

3.1 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.

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` has zero length.

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. New in version 2.0.

builtin_module_names

A tuple of strings giving 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.)

copyright

A string containing the copyright pertaining to the Python interpreter.

dllhandle

Integer specifying the handle of the Python DLL. Availability: Windows.

displayhook(value)

If *value* is not `None`, this function prints it to `sys.stdout`, and saves it in `__builtin__.__`.

`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`.

excepthook(type, value, traceback)

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

When an exception 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`.

__displayhook__ **__excepthook__**

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

`exc_info()`

This function returns a tuple of three values that give information about the exception that is currently being handled. The information returned is specific both to the current thread and to the current stack frame. If the current stack frame is not handling an exception, the information is taken from the calling stack frame, or its caller, and so on until a stack frame is found that is handling an exception. Here, "handling an exception" is defined as "executing or having executed an except clause." For any stack frame, only information about the most recently handled exception is accessible.

If no exception is being handled anywhere on the stack, a tuple containing three `None` values is returned. Otherwise, the values returned are `(type, value, traceback)`. Their meaning is: *type* gets the exception type of the exception being handled (a class object); *value* gets the exception parameter (its *associated value* or the second argument to `raise`, which is always a class instance if the exception type is a class object); *traceback* gets a traceback object (see the Reference Manual) which encapsulates the call stack at the point where the exception originally occurred.

If `exc_clear()` is called, this function will return three `None` values until either another exception is raised in the current thread or the execution stack returns to a frame where another exception is being handled.

Warning: Assigning the *traceback* return value to a local variable in a function that is handling an exception will cause a circular reference. This will prevent anything referenced by a local variable in the same function or by the traceback from being garbage collected. Since most functions don't need access to the traceback, the best solution is to use something like `exc_type, value = sys.exc_info()[: 2]` to extract only the exception type and value. If you do need the traceback, make sure to delete it after use (best done with a `try ... finally` statement) or to call `exc_info()` in a function that does not itself handle an exception. **Note:** Beginning with Python 2.2, such cycles are automatically reclaimed when garbage collection is enabled and they become unreachable, but it remains more efficient to avoid creating cycles.

`exc_clear()`

This function clears all information relating to the current or last exception that occurred in the current thread. After calling this function, `exc_info()` will return three `None` values until another exception is raised in the current thread or the execution stack returns to a frame where another exception is being handled.

This function is only needed in only a few obscure situations. These include logging and error handling systems that report information on the last or current exception. This function can also be used to try to free resources and trigger object finalization, though no guarantee is made as to what objects will be freed, if any. New in version 2.3.

`exc_type`

`exc_value`

`exc_traceback`

Deprecated since release 1.5. Use `exc_info()` instead.

Since they are global variables, they are not specific to the current thread, so their use is not safe in a multi-threaded program. When no exception is being handled, `exc_type` is set to `None` and the other two are undefined.

`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/pythonversion/config'`, and shared library modules are installed in `exec_prefix + '/lib/pythonversion/lib-dynload'`, where *version* is equal to `version[:3]`.

executable

A string giving the name of the executable binary for the Python interpreter, on systems where this makes sense.

exit([arg])

Exit from Python. This is implemented by raising the `SystemExit` exception, so cleanup actions specified by finally clauses of `try` statements are honored, and it is possible to intercept the exit attempt at an outer level. 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 produce 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 `sys.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.

exitfunc

This value is not actually defined by the module, but can be set by the user (or by a program) to specify a clean-up action at program exit. When set, it should be a parameterless function. This function will be called when the interpreter exits. Only one function may be installed in this way; to allow multiple functions which will be called at termination, use the [atexit](#) module. **Note:** The exit function is not called when the program is killed by a signal, when a Python fatal internal error is detected, or when `os._exit()` is called.

Deprecated since release 2.4. Use [atexit](#) instead.

getcheckinterval()

Return the interpreter's "check interval"; see `setcheckinterval()`. New in version 2.3.

getdefaultencoding()

Return the name of the current default string encoding used by the Unicode implementation. New in version 2.0.

getdlopenflags()

Return the current value of the flags that are used for `dlopen()` calls. The flag constants are defined in the [dl](#) and `DLFCN` modules. Availability: UNIX. New in version 2.2.

getfilesystemencoding()

Return the name of the encoding used to convert Unicode filenames into system file names, or `None` if the system default encoding is used. The result value depends on the operating system:

- On Windows 9x, the encoding is "mbcs".
- On Mac OS X, the encoding is "utf-8".
- On Unix, the encoding is the user's preference according to the result of `nl_langinfo(CODESET)`, or `None` if the `nl_langinfo(CODESET)` failed.
- On Windows NT+, file names are Unicode natively, so no conversion is performed. `getfilesystemencoding` still returns "mbcs", as this is the encoding that applications should use when they explicitly want to convert Unicode strings to byte strings that are equivalent when used as file names.

New in version 2.3.

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()`.

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()`.

_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.

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

getwindowsversion()

Return a tuple containing five components, describing the Windows version currently running. The elements are *major*, *minor*, *build*, *platform*, and *text*. *text* contains a string while all other values are integers.

platform may be one of the following values:

Constant	Platform
0 (VER_PLATFORM_WIN32S)	Win32s on Windows 3.1
1 (VER_PLATFORM_WIN32_WINDOWS)	Windows 95/98/ME
2 (VER_PLATFORM_WIN32_NT)	Windows NT/2000/XP
3 (VER_PLATFORM_WIN32_CE)	Windows CE

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

Availability: Windows. New in version 2.3.

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 `version_info` value may be used for a more human-friendly encoding of the same information. New in version 1.5.2.

last_type

last_value

last_traceback

These three variables are not always defined; they are set when an exception is not handled and the interpreter prints an error message and a stack traceback. Their 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 chapter 9, "The Python Debugger," for more information.)

The meaning of the variables is the same as that of the return values from `exc_info()` above. (Since there is only one interactive thread, thread-safety is not a concern for these variables, unlike for `exc_type` etc.)

maxint

The largest positive integer supported by Python's regular integer type. This is at least $2^{31}-1$. The largest negative integer is `-maxint-1` -- the asymmetry results from the use of 2's complement binary arithmetic.

maxunicode

An integer giving the largest supported code point for a Unicode character. The value of this depends on the configuration option that specifies whether Unicode characters are stored as UCS-2 or UCS-4.

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. Note that removing a module from this dictionary is *not* the same as calling `reload()` on the corresponding module object.

path

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

As initialized upon program startup, the first item of this list, `path[0]`, is the directory containing the script that was used to invoke the Python interpreter. If the script directory is not available (e.g. if the interpreter is invoked interactively or if the script is read from standard input), `path[0]` is the empty string, which directs Python to search modules in the current directory first. Notice that the script directory is inserted *before* the entries inserted as a result of `PYTHONPATH`.

A program is free to modify this list for its own purposes.

Changed in version 2.3: Unicode strings are no longer ignored.

platform

This string contains a platform identifier, e.g. `'sunos5'` or `'linux1'`. This can be used to append platform-specific components to `path`, for instance.

prefix

A string giving the site-specific directory prefix where the platform independent Python files are installed; by default, this is the string `'/usr/local'`. This can be set at build time with the **--prefix** argument to the **configure** script. The main collection of Python library modules is installed in the directory `prefix + '/lib/pythonversion'` while the platform independent header files (all except `pyconfig.h`) are stored in `prefix + '/include/pythonversion'`, where *version* is equal to `version[:3]`.

ps1

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.

setcheckinterval(*interval*)

Set the interpreter's "check interval". This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The default is 100, meaning the check is performed every 100 Python virtual instructions. Setting it to a larger value may increase performance for programs using threads. Setting it to a value ≤ 0 checks every virtual instruction, maximizing responsiveness as well as overhead.

setdefaultencoding(*name*)

Set the current default string encoding used by the Unicode implementation. If *name* does not match any available encoding, LookupError is raised. This function is only intended to be used by the [site](#) module implementation and, where needed, by sitecustomize. Once used by the [site](#) module, it is removed from the sys module's namespace. New in version 2.0.

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(dl.RTLD_NOW | dl.RTLD_GLOBAL). Symbolic names for the flag modules can be either found in the [dl](#) module, or in the DLFCN module. If DLFCN is not available, it can be generated from /usr/include/dlfcn.h using the **h2py** script. Availability: UNIX. New in version 2.2.

setprofile(*profilefunc*)

Set the system's profile function, which allows you to implement a Python source code profiler in Python. See chapter [10](#) 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 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.

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 she has 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.

settrace(*tracefunc*)

Set the system's trace function, which allows you to implement a Python source code debugger in Python. See section [9.2](#), "How It Works," in the chapter on the Python debugger. The function is thread-specific; for a debugger to support multiple threads, it must be registered using settrace() for each thread being debugged. **Note:** 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.

settsdump(*on_flag*)

Activate dumping of VM measurements using the Pentium timestamp counter, if *on_flag* is true. Deactivate these dumps if *on_flag* is off. The function is available only if Python was compiled with **--with-tsc**. To understand the output of this dump, read Python/ceval.c in the Python sources. New in version 2.4.

stdin
stdout
stderr

File objects corresponding to the interpreter's standard input, output and error streams. `stdin` is used for all interpreter input except for scripts but including calls to `input()` and `raw_input()`. `stdout` is used for the output of `print` and expression statements and for the prompts of `input()` and `raw_input()`. The interpreter's own prompts and (almost all of) its error messages go to `stderr`. `stdout` and `stderr` needn't be built-in file objects: any object is acceptable as long as it has a `write()` method that takes a string argument. (Changing these objects doesn't affect the standard I/O streams of processes executed by `os.popen()`, `os.system()` or the `exec*()` family of functions in the [os](#) module.)

__stdin__
__stdout__
__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 restore the actual files to known working file objects in case they have been overwritten with a broken object.

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.

version

A string containing the version number of the Python interpreter plus additional information on the build number and compiler used. It has a value of the form '*version* (*#build_number*, *build_date*, *build_time*) [*compiler*]' . The first three characters are used to identify the version in the installation directories (where appropriate on each platform). An example:

```
>>> import sys
>>> sys.version
'1.5.2 (#0 Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)]'
```

api_version

The C API version for this interpreter. Programmers may find this useful when debugging version conflicts between Python and extension modules. New in version 2.3.

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). New in version 2.0.

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.

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 first three characters of `version`. 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.

See Also:

Module [site](#):

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

Release 2.4.3, documentation updated on 29 March 2006.

See [About this document...](#) for information on suggesting changes.