler to 'mbcs' and 'replace' respectively, for consistency with versions of Python prior to 3.6.

This is equivalent to defining the `PYTHONLEGACYWINDOWSFSENCODING` environment variable before launching Python.

See also `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()`.

Availability: Windows.

Note: Changing the filesystem encoding after Python startup is risky because the old fsencoding or paths encoded by the old fsencoding may be cached somewhere. Use `PYTHONLEGACYWINDOWSFSENCODING` instead.

Added in version 3.6: See [PEP 529](#) for more details.

Deprecated since version 3.13, will be removed in version 3.16: Use `PYTHONLEGACYWINDOWSFSENCODING` instead.

`sys.stdin` `sys.stdout` `sys.stderr`

File objects used by the interpreter for standard input, output and errors:

- `stdin` is used for all interactive input (including calls to `input()`);
- `stdout` is used for the output of `print()` and `expression` statements and for the prompts of `input()`;
- The interpreter's own prompts and its error messages go to `stderr`.

These streams are regular `text files` like those returned by the `open()` function. Their parameters are chosen as follows:

- The encoding and error handling are initialized from `PyConfig.stdio_encoding` and `PyConfig.stdio_errors`.

On Windows, UTF-8 is used for the console device. Non-character devices such as disk files and pipes use the system locale encoding (i.e. the ANSI codepage). Non-console character devices such as NUL (i.e. where `isatty()` returns `True`) use the value of the console input and output codepages at startup, respectively for `stdin` and `stdout/stderr`. This defaults to the system `locale encoding` if the process is not initially attached to a console.

The special behaviour of the console can be overridden by setting the environment variable `PYTHONLEGACYWINDOWSSTUDIO` before starting Python. In that case, the console codepages are used as for any other character device.

Under all platforms, you can override the character encoding by setting the `PYTHONIOENCODING` environment variable before starting Python or by using the new `-X utf8` command line option and `PYTHONUTF8` environment variable. However, for the Windows console, this only applies when `PYTHONLEGACYWINDOWSSTUDIO` is also set.

- When interactive, the `stdout` stream is line-buffered. Otherwise, it is block-buffered like regular text files. The `stderr` stream is line-buffered in both cases. You can make both streams unbuffered by passing the `-u` command-line option or setting the `PYTHONUNBUFFERED` environment variable.

Changed in version 3.9: Non-interactive `stderr` is now line-buffered instead of fully buffered.

Note: To write or read binary data from/to the standard streams, use the underlying binary `buffer` object. For example, to write bytes to `stdout`, use `sys.stdout.buffer.write(b'abc')`.

However, if you are writing a library (and do not control in which context its code will be executed), be aware that the standard streams may be replaced with file-like objects like `io.StringIO` which do not support the `buffer` attribute.

`sys.__stdin__`
`sys.__stdout__`
`sys.__stderr__`

These objects contain the original values of `stdin`, `stderr` and `stdout` at the start of the program. They are used during finalization, and could be useful to print to the actual standard stream no matter if the `sys.std*` object has been redirected.

It can also be used to restore the actual files to known working file objects in case they have been overwritten with a broken object. However, the preferred way to do this is to explicitly save the previous stream before replacing it, and restore the saved object.

Note: Under some conditions `stdin`, `stdout` and `stderr` as well as the original values `__stdin__`, `__stdout__` and `__stderr__` can be `None`. It is usually the case for Windows GUI apps that aren't connected to a console and Python apps started with `pythonw`.

sys.`stdlib_module_names`

A frozenset of strings containing the names of standard library modules.

It is the same on all platforms. Modules which are not available on some platforms and modules disabled at Python build are also listed. All module kinds are listed: pure Python, built-in, frozen and extension modules. Test modules are excluded.

For packages, only the main package is listed: sub-packages and sub-modules are not listed. For example, the `email` package is listed, but the `email.mime` sub-package and the `email.message` sub-module are not listed.

See also the [sys.`builtin_module_names`](#) list.

Added in version 3.10.

sys.`thread_info`

A [named tuple](#) holding information about the thread implementation.

thread_info.`name`

The name of the thread implementation:

- "nt": Windows threads
- "pthread": POSIX threads
- "pthread-stubs": stub POSIX threads (on WebAssembly platforms without threading support)
- "solaris": Solaris threads

thread_info.`lock`

The name of the lock implementation:

- "semaphore": a lock uses a semaphore
- "mutex+cond": a lock uses a mutex and a condition variable
- None if this information is unknown

thread_info.`version`

The name and version of the thread library. It is a string, or `None` if this information is unknown.

Added in version 3.3.

sys.`tracebacklimit`

When this variable is set to an integer value, it determines the maximum number of levels of traceback information printed when an unhandled exception occurs. The default is `1000`. When set to `0` or less, all traceback information is suppressed and only the exception type and value are printed.

sys.`unraisablehook(unraisable, ...)`

Handle an unraisable exception.

Called when an exception has occurred but there is no way for Python to handle it. For example, when a destructor raises an exception or during garbage collection (`gc.collect()`).

The `unraisable` argument has the following attributes:

- `exc_type`: Exception type.
- `exc_value`: Exception value, can be `None`.
- `exc_traceback`: Exception traceback, can be `None`.
- `err_msg`: Error message, can be `None`.
- `object`: Object causing the exception, can be `None`.

The default hook formats `err_msg` and `object` as: `f'{err_msg}: {object!r}'`; use "Exception ignored in" error message if `err_msg` is `None`.

`sys.unraisablehook()` can be overridden to control how unraisable exceptions are handled.

See also: `excepthook()` which handles uncaught exceptions.

Warning: Storing `exc_value` using a custom hook can create a reference cycle. It should be cleared explicitly to break the reference cycle when the exception is no longer needed.

Storing `object` using a custom hook can resurrect it if it is set to an object which is being finalized. Avoid storing `object` after the custom hook completes to avoid resurrecting objects.

Raise an auditing event `sys.unraisablehook` with arguments `hook`, `unraisable` when an exception that cannot be handled occurs. The `unraisable` object is the same as what will be passed to the hook. If no hook has been set, `hook` may be `None`.

Added in version 3.8.

sys.version

A string containing the version number of the Python interpreter plus additional information on the build number and compiler used. This string is displayed when the interactive interpreter is started. Do not extract version information out of it, rather, use `version_info` and the functions provided by the `platform` module.

sys.api_version

The C API version, equivalent to the C macro `PYTHON_API_VERSION`. Defined for backwards compatibility.

Currently, this constant is not updated in new Python versions, and is not useful for versioning. This may change in the future.

sys.version_info

A tuple containing the five components of the version number: `major`, `minor`, `micro`, `releaselevel`, and `serial`. All values except `releaselevel` are integers; the release level is `'alpha'`, `'beta'`, `'candidate'`, or `'final'`. The `version_info` value corresponding to the Python version 2.0 is `(2, 0, 0, 'final', 0)`. The components can also be accessed by name, so `sys.version_info[0]` is equivalent to `sys.version_info.major` and so on.

Changed in version 3.1: Added named component attributes.

sys.warnoptions

This is an implementation detail of the warnings framework; do not modify this value. Refer to the [warnings](#) module for more information on the warnings framework.

sys.winver

The version number used to form registry keys on Windows platforms. This is stored as string resource 1000 in the Python DLL. The value is normally the major and minor versions of the running Python interpreter. It is provided in the [sys](#) module for informational purposes; modifying this value has no effect on the registry keys used by Python.

Availability: Windows.

sys.monitoring

Namespace containing functions and constants for register callbacks and controlling monitoring events.

See [sys.monitoring](#) for details.

sys._xoptions

A dictionary of the various implementation-specific flags passed through the `-X` command-line option.

Option names are either mapped to their values, if given explicitly, or to `True`. Example:

```
$ ./python -Xa=b -Xc
Python 3.2a3+ (py3k, Oct 16 2010, 20:14:50)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys._xoptions
{'a': 'b', 'c': True}
```

CPython implementation detail: This is a CPython-specific way of accessing options passed through `-X`. Other implementations may export them through other means, or not at all.

Added in version 3.2.

Citations

- [C99] ISO/IEC 9899:1999. "Programming languages – C." A public draft of this standard is available at <https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf>.