28.1. `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.

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.

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.

New in version 2.0.

sys.**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.)

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 some other code.

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.

New in version 2.6.

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.

New in version 2.5.

sys.**dllhandle**

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

sys.**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`.

sys.**dont_write_bytecode**

If this is true, Python won't try to write `.pyc` or `.pyo` 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.

New in version 2.6.

sys.**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`.

sys.**__displayhook__**
sys.**__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.

sys.**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 `exctype, 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.

`sys.`**`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.

`sys.`**`exc_type`**
`sys.`**`exc_value`**
`sys.`**`exc_traceback`**

Deprecated since version 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.

`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 `2.7`.

`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*])

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 `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.

`sys.`**`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 version 2.4: Use `atexit` instead.

`sys.`**`flags`**

The struct sequence *flags* exposes the status of command line flags. The attributes are read only.

attribute	flag
`debug`	`-d`
`py3k_warning`	`-3`

attribute	flag
`division_warning`	`-Q`
`division_new`	`-Qnew`
`inspect`	`-i`
`interactive`	`-i`
`optimize`	`-O` or `-OO`
`dont_write_bytecode`	`-B`
`no_user_site`	`-s`
`no_site`	`-S`
`ignore_environment`	`-E`
`tabcheck`	`-t` or `-tt`
`verbose`	`-v`
`unicode`	`-U`
`bytes_warning`	`-b`
`hash_randomization`	`-R`

New in version 2.6.

New in version 2.7.3: The `hash_randomization` attribute.

sys.**float_info**

A structseq 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.

attribute	float.h macro	explanation
`epsilon`	DBL_EPSILON	difference between 1 and the least value greater than 1 that is representable as a float
`dig`	DBL_DIG	maximum number of decimal digits that can be faithfully represented in a float; see below
`mant_dig`	DBL_MANT_DIG	float precision: the number of base-`radix` digits in the significand of a float
`max`	DBL_MAX	maximum representable finite float
`max_exp`	DBL_MAX_EXP	maximum integer e such that `radix**(e-1)` is a representable finite float
`max_10_exp`	DBL_MAX_10_EXP	maximum integer e such that `10**e` is in the range of representable finite floats
`min`	DBL_MIN	minimum positive normalized float
`min_exp`	DBL_MIN_EXP	minimum integer e such that `radix**(e-1)` is a normalized float

attribute	float.h macro	explanation
min_10_exp	DBL_MIN_10_EXP	minimum integer e such that `10**e` is a normalized float
radix	FLT_RADIX	radix of exponent representation
rounds	FLT_ROUNDS	integer constant representing the rounding mode used for arithmetic operations. This reflects the value of the system FLT_ROUNDS macro at interpreter startup time. See section 5.2.4.2.2 of the C99 standard for an explanation of the possible values and their meanings.

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'
```

New in version 2.6.

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 2.7 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 2.7.

New in version 2.7.

sys.**`getcheckinterval`**()

Return the interpreter's "check interval"; see **`setcheckinterval()`**.

New in version 2.3.

sys.**`getdefaultencoding`**()

Return the name of the current default string encoding used by the Unicode implementation.

New in version 2.0.

sys.**`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.

sys.**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 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.
- On Windows 9x, the encoding is `'mbcs'`.

New in version 2.3.

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()`**.

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.

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.

New in version 2.6.

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.

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.**getprofile**()

Get the profiler function as set by `setprofile()`.

New in version 2.6.

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.

New in version 2.6.

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*, and *product_type*. *service_pack* contains a string while 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 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/x64
`3 (VER_PLATFORM_WIN32_CE)`	Windows CE

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.

Availability: Windows.

New in version 2.3.

Changed in version 2.7: Changed to a named tuple and added *service_pack_minor*, *service_pack_major*, *suite_mask*, and *product_type*.

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

The `hexversion` is a 32-bit number with the following layout:

Bits (big endian order)	Meaning
1-8	`PY_MAJOR_VERSION` (the `2` in `2.1.0a3`)
9-16	`PY_MINOR_VERSION` (the `1` in `2.1.0a3`)
17-24	`PY_MICRO_VERSION` (the `0` in `2.1.0a3`)
25-28	`PY_RELEASE_LEVEL` (`0xA` for alpha, `0xB` for beta, `0xC` for release candidate and `0xF` for final)
29-32	`PY_RELEASE_SERIAL` (the `3` in `2.1.0a3`, zero for final releases)

Thus `2.1.0a3` is hexversion `0x020100a3`.

Newin version 1.5.2.

sys.**long_info**

A struct sequence that holds information about Python's internal representation of integers. The attributes are read only.

Attribute	Explanation
bits_per_digit	number of bits held in each digit. Python integers are stored internally in base `2**long_info.bits_per_digit`
sizeof_digit	size in bytes of the C type used to represent a digit

Newin version 2.7.

sys.**last_type**
sys.**last_value**
sys.**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 pdb — 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.)

sys.**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.

sys.**maxsize**

The largest positive integer supported by the platform's Py_ssize_t type, and thus the maximum size lists, strings, dicts, and many other containers can have.

sys.**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.

sys.**meta_path**

A list of finder objects that have their `find_module()` methods called to see if one of the objects can find the module to be imported. The `find_module()` method is called at least with the absolute name of the module being imported. If the module to be imported is contained in package then the parent package's `__path__` attribute is passed in as a second argument. The method returns `None` if the module cannot be found, else returns a loader.

`sys.meta_path` is searched before any implicit default finders or `sys.path`.

See **PEP 302** for the original specification.

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

sys.**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.

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 explicit finder is found on `sys.path_hooks` then `None` is stored to represent the implicit default finder should be used. If the path is not an existing path then `imp.NullImporter` is set.

Originally specified in **PEP 302**.

sys.**platform**

This string contains a platform identifier that can be used to append platform-specific components to `sys.path`, for instance.

For most Unix systems, this 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('freebsd'):
    # FreeBSD-specific code here...
elif sys.platform.startswith('linux'):
    # Linux-specific code here...
```

Changed in version 2.7.3: Since lots of code check for `sys.platform == 'linux2'`, and there is no essential change between Linux 2.x and 3.x, `sys.platform` is always set to `'linux2'`, even on Linux 3.x. In Python 3.3 and later, the value will always be set to `'linux'`, so it is recommended to always use the `startswith` idiom presented above.

For other systems, the values are:

System	`platform` value
Linux (2.x *and* 3.x)	`'linux2'`
Windows	`'win32'`
Windows/Cygwin	`'cygwin'`
Mac OS X	`'darwin'`
OS/2	`'os2'`
OS/2 EMX	`'os2emx'`
RiscOS	`'riscos'`
AtheOS	`'atheos'`

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.**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/pythonX.Y` while the platform independent header files (all except `pyconfig.h`) are stored in `prefix/include/pythonX.Y`, where *X.Y* is the version number of Python, for example `2.7`.

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.**py3kwarning**

Bool containing the status of the Python 3 warning flag. It's `True` when Python is started with the -3 option. (This should be considered read-only; setting it to a different value doesn't have an effect on Python 3 warnings.)

Newin version 2.6.

sys.**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.

sys.**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.

Newin version 2.0.

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(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.

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 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`.

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 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.

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 be registered using `settrace()` for each thread being debugged.

Trace functions should have three arguments: *frame*, *event*, and *arg*. *frame* is the current stack frame. *event* is a string: `'call'`, `'line'`, `'return'`, `'exception'`, `'c_call'`, `'c_return'`, or `'c_exception'`. *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 that scope, or `None` if the scope shouldn't be traced.

The local trace function should return a reference to itself (or to another function for further tracing in that scope), or `None` to turn off tracing in that scope.

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.

`'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.

`'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.

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

For more information on code and frame objects, refer to The standard type hierarchy.

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.

sys.**settscdump**(*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.

CPython implementation detail: This function is intimately bound to CPython implementation details and thus not likely to be implemented elsewhere.

sys.**stdin**
sys.**stdout**
sys.**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.)

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.

sys.**`subversion`**

A triple (repo, branch, version) representing the Subversion information of the Python interpreter. *repo* is the name of the repository, `'CPython'`. *branch* is a string of one of the forms `'trunk'`, `'branches/name'` or `'tags/name'`. *version* is the output of `svnversion`, if the interpreter was built from a Subversion checkout; it contains the revision number (range) and possibly a trailing 'M' if there were local modifications. If the tree was exported (or svnversion was not available), it is the revision of `Include/patchlevel.h` if the branch is a tag. Otherwise, it is `None`.

New in version 2.5.

Note: Python is now developed using Mercurial. In recent Python 2.7 bugfix releases, **`subversion`** therefore contains placeholder information. It is removed in Python 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.**`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 for this interpreter. Programmers may find this useful when debugging version conflicts between Python and extension modules.

New in version 2.3.

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.

New in version 2.0.

Changed in version 2.7: 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 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.

Citations

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