os — Miscellaneous operating system interfaces

Source code: Lib/os.py

This module provides a portable way of using operating system dependent functionality. If you just want to read or write a file see `open()`, if you want to manipulate paths, see the `os.path` module, and if you want to read all the lines in all the files on the command line see the `fileinput` module. For creating temporary files and directories see the `tempfile` module, and for high-level file and directory handling see the `shutil` module.

Notes on the availability of these functions:

- The design of all built-in operating system dependent modules of Python is such that as long as the same functionality is available, it uses the same interface; for example, the function `os.stat(path)` returns stat information about *path* in the same format (which happens to have originated with the POSIX interface).
- Extensions peculiar to a particular operating system are also available through the `os` module, but using them is of course a threat to portability.
- All functions accepting path or file names accept both bytes and string objects, and result in an object of the same type, if a path or file name is returned.
- On VxWorks, os.popen, os.fork, os.execv and os.spawn*p* are not supported.
- On WebAssembly platforms, Android and iOS, large parts of the `os` module are not available or behave differently. APIs related to processes (e.g. `fork()`, `execve()`) and resources (e.g. `nice()`) are not available. Others like `getuid()` and `getpid()` are emulated or stubs. WebAssembly platforms also lack support for signals (e.g. `kill()`, `wait()`).

Note: All functions in this module raise `OSError` (or subclasses thereof) in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type, but are not accepted by the operating system.

exception os.**error**

An alias for the built-in `OSError` exception.

os.**name**

The name of the operating system dependent module imported. The following names have currently been registered: `'posix'`, `'nt'`, `'java'`.

See also: `sys.platform` has a finer granularity. `os.uname()` gives system-dependent version information.

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

File Names, Command Line Arguments, and Environment Variables

In Python, file names, command line arguments, and environment variables are represented using the string type. On some systems, decoding these strings to and from bytes is necessary before passing them to the operating system. Python uses the filesystem encoding and error handler to perform this conversion (see `sys.getfilesystemencoding()`).

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 3.1:* On some systems, conversion using the file system encoding may fail. In this case, Python uses the surrogateescape encoding error handler, which means that undecodable bytes are replaced by a Unicode character U+DC*xx* on decoding, and these are again translated to the original byte on encoding.

The file system encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise `UnicodeError`.

See also the locale encoding.

Python UTF-8 Mode

> *Added in version 3.7:* See **PEP 540** for more details.

The Python UTF-8 Mode ignores the locale encoding and forces the usage of the UTF-8 encoding:

- Use UTF-8 as the filesystem encoding.
- `sys.getfilesystemencoding()` returns `'utf-8'`.
- `locale.getpreferredencoding()` returns `'utf-8'` (the *do_setlocale* argument has no effect).
- `sys.stdin`, `sys.stdout`, and `sys.stderr` all use UTF-8 as their text encoding, with the `surrogateescape` error handler being enabled for `sys.stdin` and `sys.stdout` (`sys.stderr` continues to use `backslashreplace` as it does in the default locale-aware mode)
- On Unix, `os.device_encoding()` returns `'utf-8'` rather than the device encoding.

Note that the standard stream settings in UTF-8 mode can be overridden by `PYTHONIOENCODING` (just as they can be in the default locale-aware mode).

As a consequence of the changes in those lower level APIs, other higher level APIs also exhibit different default behaviours:

- Command line arguments, environment variables and filenames are decoded to text using the UTF-8 encoding.
- `os.fsdecode()` and `os.fsencode()` use the UTF-8 encoding.
- `open()`, `io.open()`, and `codecs.open()` use the UTF-8 encoding by default. However, they still use the strict error handler by default so that attempting to open a binary file in text mode is likely to raise an exception rather than producing nonsense data.

The Python UTF-8 Mode is enabled if the LC_CTYPE locale is `C` or `POSIX` at Python startup (see the `PyConfig_Read()` function).

It can be enabled or disabled using the `-X utf8` command line option and the `PYTHONUTF8` environment variable.

If the PYTHONUTF8 environment variable is not set at all, then the interpreter defaults to using the current locale settings, *unless* the current locale is identified as a legacy ASCII-based locale (as described for PYTHONCOERCECLOCALE), and locale coercion is either disabled or fails. In such legacy locales, the interpreter will default to enabling UTF-8 mode unless explicitly instructed not to do so.

The Python UTF-8 Mode can only be enabled at the Python startup. Its value can be read from sys.flags.utf8_mode.

See also the UTF-8 mode on Windows and the filesystem encoding and error handler.

See also:

PEP 686
Python 3.15 will make Python UTF-8 Mode default.

Process Parameters

These functions and data items provide information and operate on the current process and user.

os.**ctermid**()
Return the filename corresponding to the controlling terminal of the process.

Availability: Unix, not WASI.

os.**environ**
A mapping object where keys and values are strings that represent the process environment. For example, environ['HOME'] is the pathname of your home directory (on some platforms), and is equivalent to getenv("HOME") in C.

This mapping is captured the first time the os module is imported, typically during Python startup as part of processing site.py. Changes to the environment made after this time are not reflected in os.environ, except for changes made by modifying os.environ directly.

This mapping may be used to modify the environment as well as query the environment. putenv() will be called automatically when the mapping is modified.

On Unix, keys and values use sys.getfilesystemencoding() and 'surrogateescape' error handler. Use environb if you would like to use a different encoding.

On Windows, the keys are converted to uppercase. This also applies when getting, setting, or deleting an item. For example, environ['monty'] = 'python' maps the key 'MONTY' to the value 'python'.

Note: Calling putenv() directly does not change os.environ, so it's better to modify os.environ.

Note: On some platforms, including FreeBSD and macOS, setting environ may cause memory leaks. Refer to the system documentation for putenv().

You can delete items in this mapping to unset environment variables. unsetenv() will be called automatically when an item is deleted from os.environ, and when one of the pop() or clear() methods is called.

See also: The `os.reload_environ()` function.

Changed in version 3.9: Updated to support **PEP 584**'s merge (`|`) and update (`|=`) operators.

`os.environb`

Bytes version of `environ`: a mapping object where both keys and values are `bytes` objects representing the process environment. `environ` and `environb` are synchronized (modifying `environb` updates `environ`, and vice versa).

`environb` is only available if `supports_bytes_environ` is `True`.

Added in version 3.2.

Changed in version 3.9: Updated to support **PEP 584**'s merge (`|`) and update (`|=`) operators.

`os.reload_environ()`

The `os.environ` and `os.environb` mappings are a cache of environment variables at the time that Python started. As such, changes to the current process environment are not reflected if made outside Python, or by `os.putenv()` or `os.unsetenv()`. Use `os.reload_environ()` to update `os.environ` and `os.environb` with any such changes to the current process environment.

Warning: This function is not thread-safe. Calling it while the environment is being modified in an other thread is an undefined behavior. Reading from `os.environ` or `os.environb`, or calling `os.getenv()` while reloading, may return an empty result.

Added in version 3.14.

`os.chdir(path)`
`os.fchdir(fd)`
`os.getcwd()`

These functions are described in Files and Directories.

`os.fsencode(filename)`

Encode path-like *filename* to the filesystem encoding and error handler; return `bytes` unchanged.

`fsdecode()` is the reverse function.

Added in version 3.2.

Changed in version 3.6: Support added to accept objects implementing the `os.PathLike` interface.

`os.fsdecode(filename)`

Decode the path-like *filename* from the filesystem encoding and error handler; return `str` unchanged.

`fsencode()` is the reverse function.

Added in version 3.2.

Changed in version 3.6: Support added to accept objects implementing the `os.PathLike` interface.

`os.fspath(path)`

Return the file system representation of the path.

If `str` or `bytes` is passed in, it is returned unchanged. Otherwise `__fspath__()` is called and its value is returned as long as it is a `str` or `bytes` object. In all other cases, `TypeError` is raised.

> *Added in version 3.6.*

class os.**PathLike**

An abstract base class for objects representing a file system path, e.g. `pathlib.PurePath`.

> *Added in version 3.6.*

abstractmethod **__fspath__**()

Return the file system path representation of the object.

The method should only return a `str` or `bytes` object, with the preference being for `str`.

os.**getenv**(*key*, *default=None*)

Return the value of the environment variable *key* as a string if it exists, or *default* if it doesn't. *key* is a string. Note that since `getenv()` uses `os.environ`, the mapping of `getenv()` is similarly also captured on import, and the function may not reflect future environment changes.

On Unix, keys and values are decoded with `sys.getfilesystemencoding()` and `'surrogateescape'` error handler. Use `os.getenvb()` if you would like to use a different encoding.

Availability: Unix, Windows.

os.**getenvb**(*key*, *default=None*)

Return the value of the environment variable *key* as bytes if it exists, or *default* if it doesn't. *key* must be bytes. Note that since `getenvb()` uses `os.environb`, the mapping of `getenvb()` is similarly also captured on import, and the function may not reflect future environment changes.

`getenvb()` is only available if `supports_bytes_environ` is `True`.

Availability: Unix.

> *Added in version 3.2.*

os.**get_exec_path**(*env=None*)

Returns the list of directories that will be searched for a named executable, similar to a shell, when launching a process. *env*, when specified, should be an environment variable dictionary to lookup the PATH in. By default, when *env* is `None`, `environ` is used.

> *Added in version 3.2.*

os.**getegid**()

Return the effective group id of the current process. This corresponds to the "set id" bit on the file being executed in the current process.

Availability: Unix, not WASI.

os.**geteuid**()

Return the current process's effective user id.

Availability: Unix, not WASI.

os.**getgid**()

Return the real group id of the current process.

Availability: Unix.

The function is a stub on WASI, see WebAssembly platforms for more information.

os.**getgrouplist**(*user*, *group*, /)

Return list of group ids that *user* belongs to. If *group* is not in the list, it is included; typically, *group* is specified as the group ID field from the password record for *user*, because that group ID will otherwise be potentially omitted.

Availability: Unix, not WASI.

> *Added in version 3.3.*

os.**getgroups**()

Return list of supplemental group ids associated with the current process.

Availability: Unix, not WASI.

> **Note:** On macOS, `getgroups()` behavior differs somewhat from other Unix platforms. If the Python interpreter was built with a deployment target of `10.5` or earlier, `getgroups()` returns the list of effective group ids associated with the current user process; this list is limited to a system-defined number of entries, typically 16, and may be modified by calls to `setgroups()` if suitably privileged. If built with a deployment target greater than `10.5`, `getgroups()` returns the current group access list for the user associated with the effective user id of the process; the group access list may change over the lifetime of the process, it is not affected by calls to `setgroups()`, and its length is not limited to 16. The deployment target value, `MACOSX_DEPLOYMENT_TARGET`, can be obtained with `sysconfig.get_config_var()`.

os.**getlogin**()

Return the name of the user logged in on the controlling terminal of the process. For most purposes, it is more useful to use `getpass.getuser()` since the latter checks the environment variables `LOGNAME` or `USERNAME` to find out who the user is, and falls back to `pwd.getpwuid(os.getuid())[0]` to get the login name of the current real user id.

Availability: Unix, Windows, not WASI.

os.**getpgid**(*pid*)

Return the process group id of the process with process id *pid*. If *pid* is 0, the process group id of the current process is returned.

Availability: Unix, not WASI.

os.**getpgrp**()

Return the id of the current process group.

Availability: Unix, not WASI.

os.**getpid**()

Return the current process id.

The function is a stub on WASI, see WebAssembly platforms for more information.

os.**getppid**()

Return the parent's process id. When the parent process has exited, on Unix the id returned is the one of the init process (1), on Windows it is still the same id, which may be already reused by another process.

Availability: Unix, Windows, not WASI.

> *Changed in version 3.2:* Added support for Windows.

os.**getpriority**(*which*, *who*)

Get program scheduling priority. The value *which* is one of `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`, and *who* is interpreted relative to *which* (a process identifier for `PRIO_PROCESS`, process group identifier for `PRIO_PGRP`, and a user ID for `PRIO_USER`). A zero value for *who* denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process.

Availability: Unix, not WASI.

> *Added in version 3.3.*

os.**PRIO_PROCESS**
os.**PRIO_PGRP**
os.**PRIO_USER**

Parameters for the `getpriority()` and `setpriority()` functions.

Availability: Unix, not WASI.

> *Added in version 3.3.*

os.**PRIO_DARWIN_THREAD**
os.**PRIO_DARWIN_PROCESS**
os.**PRIO_DARWIN_BG**
os.**PRIO_DARWIN_NONUI**

Parameters for the `getpriority()` and `setpriority()` functions.

Availability: macOS

> *Added in version 3.12.*

os.**getresuid**()

Return a tuple (ruid, euid, suid) denoting the current process's real, effective, and saved user ids.

Availability: Unix, not WASI.

> *Added in version 3.2.*

os.**getresgid**()

Return a tuple (rgid, egid, sgid) denoting the current process's real, effective, and saved group ids.

Availability: Unix, not WASI.

> *Added in version 3.2.*

os.**getuid**()

Return the current process's real user id.

Availability: Unix.

The function is a stub on WASI, see WebAssembly platforms for more information.

os.**initgroups**(*username*, *gid*, /)

Call the system initgroups() to initialize the group access list with all of the groups of which the specified username is a member, plus the specified group id.

Availability: Unix, not WASI, not Android.

> *Added in version 3.2.*

os.**putenv**(*key*, *value*, /)

Set the environment variable named *key* to the string *value*. Such changes to the environment affect sub-processes started with `os.system()`, `popen()` or `fork()` and `execv()`.

Assignments to items in `os.environ` are automatically translated into corresponding calls to `putenv()`; however, calls to `putenv()` don't update `os.environ`, so it is actually preferable to assign to items of `os.environ`. This also applies to `getenv()` and `getenvb()`, which respectively use `os.environ` and `os.environb` in their implementations.

See also the `os.reload_environ()` function.

> **Note:** On some platforms, including FreeBSD and macOS, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv()`.

Raises an auditing event `os.putenv` with arguments `key`, `value`.

> *Changed in version 3.9:* The function is now always available.

os.**setegid**(*egid*, /)

Set the current process's effective group id.

Availability: Unix, not WASI, not Android.

os.**seteuid**(*euid*, /)

Set the current process's effective user id.

Availability: Unix, not WASI, not Android.

os.**setgid**(*gid*, /)

Set the current process' group id.

Availability: Unix, not WASI, not Android.

os.**setgroups**(*groups*, /)

Set the list of supplemental group ids associated with the current process to *groups*. *groups* must be a sequence, and each element must be an integer identifying a group. This operation is typically available only to the superuser.

Availability: Unix, not WASI.

> **Note:** On macOS, the length of *groups* may not exceed the system-defined maximum number of effective group ids, typically 16. See the documentation for `getgroups()` for cases where it may not return the same group list set by calling setgroups().

os.**setns**(*fd*, *nstype=0*)

Reassociate the current thread with a Linux namespace. See the *setns(2)* and *namespaces(7)* man pages for more details.

If *fd* refers to a `/proc/pid/ns/` link, `setns()` reassociates the calling thread with the namespace associated with that link, and *nstype* may be set to one of the CLONE_NEW* constants to impose constraints on the operation (`0` means no constraints).

Since Linux 5.8, *fd* may refer to a PID file descriptor obtained from `pidfd_open()`. In this case, `setns()` reassociates the calling thread into one or more of the same namespaces as the thread referred to by *fd*. This is subject to any constraints imposed by *nstype*, which is a bit mask combining one or more of the CLONE_NEW* constants, e.g. `setns(fd, os.CLONE_NEWUTS | os.CLONE_NEWPID)`. The caller's memberships in unspecified namespaces are left unchanged.

fd can be any object with a `fileno()` method, or a raw file descriptor.

This example reassociates the thread with the `init` process's network namespace:

```
fd = os.open("/proc/1/ns/net", os.O_RDONLY)
os.setns(fd, os.CLONE_NEWNET)
os.close(fd)
```

Availability: Linux >= 3.0 with glibc >= 2.14.

> *Added in version 3.12.*

> **See also:** The `unshare()` function.

os.**setpgrp**()

Call the system call `setpgrp()` or `setpgrp(0, 0)` depending on which version is implemented (if any). See the Unix manual for the semantics.

Availability: Unix, not WASI.

os.**setpgid**(*pid*, *pgrp*, /)

Call the system call `setpgid()` to set the process group id of the process with id *pid* to the process group with id *pgrp*. See the Unix manual for the semantics.

Availability: Unix, not WASI.

os.**setpriority**(*which*, *who*, *priority*)

Set program scheduling priority. The value *which* is one of `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`, and *who* is interpreted relative to *which* (a process identifier for `PRIO_PROCESS`, process group identifier for `PRIO_PGRP`, and a user ID for `PRIO_USER`). A zero value for *who* denotes (respectively) the calling process, the process group of the calling process, or the real user ID of the calling process. *priority* is a value in the range -20 to 19. The default priority is 0; lower priorities cause more favorable scheduling.

Availability: Unix, not WASI.

> *Added in version 3.3.*

os.**setregid**(*rgid*, *egid*, /)

Set the current process's real and effective group ids.

Availability: Unix, not WASI, not Android.

os.**setresgid**(*rgid*, *egid*, *sgid*, /)

Set the current process's real, effective, and saved group ids.

Availability: Unix, not WASI, not Android.

> *Added in version 3.2.*

os.**setresuid**(*ruid*, *euid*, *suid*, /)

Set the current process's real, effective, and saved user ids.

Availability: Unix, not WASI, not Android.

> *Added in version 3.2.*

os.**setreuid**(*ruid*, *euid*, /)

Set the current process's real and effective user ids.

Availability: Unix, not WASI, not Android.

os.**getsid**(*pid*, /)

Call the system call `getsid()`. See the Unix manual for the semantics.

Availability: Unix, not WASI.

os.**setsid**()

Call the system call `setsid()`. See the Unix manual for the semantics.

Availability: Unix, not WASI.

os.**setuid**(*uid*, /)

Set the current process's user id.

Availability: Unix, not WASI, not Android.

os.**strerror**(*code*, /)

Return the error message corresponding to the error code in *code*. On platforms where `strerror()` returns `NULL` when given an unknown error number, `ValueError` is raised.

os.**supports_bytes_environ**

`True` if the native OS type of the environment is bytes (eg. `False` on Windows).

> *Added in version 3.2.*

`os.umask(mask, /)`

Set the current numeric umask and return the previous umask.

The function is a stub on WASI, see WebAssembly platforms for more information.

`os.uname()`

Returns information identifying the current operating system. The return value is an object with five attributes:

- `sysname` - operating system name
- `nodename` - name of machine on network (implementation-defined)
- `release` - operating system release
- `version` - operating system version
- `machine` - hardware identifier

For backwards compatibility, this object is also iterable, behaving like a five-tuple containing `sysname`, `nodename`, `release`, `version`, and `machine` in that order.

Some systems truncate `nodename` to 8 characters or to the leading component; a better way to get the hostname is `socket.gethostname()` or even `socket.gethostbyaddr(socket.gethostname())`.

On macOS, iOS and Android, this returns the *kernel* name and version (i.e., `'Darwin'` on macOS and iOS; `'Linux'` on Android). `platform.uname()` can be used to get the user-facing operating system name and version on iOS and Android.

Availability: Unix.

> *Changed in version 3.3:* Return type changed from a tuple to a tuple-like object with named attributes.

`os.unsetenv(key, /)`

Unset (delete) the environment variable named *key*. Such changes to the environment affect subprocesses started with `os.system()`, `popen()` or `fork()` and `execv()`.

Deletion of items in `os.environ` is automatically translated into a corresponding call to `unsetenv()`; however, calls to `unsetenv()` don't update `os.environ`, so it is actually preferable to delete items of `os.environ`.

See also the `os.reload_environ()` function.

Raises an auditing event `os.unsetenv` with argument `key`.

> *Changed in version 3.9:* The function is now always available and is also available on Windows.

`os.unshare(flags)`

Disassociate parts of the process execution context, and move them into a newly created namespace. See the *unshare(2)* man page for more details. The *flags* argument is a bit mask, combining zero or more of the CLONE_* constants, that specifies which parts of the execution context should be unshared from their

existing associations and moved to a new namespace. If the *flags* argument is `0`, no changes are made to the calling process's execution context.

Availability: Linux >= 2.6.16.

> *Added in version 3.12.*

See also: The `setns()` function.

Flags to the `unshare()` function, if the implementation supports them. See *unshare(2)* in the Linux manual for their exact effect and availability.

os.**CLONE_FILES**
os.**CLONE_FS**
os.**CLONE_NEWCGROUP**
os.**CLONE_NEWIPC**
os.**CLONE_NEWNET**
os.**CLONE_NEWNS**
os.**CLONE_NEWPID**
os.**CLONE_NEWTIME**
os.**CLONE_NEWUSER**
os.**CLONE_NEWUTS**
os.**CLONE_SIGHAND**
os.**CLONE_SYSVSEM**
os.**CLONE_THREAD**
os.**CLONE_VM**

File Object Creation

These functions create new file objects. (See also `open()` for opening file descriptors.)

os.**fdopen**(*fd*, **args*, ***kwargs*)

Return an open file object connected to the file descriptor *fd*. This is an alias of the `open()` built-in function and accepts the same arguments. The only difference is that the first argument of `fdopen()` must always be an integer.

File Descriptor Operations

These functions operate on I/O streams referenced using file descriptors.

File descriptors are small integers corresponding to a file that has been opened by the current process. For example, standard input is usually file descriptor 0, standard output is 1, and standard error is 2. Further files opened by a process will then be assigned 3, 4, 5, and so forth. The name "file descriptor" is slightly deceptive; on Unix platforms, sockets and pipes are also referenced by file descriptors.

The `fileno()` method can be used to obtain the file descriptor associated with a file object when required. Note that using the file descriptor directly will bypass the file object methods, ignoring aspects such as internal buffering of data.

os.**close**(*fd*)

Close file descriptor *fd*.

> **Note:** This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To close a "file object" returned by the built-in function `open()` or by `popen()` or `fdopen()`, use its `close()` method.

os.**closerange**(*fd_low*, *fd_high*, */*)

Close all file descriptors from *fd_low* (inclusive) to *fd_high* (exclusive), ignoring errors. Equivalent to (but much faster than):

```
for fd in range(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

os.**copy_file_range**(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*)

Copy *count* bytes from file descriptor *src*, starting from offset *offset_src*, to file descriptor *dst*, starting from offset *offset_dst*. If *offset_src* is `None`, then *src* is read from the current position; respectively for *offset_dst*.

In Linux kernel older than 5.3, the files pointed to by *src* and *dst* must reside in the same filesystem, otherwise an `OSError` is raised with `errno` set to `errno.EXDEV`.

This copy is done without the additional cost of transferring data from the kernel to user space and then back into the kernel. Additionally, some filesystems could implement extra optimizations, such as the use of reflinks (i.e., two or more inodes that share pointers to the same copy-on-write disk blocks; supported file systems include btrfs and XFS) and server-side copy (in the case of NFS).

The function copies bytes between two file descriptors. Text options, like the encoding and the line ending, are ignored.

The return value is the amount of bytes copied. This could be less than the amount requested.

> **Note:** On Linux, `os.copy_file_range()` should not be used for copying a range of a pseudo file from a special filesystem like procfs and sysfs. It will always copy no bytes and return 0 as if the file was empty because of a known Linux kernel issue.

Availability: Linux >= 4.5 with glibc >= 2.27.

> *Added in version 3.8.*

os.**device_encoding**(*fd*)

Return a string describing the encoding of the device associated with *fd* if it is connected to a terminal; else return `None`.

On Unix, if the Python UTF-8 Mode is enabled, return `'UTF-8'` rather than the device encoding.

> *Changed in version 3.10:* On Unix, the function now implements the Python UTF-8 Mode.

os.**dup**(*fd*, */*)

Return a duplicate of file descriptor *fd*. The new file descriptor is non-inheritable.

On Windows, when duplicating a standard stream (0: stdin, 1: stdout, 2: stderr), the new file descriptor is inheritable.

Availability: not WASI.

> *Changed in version 3.4:* The new file descriptor is now non-inheritable.

os.**dup2**(*fd*, *fd2*, *inheritable=True*)

Duplicate file descriptor *fd* to *fd2*, closing the latter first if necessary. Return *fd2*. The new file descriptor is inheritable by default or non-inheritable if *inheritable* is `False`.

Availability: not WASI.

> *Changed in version 3.4:* Add the optional *inheritable* parameter.

> *Changed in version 3.7:* Return *fd2* on success. Previously, `None` was always returned.

os.**fchmod**(*fd*, *mode*)

Change the mode of the file given by *fd* to the numeric *mode*. See the docs for `chmod()` for possible values of *mode*. As of Python 3.3, this is equivalent to `os.chmod(fd, mode)`.

Raises an auditing event `os.chmod` with arguments `path`, `mode`, `dir_fd`.

Availability: Unix, Windows.

The function is limited on WASI, see WebAssembly platforms for more information.

> *Changed in version 3.13:* Added support on Windows.

os.**fchown**(*fd*, *uid*, *gid*)

Change the owner and group id of the file given by *fd* to the numeric *uid* and *gid*. To leave one of the ids unchanged, set it to -1. See `chown()`. As of Python 3.3, this is equivalent to `os.chown(fd, uid, gid)`.

Raises an auditing event `os.chown` with arguments `path`, `uid`, `gid`, `dir_fd`.

Availability: Unix.

The function is limited on WASI, see WebAssembly platforms for more information.

os.**fdatasync**(*fd*)

Force write of file with filedescriptor *fd* to disk. Does not force update of metadata.

Availability: Unix.

> **Note:** This function is not available on MacOS.

os.**fpathconf**(*fd*, *name*, */*)

Return system configuration information relevant to an open file. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If *name* is a string and is not known, `ValueError` is raised. If a specific value for *name* is not supported by the host system, even if it is included in `pathconf_names`, an `OSError` is raised with `errno.EINVAL` for the error number.

As of Python 3.3, this is equivalent to `os.pathconf(fd, name)`.

Availability: Unix.

`os.fstat(fd)`

Get the status of the file descriptor *fd*. Return a `stat_result` object.

As of Python 3.3, this is equivalent to `os.stat(fd)`.

> **See also:** The `stat()` function.

`os.fstatvfs(fd, /)`

Return information about the filesystem containing the file associated with file descriptor *fd*, like `statvfs()`. As of Python 3.3, this is equivalent to `os.statvfs(fd)`.

Availability: Unix.

`os.fsync(fd)`

Force write of file with filedescriptor *fd* to disk. On Unix, this calls the native `fsync()` function; on Windows, the MS `_commit()` function.

If you're starting with a buffered Python file object *f*, first do `f.flush()`, and then do `os.fsync(f.fileno())`, to ensure that all internal buffers associated with *f* are written to disk.

Availability: Unix, Windows.

`os.ftruncate(fd, length, /)`

Truncate the file corresponding to file descriptor *fd*, so that it is at most *length* bytes in size. As of Python 3.3, this is equivalent to `os.truncate(fd, length)`.

Raises an auditing event `os.truncate` with arguments `fd`, `length`.

Availability: Unix, Windows.

> *Changed in version 3.5:* Added support for Windows

`os.get_blocking(fd, /)`

Get the blocking mode of the file descriptor: `False` if the `O_NONBLOCK` flag is set, `True` if the flag is cleared.

See also `set_blocking()` and `socket.socket.setblocking()`.

Availability: Unix, Windows.

The function is limited on WASI, see WebAssembly platforms for more information.

On Windows, this function is limited to pipes.

> *Added in version 3.5.*

> *Changed in version 3.12:* Added support for pipes on Windows.

os.**grantpt**(*fd*, /)

Grant access to the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the C standard library function `grantpt()`.

Availability: Unix, not WASI.

> *Added in version 3.13.*

os.**isatty**(*fd*, /)

Return `True` if the file descriptor *fd* is open and connected to a tty(-like) device, else `False`.

os.**lockf**(*fd*, *cmd*, *len*, /)

Apply, test or remove a POSIX lock on an open file descriptor. *fd* is an open file descriptor. *cmd* specifies the command to use - one of `F_LOCK`, `F_TLOCK`, `F_ULOCK` or `F_TEST`. *len* specifies the section of the file to lock.

Raises an auditing event `os.lockf` with arguments `fd`, `cmd`, `len`.

Availability: Unix.

> *Added in version 3.3.*

os.**F_LOCK**
os.**F_TLOCK**
os.**F_ULOCK**
os.**F_TEST**

Flags that specify what action `lockf()` will take.

Availability: Unix.

> *Added in version 3.3.*

os.**login_tty**(*fd*, /)

Prepare the tty of which fd is a file descriptor for a new login session. Make the calling process a session leader; make the tty the controlling tty, the stdin, the stdout, and the stderr of the calling process; close fd.

Availability: Unix, not WASI.

> *Added in version 3.11.*

os.**lseek**(*fd*, *pos*, *whence*, /)

Set the current position of file descriptor *fd* to position *pos*, modified by *whence*, and return the new position in bytes relative to the start of the file. Valid values for *whence* are:

- `SEEK_SET` or `0` – set *pos* relative to the beginning of the file
- `SEEK_CUR` or `1` – set *pos* relative to the current file position
- `SEEK_END` or `2` – set *pos* relative to the end of the file
- `SEEK_HOLE` – set *pos* to the next data location, relative to *pos*
- `SEEK_DATA` – set *pos* to the next data hole, relative to *pos*

> *Changed in version 3.3:* Add support for `SEEK_HOLE` and `SEEK_DATA`.

os.**SEEK_SET**
os.**SEEK_CUR**
os.**SEEK_END**

Parameters to the `lseek()` function and the `seek()` method on file-like objects, for whence to adjust the file position indicator.

`SEEK_SET`

Adjust the file position relative to the beginning of the file.

`SEEK_CUR`

Adjust the file position relative to the current file position.

`SEEK_END`

Adjust the file position relative to the end of the file.

Their values are 0, 1, and 2, respectively.

os.**SEEK_HOLE**
os.**SEEK_DATA**

Parameters to the `lseek()` function and the `seek()` method on file-like objects, for seeking file data and holes on sparsely allocated files.

`SEEK_DATA`

Adjust the file offset to the next location containing data, relative to the seek position.

`SEEK_HOLE`

Adjust the file offset to the next location containing a hole, relative to the seek position. A hole is defined as a sequence of zeros.

> **Note:** These operations only make sense for filesystems that support them.

Availability: Linux >= 3.1, macOS, Unix

> *Added in version 3.3.*

os.**open**(*path*, *flags*, *mode=0o777*, ***, *dir_fd=None*)

Open the file *path* and set various flags according to *flags* and possibly its mode according to *mode*. When computing *mode*, the current umask value is first masked out. Return the file descriptor for the newly opened file. The new file descriptor is non-inheritable.

For a description of the flag and mode values, see the C run-time documentation; flag constants (like `O_RDONLY` and `O_WRONLY`) are defined in the `os` module. In particular, on Windows adding `O_BINARY` is needed to open files in binary mode.

This function can support paths relative to directory descriptors with the *dir_fd* parameter.

Raises an auditing event `open` with arguments `path`, `mode`, `flags`.

> *Changed in version 3.4:* The new file descriptor is now non-inheritable.

Note: This function is intended for low-level I/O. For normal usage, use the built-in function `open()`, which returns a file object with `read()` and `write()` methods (and many more). To wrap a file descriptor in a file object, use `fdopen()`.

> *Changed in version 3.3:* Added the *dir_fd* parameter.

> *Changed in version 3.5:* If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

> *Changed in version 3.6:* Accepts a path-like object.

The following constants are options for the *flags* parameter to the `open()` function. They can be combined using the bitwise OR operator `|`. Some of them are not available on all platforms. For descriptions of their availability and use, consult the *open(2)* manual page on Unix or the MSDN on Windows.

os.**O_RDONLY**
os.**O_WRONLY**
os.**O_RDWR**
os.**O_APPEND**
os.**O_CREAT**
os.**O_EXCL**
os.**O_TRUNC**

The above constants are available on Unix and Windows.

os.**O_DSYNC**
os.**O_RSYNC**
os.**O_SYNC**
os.**O_NDELAY**
os.**O_NONBLOCK**
os.**O_NOCTTY**
os.**O_CLOEXEC**

The above constants are only available on Unix.

> *Changed in version 3.3:* Add `O_CLOEXEC` constant.

os.**O_BINARY**
os.**O_NOINHERIT**
os.**O_SHORT_LIVED**
os.**O_TEMPORARY**
os.**O_RANDOM**
os.**O_SEQUENTIAL**
os.**O_TEXT**

The above constants are only available on Windows.

os.**O_EVTONLY**
os.**O_FSYNC**
os.**O_SYMLINK**
os.**O_NOFOLLOW_ANY**

The above constants are only available on macOS.

> *Changed in version 3.10:* Add `O_EVTONLY`, `O_FSYNC`, `O_SYMLINK` and `O_NOFOLLOW_ANY` constants.

os.**O_ASYNC**

os.**O_DIRECT**
os.**O_DIRECTORY**
os.**O_NOFOLLOW**
os.**O_NOATIME**
os.**O_PATH**
os.**O_TMPFILE**
os.**O_SHLOCK**
os.**O_EXLOCK**

The above constants are extensions and not present if they are not defined by the C library.

> *Changed in version 3.4:* Add `O_PATH` on systems that support it. Add `O_TMPFILE`, only available on Linux Kernel 3.11 or newer.

os.**openpty**()

Open a new pseudo-terminal pair. Return a pair of file descriptors `(master, slave)` for the pty and the tty, respectively. The new file descriptors are non-inheritable. For a (slightly) more portable approach, use the `pty` module.

Availability: Unix, not WASI.

> *Changed in version 3.4:* The new file descriptors are now non-inheritable.

os.**pipe**()

Create a pipe. Return a pair of file descriptors `(r, w)` usable for reading and writing, respectively. The new file descriptor is non-inheritable.

Availability: Unix, Windows.

> *Changed in version 3.4:* The new file descriptors are now non-inheritable.

os.**pipe2**(*flags*, /)

Create a pipe with *flags* set atomically. *flags* can be constructed by ORing together one or more of these values: `O_NONBLOCK`, `O_CLOEXEC`. Return a pair of file descriptors `(r, w)` usable for reading and writing, respectively.

Availability: Unix, not WASI.

> *Added in version 3.3.*

os.**posix_fallocate**(*fd*, *offset*, *len*, /)

Ensures that enough disk space is allocated for the file specified by *fd* starting from *offset* and continuing for *len* bytes.

Availability: Unix.

> *Added in version 3.3.*

os.**posix_fadvise**(*fd*, *offset*, *len*, *advice*, /)

Announces an intention to access data in a specific pattern thus allowing the kernel to make optimizations. The advice applies to the region of the file specified by *fd* starting at *offset* and continuing for *len* bytes. *advice* is one of `POSIX_FADV_NORMAL`, `POSIX_FADV_SEQUENTIAL`, `POSIX_FADV_RANDOM`, `POSIX_FADV_NOREUSE`, `POSIX_FADV_WILLNEED` or `POSIX_FADV_DONTNEED`.

Availability: Unix.

> *Added in version 3.3.*

os.**POSIX_FADV_NORMAL**
os.**POSIX_FADV_SEQUENTIAL**
os.**POSIX_FADV_RANDOM**
os.**POSIX_FADV_NOREUSE**
os.**POSIX_FADV_WILLNEED**
os.**POSIX_FADV_DONTNEED**

Flags that can be used in *advice* in `posix_fadvise()` that specify the access pattern that is likely to be used.

Availability: Unix.

> *Added in version 3.3.*

os.**pread**(*fd*, *n*, *offset*, /)

Read at most *n* bytes from file descriptor *fd* at a position of *offset*, leaving the file offset unchanged.

Return a bytestring containing the bytes read. If the end of the file referred to by *fd* has been reached, an empty bytes object is returned.

Availability: Unix.

> *Added in version 3.3.*

os.**posix_openpt**(*oflag*, /)

Open and return a file descriptor for a master pseudo-terminal device.

Calls the C standard library function `posix_openpt()`. The *oflag* argument is used to set file status flags and file access modes as specified in the manual page of `posix_openpt()` of your system.

The returned file descriptor is non-inheritable. If the value `O_CLOEXEC` is available on the system, it is added to *oflag*.

Availability: Unix, not WASI.

> *Added in version 3.13.*

os.**preadv**(*fd*, *buffers*, *offset*, *flags=0*, /)

Read from a file descriptor *fd* at a position of *offset* into mutable bytes-like objects *buffers*, leaving the file offset unchanged. Transfer data into each buffer until it is full and then move on to the next buffer in the sequence to hold the rest of the data.

The flags argument contains a bitwise OR of zero or more of the following flags:

- `RWF_HIPRI`
- `RWF_NOWAIT`

Return the total number of bytes actually read which can be less than the total capacity of all the objects.

The operating system may set a limit (`sysconf()` value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Combine the functionality of `os.readv()` and `os.pread()`.

Availability: Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

Using flags requires Linux >= 4.6.

> *Added in version 3.7.*

os.**RWF_NOWAIT**

Do not wait for data which is not immediately available. If this flag is specified, the system call will return instantly if it would have to read data from the backing storage or wait for a lock.

If some data was successfully read, it will return the number of bytes read. If no bytes were read, it will return `-1` and set errno to `errno.EAGAIN`.

Availability: Linux >= 4.14.

> *Added in version 3.7.*

os.**RWF_HIPRI**

High priority read/write. Allows block-based filesystems to use polling of the device, which provides lower latency, but may use additional resources.

Currently, on Linux, this feature is usable only on a file descriptor opened using the `O_DIRECT` flag.

Availability: Linux >= 4.6.

> *Added in version 3.7.*

os.**ptsname**(*fd*, */*)

Return the name of the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the reentrant C standard library function `ptsname_r()` if it is available; otherwise, the C standard library function `ptsname()`, which is not guaranteed to be thread-safe, is called.

Availability: Unix, not WASI.

> *Added in version 3.13.*

os.**pwrite**(*fd*, *str*, *offset*, */*)

Write the bytestring in *str* to file descriptor *fd* at position of *offset*, leaving the file offset unchanged.

Return the number of bytes actually written.

Availability: Unix.

> *Added in version 3.3.*

os.**pwritev**(*fd*, *buffers*, *offset*, *flags=0*, */*)

Write the *buffers* contents to file descriptor *fd* at an offset *offset*, leaving the file offset unchanged. *buffers* must be a sequence of bytes-like objects. Buffers are processed in array order. Entire contents of the first buffer is written before proceeding to the second, and so on.

The flags argument contains a bitwise OR of zero or more of the following flags:

- `RWF_DSYNC`
- `RWF_SYNC`
- `RWF_APPEND`

Return the total number of bytes actually written.

The operating system may set a limit (`sysconf()` value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Combine the functionality of `os.writev()` and `os.pwrite()`.

Availability: Linux >= 2.6.30, FreeBSD >= 6.0, OpenBSD >= 2.7, AIX >= 7.1.

Using flags requires Linux >= 4.6.

> *Added in version 3.7.*

os.**RWF_DSYNC**

Provide a per-write equivalent of the `O_DSYNC` `os.open()` flag. This flag effect applies only to the data range written by the system call.

Availability: Linux >= 4.7.

> *Added in version 3.7.*

os.**RWF_SYNC**

Provide a per-write equivalent of the `O_SYNC` `os.open()` flag. This flag effect applies only to the data range written by the system call.

Availability: Linux >= 4.7.

> *Added in version 3.7.*

os.**RWF_APPEND**

Provide a per-write equivalent of the `O_APPEND` `os.open()` flag. This flag is meaningful only for `os.pwritev()`, and its effect applies only to the data range written by the system call. The *offset* argument does not affect the write operation; the data is always appended to the end of the file. However, if the *offset* argument is `-1`, the current file *offset* is updated.

Availability: Linux >= 4.16.

> *Added in version 3.10.*

os.**read**(*fd*, *n*, /)

Read at most *n* bytes from file descriptor *fd*.

Return a bytestring containing the bytes read. If the end of the file referred to by *fd* has been reached, an empty bytes object is returned.

> **Note:** This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To read a “file object” returned by the built-in function `open()` or by `popen()`

or `fdopen()`, or `sys.stdin`, use its `read()` or `readline()` methods.

> *Changed in version 3.5:* If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

os.**readinto**(*fd*, *buffer*, */*)

Read from a file descriptor *fd* into a mutable buffer object *buffer*.

The *buffer* should be mutable and bytes-like. On success, returns the number of bytes read. Less bytes may be read than the size of the buffer. The underlying system call will be retried when interrupted by a signal, unless the signal handler raises an exception. Other errors will not be retried and an error will be raised.

Returns 0 if *fd* is at end of file or if the provided *buffer* has length 0 (which can be used to check for errors without reading data). Never returns negative.

> **Note:** This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `os.pipe()`. To read a "file object" returned by the built-in function `open()`, or `sys.stdin`, use its member functions, for example `io.BufferedIOBase.readinto()`, `io.BufferedIOBase.read()`, or `io.TextIOBase.read()`

> *Added in version 3.14.*

os.**sendfile**(*out_fd*, *in_fd*, *offset*, *count*)
os.**sendfile**(*out_fd*, *in_fd*, *offset*, *count*, *headers=()*, *trailers=()*, *flags=0*)

Copy *count* bytes from file descriptor *in_fd* to file descriptor *out_fd* starting at *offset*. Return the number of bytes sent. When EOF is reached return `0`.

The first function notation is supported by all platforms that define `sendfile()`.

On Linux, if *offset* is given as `None`, the bytes are read from the current position of *in_fd* and the position of *in_fd* is updated.

The second case may be used on macOS and FreeBSD where *headers* and *trailers* are arbitrary sequences of buffers that are written before and after the data from *in_fd* is written. It returns the same as the first case.

On macOS and FreeBSD, a value of `0` for *count* specifies to send until the end of *in_fd* is reached.

All platforms support sockets as *out_fd* file descriptor, and some platforms allow other types (e.g. regular file, pipe) as well.

Cross-platform applications should not use *headers*, *trailers* and *flags* arguments.

Availability: Unix, not WASI.

> **Note:** For a higher-level wrapper of `sendfile()`, see `socket.socket.sendfile()`.

> *Added in version 3.3.*

> *Changed in version 3.9:* Parameters *out* and *in* was renamed to *out_fd* and *in_fd*.

os.**SF_NODISKIO**
os.**SF_MNOWAIT**
os.**SF_SYNC**

Parameters to the `sendfile()` function, if the implementation supports them.

Availability: Unix, not WASI.

> *Added in version 3.3.*

os.**SF_NOCACHE**

Parameter to the `sendfile()` function, if the implementation supports it. The data won't be cached in the virtual memory and will be freed afterwards.

Availability: Unix, not WASI.

> *Added in version 3.11.*

os.**set_blocking**(*fd*, *blocking*, /)

Set the blocking mode of the specified file descriptor. Set the `O_NONBLOCK` flag if blocking is `False`, clear the flag otherwise.

See also `get_blocking()` and `socket.socket.setblocking()`.

Availability: Unix, Windows.

The function is limited on WASI, see WebAssembly platforms for more information.

On Windows, this function is limited to pipes.

> *Added in version 3.5.*

> *Changed in version 3.12:* Added support for pipes on Windows.

os.**splice**(*src*, *dst*, *count*, *offset_src=None*, *offset_dst=None*, *flags=0*)

Transfer *count* bytes from file descriptor *src*, starting from offset *offset_src*, to file descriptor *dst*, starting from offset *offset_dst*.

The splicing behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the | operator):

- If `SPLICE_F_MOVE` is specified, the kernel is asked to move pages instead of copying, but pages may still be copied if the kernel cannot move the pages from the pipe.
- If `SPLICE_F_NONBLOCK` is specified, the kernel is asked to not block on I/O. This makes the splice pipe operations nonblocking, but splice may nevertheless block because the spliced file descriptors may block.
- If `SPLICE_F_MORE` is specified, it hints to the kernel that more data will be coming in a subsequent splice.

At least one of the file descriptors must refer to a pipe. If *offset_src* is `None`, then *src* is read from the current position; respectively for *offset_dst*. The offset associated to the file descriptor that refers to a pipe

must be `None`. The files pointed to by *src* and *dst* must reside in the same filesystem, otherwise an `OSError` is raised with `errno` set to `errno.EXDEV`.

This copy is done without the additional cost of transferring data from the kernel to user space and then back into the kernel. Additionally, some filesystems could implement extra optimizations. The copy is done as if both files are opened as binary.

Upon successful completion, returns the number of bytes spliced to or from the pipe. A return value of 0 means end of input. If *src* refers to a pipe, then this means that there was no data to transfer, and it would not make sense to block because there are no writers connected to the write end of the pipe.

> **See also:** The *splice(2)* man page.

Availability: Linux >= 2.6.17 with glibc >= 2.5

> *Added in version 3.10.*

os.**SPLICE_F_MOVE**
os.**SPLICE_F_NONBLOCK**
os.**SPLICE_F_MORE**

> *Added in version 3.10.*

os.**readv**(*fd*, *buffers*, */*)

Read from a file descriptor *fd* into a number of mutable bytes-like objects *buffers*. Transfer data into each buffer until it is full and then move on to the next buffer in the sequence to hold the rest of the data.

Return the total number of bytes actually read which can be less than the total capacity of all the objects.

The operating system may set a limit (`sysconf()` value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Availability: Unix.

> *Added in version 3.3.*

os.**tcgetpgrp**(*fd*, */*)

Return the process group associated with the terminal given by *fd* (an open file descriptor as returned by `os.open()`).

Availability: Unix, not WASI.

os.**tcsetpgrp**(*fd*, *pg*, */*)

Set the process group associated with the terminal given by *fd* (an open file descriptor as returned by `os.open()`) to *pg*.

Availability: Unix, not WASI.

os.**ttyname**(*fd*, */*)

Return a string which specifies the terminal device associated with file descriptor *fd*. If *fd* is not associated with a terminal device, an exception is raised.

Availability: Unix.

os.**unlockpt**(*fd*, */*)

Unlock the slave pseudo-terminal device associated with the master pseudo-terminal device to which the file descriptor *fd* refers. The file descriptor *fd* is not closed upon failure.

Calls the C standard library function `unlockpt()`.

Availability: Unix, not WASI.

> *Added in version 3.13.*

os.**write**(*fd*, *str*, */*)

Write the bytestring in *str* to file descriptor *fd*.

Return the number of bytes actually written.

> **Note:** This function is intended for low-level I/O and must be applied to a file descriptor as returned by `os.open()` or `pipe()`. To write a "file object" returned by the built-in function `open()` or by `popen()` or `fdopen()`, or `sys.stdout` or `sys.stderr`, use its `write()` method.

> *Changed in version 3.5:* If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

os.**writev**(*fd*, *buffers*, */*)

Write the contents of *buffers* to file descriptor *fd*. *buffers* must be a sequence of bytes-like objects. Buffers are processed in array order. Entire contents of the first buffer is written before proceeding to the second, and so on.

Returns the total number of bytes actually written.

The operating system may set a limit (`sysconf()` value `'SC_IOV_MAX'`) on the number of buffers that can be used.

Availability: Unix.

> *Added in version 3.3.*

Querying the size of a terminal

> *Added in version 3.3.*

os.**get_terminal_size**(*fd=STDOUT_FILENO*, */*)

Return the size of the terminal window as `(columns, lines)`, tuple of type `terminal_size`.

The optional argument `fd` (default `STDOUT_FILENO`, or standard output) specifies which file descriptor should be queried.

If the file descriptor is not connected to a terminal, an `OSError` is raised.

`shutil.get_terminal_size()` is the high-level function which should normally be used, `os.get_terminal_size` is the low-level implementation.

Availability: Unix, Windows.

class os.**terminal_size**

A subclass of tuple, holding `(columns, lines)` of the terminal window size.

columns

Width of the terminal window in characters.

lines

Height of the terminal window in characters.

Inheritance of File Descriptors

> *Added in version 3.4.*

A file descriptor has an "inheritable" flag which indicates if the file descriptor can be inherited by child processes. Since Python 3.4, file descriptors created by Python are non-inheritable by default.

On UNIX, non-inheritable file descriptors are closed in child processes at the execution of a new program, other file descriptors are inherited.

On Windows, non-inheritable handles and file descriptors are closed in child processes, except for standard streams (file descriptors 0, 1 and 2: stdin, stdout and stderr), which are always inherited. Using `spawn*` functions, all inheritable handles and all inheritable file descriptors are inherited. Using the `subprocess` module, all file descriptors except standard streams are closed, and inheritable handles are only inherited if the *close_fds* parameter is `False`.

On WebAssembly platforms, the file descriptor cannot be modified.

os.**get_inheritable**(*fd*, */*)

Get the "inheritable" flag of the specified file descriptor (a boolean).

os.**set_inheritable**(*fd*, *inheritable*, */*)

Set the "inheritable" flag of the specified file descriptor.

os.**get_handle_inheritable**(*handle*, */*)

Get the "inheritable" flag of the specified handle (a boolean).

Availability: Windows.

os.**set_handle_inheritable**(*handle*, *inheritable*, */*)

Set the "inheritable" flag of the specified handle.

Availability: Windows.

Files and Directories

On some Unix platforms, many of these functions support one or more of these features:

- **specifying a file descriptor:** Normally the *path* argument provided to functions in the `os` module must be a string specifying a file path. However, some functions now alternatively accept an open file descriptor for their *path* argument. The function will then operate on the file referred to by the descriptor. For POSIX systems, Python will call the variant of the function prefixed with `f` (e.g. call `fchdir` instead of `chdir`).

You can check whether or not *path* can be specified as a file descriptor for a particular function on your platform using `os.supports_fd`. If this functionality is unavailable, using it will raise a `NotImplementedError`.

If the function also supports *dir_fd* or *follow_symlinks* arguments, it's an error to specify one of those when supplying *path* as a file descriptor.

- **paths relative to directory descriptors:** If *dir_fd* is not `None`, it should be a file descriptor referring to a directory, and the path to operate on should be relative; path will then be relative to that directory. If the path is absolute, *dir_fd* is ignored. For POSIX systems, Python will call the variant of the function with an `at` suffix and possibly prefixed with `f` (e.g. call `faccessat` instead of `access`).

  You can check whether or not *dir_fd* is supported for a particular function on your platform using `os.supports_dir_fd`. If it's unavailable, using it will raise a `NotImplementedError`.

- **not following symlinks:** If *follow_symlinks* is `False`, and the last element of the path to operate on is a symbolic link, the function will operate on the symbolic link itself rather than the file pointed to by the link. For POSIX systems, Python will call the `l...` variant of the function.

  You can check whether or not *follow_symlinks* is supported for a particular function on your platform using `os.supports_follow_symlinks`. If it's unavailable, using it will raise a `NotImplementedError`.

os.**access**(*path*, *mode*, ***, *dir_fd=None*, *effective_ids=False*, *follow_symlinks=True*)

Use the real uid/gid to test for access to *path*. Note that most operations will use the effective uid/gid, therefore this routine can be used in a suid/sgid environment to test if the invoking user has the specified access to *path*. *mode* should be `F_OK` to test the existence of *path*, or it can be the inclusive OR of one or more of `R_OK`, `W_OK`, and `X_OK` to test permissions. Return `True` if access is allowed, `False` if not. See the Unix man page *access(2)* for more information.

This function can support specifying paths relative to directory descriptors and not following symlinks.

If *effective_ids* is `True`, `access()` will perform its access checks using the effective uid/gid instead of the real uid/gid. *effective_ids* may not be supported on your platform; you can check whether or not it is available using `os.supports_effective_ids`. If it is unavailable, using it will raise a `NotImplementedError`.

> **Note:** Using `access()` to check if a user is authorized to e.g. open a file before actually doing so using `open()` creates a security hole, because the user might exploit the short time interval between checking and opening the file to manipulate it. It's preferable to use EAFP techniques. For example:
>
> ```
> if os.access("myfile", os.R_OK):
>     with open("myfile") as fp:
>         return fp.read()
> return "some default data"
> ```
>
> is better written as:
>
> ```
> try:
>     fp = open("myfile")
> except PermissionError:
>     return "some default data"
> else:
>     with fp:
>         return fp.read()
> ```

Note: I/O operations may fail even when `access()` indicates that they would succeed, particularly for operations on network filesystems which may have permissions semantics beyond the usual POSIX permission-bit model.

> *Changed in version 3.3:* Added the *dir_fd*, *effective_ids*, and *follow_symlinks* parameters.

> *Changed in version 3.6:* Accepts a path-like object.

os.**F_OK**
os.**R_OK**
os.**W_OK**
os.**X_OK**

Values to pass as the *mode* parameter of `access()` to test the existence, readability, writability and executability of *path*, respectively.

os.**chdir**(*path*)

Change the current working directory to *path*.

This function can support specifying a file descriptor. The descriptor must refer to an opened directory, not an open file.

This function can raise `OSError` and subclasses such as `FileNotFoundError`, `PermissionError`, and `NotADirectoryError`.

Raises an auditing event `os.chdir` with argument `path`.

> *Changed in version 3.3:* Added support for specifying *path* as a file descriptor on some platforms.

> *Changed in version 3.6:* Accepts a path-like object.

os.**chflags**(*path, flags, *, follow_symlinks=True*)

Set the flags of *path* to the numeric *flags*. *flags* may take a combination (bitwise OR) of the following values (as defined in the `stat` module):

- `stat.UF_NODUMP`
- `stat.UF_IMMUTABLE`
- `stat.UF_APPEND`
- `stat.UF_OPAQUE`
- `stat.UF_NOUNLINK`
- `stat.UF_COMPRESSED`
- `stat.UF_HIDDEN`
- `stat.SF_ARCHIVED`
- `stat.SF_IMMUTABLE`
- `stat.SF_APPEND`
- `stat.SF_NOUNLINK`
- `stat.SF_SNAPSHOT`

This function can support not following symlinks.

Raises an auditing event `os.chflags` with arguments `path`, `flags`.

Availability: Unix, not WASI.

> *Changed in version 3.3:* Added the *follow_symlinks* parameter.

> *Changed in version 3.6:* Accepts a path-like object.

os.**chmod**(*path*, *mode*, ***, *dir_fd=None*, *follow_symlinks=True*)

Change the mode of *path* to the numeric *mode*. *mode* may take one of the following values (as defined in the `stat` module) or bitwise ORed combinations of them:

- `stat.S_ISUID`
- `stat.S_ISGID`
- `stat.S_ENFMT`
- `stat.S_ISVTX`
- `stat.S_IREAD`
- `stat.S_IWRITE`
- `stat.S_IEXEC`
- `stat.S_IRWXU`
- `stat.S_IRUSR`
- `stat.S_IWUSR`
- `stat.S_IXUSR`
- `stat.S_IRWXG`
- `stat.S_IRGRP`
- `stat.S_IWGRP`
- `stat.S_IXGRP`
- `stat.S_IRWXO`
- `stat.S_IROTH`
- `stat.S_IWOTH`
- `stat.S_IXOTH`

This function can support specifying a file descriptor, paths relative to directory descriptors and not following symlinks.

> **Note:** Although Windows supports `chmod()`, you can only set the file's read-only flag with it (via the `stat.S_IWRITE` and `stat.S_IREAD` constants or a corresponding integer value). All other bits are ignored. The default value of *follow_symlinks* is `False` on Windows.
>
> The function is limited on WASI, see WebAssembly platforms for more information.

Raises an auditing event `os.chmod` with arguments `path`, `mode`, `dir_fd`.

> *Changed in version 3.3:* Added support for specifying *path* as an open file descriptor, and the *dir_fd* and *follow_symlinks* arguments.

> *Changed in version 3.6:* Accepts a path-like object.

> *Changed in version 3.13:* Added support for a file descriptor and the *follow_symlinks* argument on Windows.

os.**chown**(*path*, *uid*, *gid*, ***, *dir_fd=None*, *follow_symlinks=True*)

Change the owner and group id of *path* to the numeric *uid* and *gid*. To leave one of the ids unchanged, set it to -1.

This function can support specifying a file descriptor, paths relative to directory descriptors and not following symlinks.

See `shutil.chown()` for a higher-level function that accepts names in addition to numeric ids.

Raises an auditing event `os.chown` with arguments `path`, `uid`, `gid`, `dir_fd`.

Availability: Unix.

The function is limited on WASI, see WebAssembly platforms for more information.

> *Changed in version 3.3:* Added support for specifying *path* as an open file descriptor, and the *dir_fd* and *follow_symlinks* arguments.

> *Changed in version 3.6:* Supports a path-like object.

os.**chroot**(*path*)

Change the root directory of the current process to *path*.

Availability: Unix, not WASI, not Android.

> *Changed in version 3.6:* Accepts a path-like object.

os.**fchdir**(*fd*)

Change the current working directory to the directory represented by the file descriptor *fd*. The descriptor must refer to an opened directory, not an open file. As of Python 3.3, this is equivalent to `os.chdir(fd)`.

Raises an auditing event `os.chdir` with argument `path`.

Availability: Unix.

os.**getcwd**()

Return a string representing the current working directory.

os.**getcwdb**()

Return a bytestring representing the current working directory.

> *Changed in version 3.8:* The function now uses the UTF-8 encoding on Windows, rather than the ANSI code page: see **PEP 529** for the rationale. The function is no longer deprecated on Windows.

os.**lchflags**(*path, flags*)

Set the flags of *path* to the numeric *flags*, like `chflags()`, but do not follow symbolic links. As of Python 3.3, this is equivalent to `os.chflags(path, flags, follow_symlinks=False)`.

Raises an auditing event `os.chflags` with arguments `path`, `flags`.

Availability: Unix, not WASI.

> *Changed in version 3.6:* Accepts a path-like object.

os.**lchmod**(*path, mode*)

Change the mode of *path* to the numeric *mode*. If path is a symlink, this affects the symlink rather than the target. See the docs for `chmod()` for possible values of *mode*. As of Python 3.3, this is equivalent to `os.chmod(path, mode, follow_symlinks=False)`.

`lchmod()` is not part of POSIX, but Unix implementations may have it if changing the mode of symbolic links is supported.

Raises an auditing event `os.chmod` with arguments `path`, `mode`, `dir_fd`.

Availability: Unix, Windows, not Linux, FreeBSD >= 1.3, NetBSD >= 1.3, not OpenBSD

> *Changed in version 3.6:* Accepts a path-like object.

> *Changed in version 3.13:* Added support on Windows.

os.lchown(*path, uid, gid*)

Change the owner and group id of *path* to the numeric *uid* and *gid*. This function will not follow symbolic links. As of Python 3.3, this is equivalent to `os.chown(path, uid, gid, follow_symlinks=False)`.

Raises an auditing event `os.chown` with arguments `path`, `uid`, `gid`, `dir_fd`.

Availability: Unix.

> *Changed in version 3.6:* Accepts a path-like object.

os.link(*src, dst, *, src_dir_fd=None, dst_dir_fd=None, follow_symlinks=True*)

Create a hard link pointing to *src* named *dst*.

This function can support specifying *src_dir_fd* and/or *dst_dir_fd* to supply paths relative to directory descriptors, and not following symlinks. The default value of *follow_symlinks* is `False` on Windows.

Raises an auditing event `os.link` with arguments `src`, `dst`, `src_dir_fd`, `dst_dir_fd`.

Availability: Unix, Windows.

> *Changed in version 3.2:* Added Windows support.

> *Changed in version 3.3:* Added the *src_dir_fd*, *dst_dir_fd*, and *follow_symlinks* parameters.

> *Changed in version 3.6:* Accepts a path-like object for *src* and *dst*.

os.listdir(*path='.'*)

Return a list containing the names of the entries in the directory given by *path*. The list is in arbitrary order, and does not include the special entries `'.'` and `'..'` even if they are present in the directory. If a file is removed from or added to the directory during the call of this function, whether a name for that file be included is unspecified.

path may be a path-like object. If *path* is of type `bytes` (directly or indirectly through the `PathLike` interface), the filenames returned will also be of type `bytes`; in all other circumstances, they will be of type `str`.

This function can also support specifying a file descriptor; the file descriptor must refer to a directory.

Raises an auditing event `os.listdir` with argument `path`.

Note: To encode `str` filenames to `bytes`, use `fsencode()`.

See also: The `scandir()` function returns directory entries along with file attribute information, giving better performance for many common use cases.

> *Changed in version 3.2:* The *path* parameter became optional.

> *Changed in version 3.3:* Added support for specifying *path* as an open file descriptor.

> *Changed in version 3.6:* Accepts a path-like object.

`os.listdrives()`

Return a list containing the names of drives on a Windows system.

A drive name typically looks like `'C:\\'`. Not every drive name will be associated with a volume, and some may be inaccessible for a variety of reasons, including permissions, network connectivity or missing media. This function does not test for access.

May raise `OSError` if an error occurs collecting the drive names.

Raises an auditing event `os.listdrives` with no arguments.

Availability: Windows

> *Added in version 3.12.*

`os.listmounts(volume)`

Return a list containing the mount points for a volume on a Windows system.

volume must be represented as a GUID path, like those returned by `os.listvolumes()`. Volumes may be mounted in multiple locations or not at all. In the latter case, the list will be empty. Mount points that are not associated with a volume will not be returned by this function.

The mount points return by this function will be absolute paths, and may be longer than the drive name.

Raises `OSError` if the volume is not recognized or if an error occurs collecting the paths.

Raises an auditing event `os.listmounts` with argument `volume`.

Availability: Windows

> *Added in version 3.12.*

`os.listvolumes()`

Return a list containing the volumes in the system.

Volumes are typically represented as a GUID path that looks like `\\?\Volume{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}\`. Files can usually be accessed through a GUID path, permissions allowing. However, users are generally not familiar with them, and so the recommended use of this function is to retrieve mount points using `os.listmounts()`.

May raise `OSError` if an error occurs collecting the volumes.

Raises an auditing event `os.listvolumes` with no arguments.

Availability: Windows

> *Added in version 3.12.*

`os.lstat(path, *, dir_fd=None)`

Perform the equivalent of an `lstat()` system call on the given path. Similar to `stat()`, but does not follow symbolic links. Return a `stat_result` object.

On platforms that do not support symbolic links, this is an alias for `stat()`.

As of Python 3.3, this is equivalent to `os.stat(path, dir_fd=dir_fd, follow_symlinks=False)`.

This function can also support paths relative to directory descriptors.

See also: The `stat()` function.

> *Changed in version 3.2:* Added support for Windows 6.0 (Vista) symbolic links.

> *Changed in version 3.3:* Added the *dir_fd* parameter.

> *Changed in version 3.6:* Accepts a path-like object.

> *Changed in version 3.8:* On Windows, now opens reparse points that represent another path (name surrogates), including symbolic links and directory junctions. Other kinds of reparse points are resolved by the operating system as for `stat()`.

`os.mkdir(path, mode=0o777, *, dir_fd=None)`

Create a directory named *path* with numeric mode *mode*.

If the directory already exists, `FileExistsError` is raised. If a parent directory in the path does not exist, `FileNotFoundError` is raised.

On some systems, *mode* is ignored. Where it is used, the current umask value is first masked out. If bits other than the last 9 (i.e. the last 3 digits of the octal representation of the *mode*) are set, their meaning is platform-dependent. On some platforms, they are ignored and you should call `chmod()` explicitly to set them.

On Windows, a *mode* of `0o700` is specifically handled to apply access control to the new directory such that only the current user and administrators have access. Other values of *mode* are ignored.

This function can also support paths relative to directory descriptors.

It is also possible to create temporary directories; see the `tempfile` module's `tempfile.mkdtemp()` function.

Raises an auditing event `os.mkdir` with arguments `path`, `mode`, `dir_fd`.

> *Changed in version 3.3:* Added the *dir_fd* parameter.

> *Changed in version 3.6:* Accepts a path-like object.

> *Changed in version 3.13:* Windows now handles a *mode* of `0o700`.

os.**makedirs**(*name, mode=0o777, exist_ok=False*)

Recursive directory creation function. Like `mkdir()`, but makes all intermediate-level directories needed to contain the leaf directory.

The *mode* parameter is passed to `mkdir()` for creating the leaf directory; see the mkdir() description for how it is interpreted. To set the file permission bits of any newly created parent directories you can set the umask before invoking `makedirs()`. The file permission bits of existing parent directories are not changed.

If *exist_ok* is `False` (the default), a `FileExistsError` is raised if the target directory already exists.

> **Note:** `makedirs()` will become confused if the path elements to create include `pardir` (eg. ".." on UNIX systems).

This function handles UNC paths correctly.

Raises an auditing event `os.mkdir` with arguments `path`, `mode`, `dir_fd`.

> *Changed in version 3.2:* Added the *exist_ok* parameter.

> *Changed in version 3.4.1:* Before Python 3.4.1, if *exist_ok* was `True` and the directory existed, `makedirs()` would still raise an error if *mode* did not match the mode of the existing directory. Since this behavior was impossible to implement safely, it was removed in Python 3.4.1. See bpo-21082.

> *Changed in version 3.6:* Accepts a path-like object.

> *Changed in version 3.7:* The *mode* argument no longer affects the file permission bits of newly created intermediate-level directories.

os.**mkfifo**(*path, mode=0o666, *, dir_fd=None*)

Create a FIFO (a named pipe) named *path* with numeric mode *mode*. The current umask value is first masked out from the mode.

This function can also support paths relative to directory descriptors.

FIFOs are pipes that can be accessed like regular files. FIFOs exist until they are deleted (for example with `os.unlink()`). Generally, FIFOs are used as rendezvous between "client" and "server" type processes: the server opens the FIFO for reading, and the client opens it for writing. Note that `mkfifo()` doesn't open the FIFO — it just creates the rendezvous point.

Availability: Unix, not WASI.

> *Changed in version 3.3:* Added the *dir_fd* parameter.

> *Changed in version 3.6:* Accepts a path-like object.

os.**mknod**(*path, mode=0o600, device=0, *, dir_fd=None*)

Create a filesystem node (file, device special file or named pipe) named *path*. *mode* specifies both the permissions to use and the type of node to be created, being combined (bitwise OR) with one of `stat.S_IFREG`, `stat.S_IFCHR`, `stat.S_IFBLK`, and `stat.S_IFIFO` (those constants are available in `stat`).

For `stat.S_IFCHR` and `stat.S_IFBLK`, *device* defines the newly created device special file (probably using `os.makedev()`), otherwise it is ignored.

This function can also support paths relative to directory descriptors.

Availability: Unix, not WASI.

> *Changed in version 3.3:* Added the *dir_fd* parameter.

> *Changed in version 3.6:* Accepts a path-like object.

os.**major**(*device*, /)

Extract the device major number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

os.**minor**(*device*, /)

Extract the device minor number from a raw device number (usually the `st_dev` or `st_rdev` field from `stat`).

os.**makedev**(*major*, *minor*, /)

Compose a raw device number from the major and minor device numbers.

os.**pathconf**(*path*, *name*)

Return system configuration information relevant to a named file. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX.1, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given in the `pathconf_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If *name* is a string and is not known, `ValueError` is raised. If a specific value for *name* is not supported by the host system, even if it is included in `pathconf_names`, an `OSError` is raised with `errno.EINVAL` for the error number.

This function can support specifying a file descriptor.

Availability: Unix.

> *Changed in version 3.6:* Accepts a path-like object.

os.**pathconf_names**

Dictionary mapping names accepted by `pathconf()` and `fpathconf()` to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

os.**readlink**(*path*, *, *dir_fd=None*)

Return a string representing the path to which the symbolic link points. The result may be either an absolute or relative pathname; if it is relative, it may be converted to an absolute pathname using `os.path.join(os.path.dirname(path), result)`.

If the *path* is a string object (directly or indirectly through a `PathLike` interface), the result will also be a string object, and the call may raise a UnicodeDecodeError. If the *path* is a bytes object (direct or indirectly), the result will be a bytes object.

This function can also support paths relative to directory descriptors.

When trying to resolve a path that may contain links, use `realpath()` to properly handle recursion and platform differences.

Availability: Unix, Windows.

> *Changed in version 3.2:* Added support for Windows 6.0 (Vista) symbolic links.

> *Changed in version 3.3:* Added the *dir_fd* parameter.

> *Changed in version 3.6:* Accepts a path-like object on Unix.

> *Changed in version 3.8:* Accepts a path-like object and a bytes object on Windows.
>
> Added support for directory junctions, and changed to return the substitution path (which typically includes `\\?\` prefix) rather than the optional "print name" field that was previously returned.

os.**remove**(*path*, *, *dir_fd=None*)

Remove (delete) the file *path*. If *path* is a directory, an `OSError` is raised. Use `rmdir()` to remove directories. If the file does not exist, a `FileNotFoundError` is raised.

This function can support paths relative to directory descriptors.

On Windows, attempting to remove a file that is in use causes an exception to be raised; on Unix, the directory entry is removed but the storage allocated to the file is not made available until the original file is no longer in use.

This function is semantically identical to `unlink()`.

Raises an auditing event `os.remove` with arguments `path`, `dir_fd`.

> *Changed in version 3.3:* Added the *dir_fd* parameter.

> *Changed in version 3.6:* Accepts a path-like object.

os.**removedirs**(*name*)

Remove directories recursively. Works like `rmdir()` except that, if the leaf directory is successfully removed, `removedirs()` tries to successively remove every parent directory mentioned in *path* until an error is raised (which is ignored, because it generally means that a parent directory is not empty). For example, `os.removedirs('foo/bar/baz')` will first remove the directory `'foo/bar/baz'`, and then remove `'foo/bar'` and `'foo'` if they are empty. Raises `OSError` if the leaf directory could not be successfully removed.

Raises an auditing event `os.remove` with arguments `path`, `dir_fd`.

> *Changed in version 3.6:* Accepts a path-like object.

os.**rename**(*src*, *dst*, *, *src_dir_fd=None*, *dst_dir_fd=None*)

Rename the file or directory *src* to *dst*. If *dst* exists, the operation will fail with an `OSError` subclass in a number of cases:

On Windows, if *dst* exists a `FileExistsError` is always raised. The operation may fail if *src* and *dst* are on different filesystems. Use `shutil.move()` to support moves to a different filesystem.

On Unix, if *src* is a file and *dst* is a directory or vice-versa, an `IsADirectoryError` or a `NotADirectoryError` will be raised respectively. If both are directories and *dst* is empty, *dst* will be silently replaced. If *dst* is a non-empty directory, an `OSError` is raised. If both are files, *dst* will be replaced silently if the user has permission. The operation may fail on some Unix flavors if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

This function can support specifying *src_dir_fd* and/or *dst_dir_fd* to supply paths relative to directory descriptors.

If you want cross-platform overwriting of the destination, use `replace()`.

Raises an auditing event `os.rename` with arguments `src`, `dst`, `src_dir_fd`, `dst_dir_fd`.

> *Changed in version 3.3:* Added the *src_dir_fd* and *dst_dir_fd* parameters.

> *Changed in version 3.6:* Accepts a path-like object for *src* and *dst*.

`os.`**`renames`**`(`*`old, new`*`)`

Recursive directory or file renaming function. Works like `rename()`, except creation of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned away using `removedirs()`.

> **Note:** This function can fail with the new directory structure made if you lack permissions needed to remove the leaf directory or file.

Raises an auditing event `os.rename` with arguments `src`, `dst`, `src_dir_fd`, `dst_dir_fd`.

> *Changed in version 3.6:* Accepts a path-like object for *old* and *new*.

`os.`**`replace`**`(`*`src, dst, *, src_dir_fd=None, dst_dir_fd=None`*`)`

Rename the file or directory *src* to *dst*. If *dst* is a non-empty directory, `OSError` will be raised. If *dst* exists and is a file, it will be replaced silently if the user has permission. The operation may fail if *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operation (this is a POSIX requirement).

This function can support specifying *src_dir_fd* and/or *dst_dir_fd* to supply paths relative to directory descriptors.

Raises an auditing event `os.rename` with arguments `src`, `dst`, `src_dir_fd`, `dst_dir_fd`.

> *Added in version 3.3.*

> *Changed in version 3.6:* Accepts a path-like object for *src* and *dst*.

`os.`**`rmdir`**`(`*`path, *, dir_fd=None`*`)`

Remove (delete) the directory *path*. If the directory does not exist or is not empty, a `FileNotFoundError` or an `OSError` is raised respectively. In order to remove whole directory trees, `shutil.rmtree()` can be used.

This function can support paths relative to directory descriptors.

Raises an auditing event `os.rmdir` with arguments `path`, `dir_fd`.

> *Changed in version 3.3:* Added the *dir_fd* parameter.

> *Changed in version 3.6:* Accepts a path-like object.

os.**scandir**(*path='.'*)

Return an iterator of `os.DirEntry` objects corresponding to the entries in the directory given by *path*. The entries are yielded in arbitrary order, and the special entries `'.'` and `'..'` are not included. If a file is removed from or added to the directory after creating the iterator, whether an entry for that file be included is unspecified.

Using `scandir()` instead of `listdir()` can significantly increase the performance of code that also needs file type or file attribute information, because `os.DirEntry` objects expose this information if the operating system provides it when scanning a directory. All `os.DirEntry` methods may perform a system call, but `is_dir()` and `is_file()` usually only require a system call for symbolic links; `os.DirEntry.stat()` always requires a system call on Unix but only requires one for symbolic links on Windows.

path may be a path-like object. If *path* is of type `bytes` (directly or indirectly through the `PathLike` interface), the type of the `name` and `path` attributes of each `os.DirEntry` will be `bytes`; in all other circumstances, they will be of type `str`.

This function can also support specifying a file descriptor; the file descriptor must refer to a directory.

Raises an auditing event `os.scandir` with argument `path`.

The `scandir()` iterator supports the context manager protocol and has the following method:

scandir.**close**()

Close the iterator and free acquired resources.

This is called automatically when the iterator is exhausted or garbage collected, or when an error happens during iterating. However it is advisable to call it explicitly or use the `with` statement.

> *Added in version 3.6.*

The following example shows a simple use of `scandir()` to display all the files (excluding directories) in the given *path* that don't start with `'.'`. The `entry.is_file()` call will generally not make an additional system call:

```
with os.scandir(path) as it:
    for entry in it:
        if not entry.name.startswith('.') and entry.is_file():
            print(entry.name)
```

Note: On Unix-based systems, `scandir()` uses the system's opendir() and readdir() functions. On Windows, it uses the Win32 FindFirstFileW and FindNextFileW functions.

> *Added in version 3.5.*

> *Changed in version 3.6:* Added support for the context manager protocol and the `close()` method. If a `scandir()` iterator is neither exhausted nor explicitly closed a `ResourceWarning` will be emitted in its destructor.
>
> The function accepts a path-like object.

> *Changed in version 3.7:* Added support for file descriptors on Unix.

class os.**DirEntry**

Object yielded by `scandir()` to expose the file path and other file attributes of a directory entry.

`scandir()` will provide as much of this information as possible without making additional system calls. When a `stat()` or `lstat()` system call is made, the `os.DirEntry` object will cache the result.

`os.DirEntry` instances are not intended to be stored in long-lived data structures; if you know the file metadata has changed or if a long time has elapsed since calling `scandir()`, call `os.stat(entry.path)` to fetch up-to-date information.

Because the `os.DirEntry` methods can make operating system calls, they may also raise `OSError`. If you need very fine-grained control over errors, you can catch `OSError` when calling one of the `os.DirEntry` methods and handle as appropriate.

To be directly usable as a path-like object, `os.DirEntry` implements the `PathLike` interface.

Attributes and methods on a `os.DirEntry` instance are as follows:

name

The entry's base filename, relative to the `scandir()` *path* argument.

The `name` attribute will be `bytes` if the `scandir()` *path* argument is of type `bytes` and `str` otherwise. Use `fsdecode()` to decode byte filenames.

path

The entry's full path name: equivalent to `os.path.join(scandir_path, entry.name)` where *scandir_path* is the `scandir()` *path* argument. The path is only absolute if the `scandir()` *path* argument was absolute. If the `scandir()` *path* argument was a file descriptor, the `path` attribute is the same as the `name` attribute.

The `path` attribute will be `bytes` if the `scandir()` *path* argument is of type `bytes` and `str` otherwise. Use `fsdecode()` to decode byte filenames.

inode()

Return the inode number of the entry.

The result is cached on the `os.DirEntry` object. Use `os.stat(entry.path, follow_symlinks=False).st_ino` to fetch up-to-date information.

On the first, uncached call, a system call is required on Windows but not on Unix.

is_dir(**, follow_symlinks=True*)

Return `True` if this entry is a directory or a symbolic link pointing to a directory; return `False` if the entry is or points to any other kind of file, or if it doesn't exist anymore.

If *follow_symlinks* is `False`, return `True` only if this entry is a directory (without following symlinks); return `False` if the entry is any other kind of file or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object, with a separate cache for *follow_symlinks* `True` and `False`. Call `os.stat()` along with `stat.S_ISDIR()` to fetch up-to-date information.

On the first, uncached call, no system call is required in most cases. Specifically, for non-symlinks, neither Windows or Unix require a system call, except on certain Unix file systems, such as network file systems, that return `dirent.d_type == DT_UNKNOWN`. If the entry is a symlink, a system call will be required to follow the symlink unless *follow_symlinks* is `False`.

This method can raise `OSError`, such as `PermissionError`, but `FileNotFoundError` is caught and not raised.

is_file(**, follow_symlinks=True*)

Return `True` if this entry is a file or a symbolic link pointing to a file; return `False` if the entry is or points to a directory or other non-file entry, or if it doesn't exist anymore.

If *follow_symlinks* is `False`, return `True` only if this entry is a file (without following symlinks); return `False` if the entry is a directory or other non-file entry, or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object. Caching, system calls made, and exceptions raised are as per `is_dir()`.

is_symlink()

Return `True` if this entry is a symbolic link (even if broken); return `False` if the entry points to a directory or any kind of file, or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object. Call `os.path.islink()` to fetch up-to-date information.

On the first, uncached call, no system call is required in most cases. Specifically, neither Windows or Unix require a system call, except on certain Unix file systems, such as network file systems, that return `dirent.d_type == DT_UNKNOWN`.

This method can raise `OSError`, such as `PermissionError`, but `FileNotFoundError` is caught and not raised.

is_junction()

Return `True` if this entry is a junction (even if broken); return `False` if the entry points to a regular directory, any kind of file, a symlink, or if it doesn't exist anymore.

The result is cached on the `os.DirEntry` object. Call `os.path.isjunction()` to fetch up-to-date information.

> *Added in version 3.12.*

stat(**, follow_symlinks=True*)

Return a `stat_result` object for this entry. This method follows symbolic links by default; to stat a symbolic link add the `follow_symlinks=False` argument.

On Unix, this method always requires a system call. On Windows, it only requires a system call if *follow_symlinks* is `True` and the entry is a reparse point (for example, a symbolic link or directory junction).

On Windows, the `st_ino`, `st_dev` and `st_nlink` attributes of the `stat_result` are always set to zero. Call `os.stat()` to get these attributes.

The result is cached on the `os.DirEntry` object, with a separate cache for *follow_symlinks* `True` and `False`. Call `os.stat()` to fetch up-to-date information.

Note that there is a nice correspondence between several attributes and methods of `os.DirEntry` and of `pathlib.Path`. In particular, the `name` attribute has the same meaning, as do the `is_dir()`, `is_file()`, `is_symlink()`, `is_junction()`, and `stat()` methods.

> *Added in version 3.5.*

> *Changed in version 3.6:* Added support for the `PathLike` interface. Added support for `bytes` paths on Windows.

> *Changed in version 3.12:* The `st_ctime` attribute of a stat result is deprecated on Windows. The file creation time is properly available as `st_birthtime`, and in the future `st_ctime` may be changed to return zero or the metadata change time, if available.

os.**stat**(*path*, ***, *dir_fd=None*, *follow_symlinks=True*)

Get the status of a file or a file descriptor. Perform the equivalent of a `stat()` system call on the given path. *path* may be specified as either a string or bytes – directly or indirectly through the `PathLike` interface – or as an open file descriptor. Return a `stat_result` object.

This function normally follows symlinks; to stat a symlink add the argument `follow_symlinks=False`, or use `lstat()`.

This function can support specifying a file descriptor and not following symlinks.

On Windows, passing `follow_symlinks=False` will disable following all name-surrogate reparse points, which includes symlinks and directory junctions. Other types of reparse points that do not resemble links or that the operating system is unable to follow will be opened directly. When following a chain of multiple links, this may result in the original link being returned instead of the non-link that prevented full traversal. To obtain stat results for the final path in this case, use the `os.path.realpath()` function to resolve the path name as far as possible and call `lstat()` on the result. This does not apply to dangling symlinks or junction points, which will raise the usual exceptions.

Example:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
os.stat_result(st_mode=33188, st_ino=7876932, st_dev=234881026,
st_nlink=1, st_uid=501, st_gid=501, st_size=264, st_atime=1297230295,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```

See also: `fstat()` and `lstat()` functions.

> *Changed in version 3.3:* Added the *dir_fd* and *follow_symlinks* parameters, specifying a file descriptor instead of a path.

> *Changed in version 3.6:* Accepts a path-like object.

> *Changed in version 3.8:* On Windows, all reparse points that can be resolved by the operating system are now followed, and passing `follow_symlinks=False` disables following all name surrogate reparse points. If the operating system reaches a reparse point that it is not able to follow, *stat* now returns the information for the original path as if `follow_symlinks=False` had been specified instead of raising an error.

class os.**stat_result**

Object whose attributes correspond roughly to the members of the `stat` structure. It is used for the result of `os.stat()`, `os.fstat()` and `os.lstat()`.

Attributes:

st_mode

File mode: file type and file mode bits (permissions).

st_ino

Platform dependent, but if non-zero, uniquely identifies the file for a given value of `st_dev`. Typically:

- the inode number on Unix,
- the file index on Windows

st_dev

Identifier of the device on which this file resides.

st_nlink

Number of hard links.

st_uid

User identifier of the file owner.

st_gid

Group identifier of the file owner.

st_size

Size of the file in bytes, if it is a regular file or a symbolic link. The size of a symbolic link is the length of the pathname it contains, without a terminating null byte.

Timestamps:

st_atime

Time of most recent access expressed in seconds.

st_mtime

Time of most recent content modification expressed in seconds.

st_ctime

Time of most recent metadata change expressed in seconds.

> *Changed in version 3.12:* `st_ctime` is deprecated on Windows. Use `st_birthtime` for the file creation time. In the future, `st_ctime` will contain the time of the most recent metadata change, as for other platforms.

st_atime_ns

Time of most recent access expressed in nanoseconds as an integer.

> *Added in version 3.3.*

st_mtime_ns

Time of most recent content modification expressed in nanoseconds as an integer.

> *Added in version 3.3.*

st_ctime_ns

Time of most recent metadata change expressed in nanoseconds as an integer.

> *Added in version 3.3.*

> *Changed in version 3.12:* `st_ctime_ns` is deprecated on Windows. Use `st_birthtime_ns` for the file creation time. In the future, `st_ctime` will contain the time of the most recent metadata change, as for other platforms.

st_birthtime

Time of file creation expressed in seconds. This attribute is not always available, and may raise `AttributeError`.

> *Changed in version 3.12:* `st_birthtime` is now available on Windows.

st_birthtime_ns

Time of file creation expressed in nanoseconds as an integer. This attribute is not always available, and may raise `AttributeError`.

> *Added in version 3.12.*

Note: The exact meaning and resolution of the `st_atime`, `st_mtime`, `st_ctime` and `st_birthtime` attributes depend on the operating system and the file system. For example, on Windows systems using the FAT32 file systems, `st_mtime` has 2-second resolution, and `st_atime` has only 1-day resolution. See your operating system documentation for details.

Similarly, although `st_atime_ns`, `st_mtime_ns`, `st_ctime_ns` and `st_birthtime_ns` are always expressed in nanoseconds, many systems do not provide nanosecond precision. On systems that do provide nanosecond precision, the floating-point object used to store `st_atime`, `st_mtime`, `st_ctime` and `st_birthtime` cannot preserve all of it, and as such will be slightly inexact. If you need the exact timestamps you should always use `st_atime_ns`, `st_mtime_ns`, `st_ctime_ns` and `st_birthtime_ns`.

On some Unix systems (such as Linux), the following attributes may also be available:

st_blocks

Number of 512-byte blocks allocated for file. This may be smaller than `st_size`/512 when the file has holes.

st_blksize

"Preferred" blocksize for efficient file system I/O. Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.

st_rdev

Type of device if an inode device.

st_flags

User defined flags for file.

On other Unix systems (such as FreeBSD), the following attributes may be available (but may be only filled out if root tries to use them):

st_gen

File generation number.

On Solaris and derivatives, the following attributes may also be available:

st_fstype

String that uniquely identifies the type of the filesystem that contains the file.

On macOS systems, the following attributes may also be available:

st_rsize

Real size of the file.

st_creator

Creator of the file.

st_type

File type.

On Windows systems, the following attributes are also available:

st_file_attributes

Windows file attributes: `dwFileAttributes` member of the `BY_HANDLE_FILE_INFORMATION` structure returned by `GetFileInformationByHandle()`. See the `FILE_ATTRIBUTE_*` `<stat.FILE_ATTRIBUTE_ARCHIVE>` constants in the `stat` module.

> *Added in version 3.5.*

st_reparse_tag

When `st_file_attributes` has the `FILE_ATTRIBUTE_REPARSE_POINT` set, this field contains the tag identifying the type of reparse point. See the `IO_REPARSE_TAG_*` constants in the `stat` module.

The standard module `stat` defines functions and constants that are useful for extracting information from a `stat` structure. (On Windows, some items are filled with dummy values.)

For backward compatibility, a `stat_result` instance is also accessible as a tuple of at least 10 integers giving the most important (and portable) members of the `stat` structure, in the order `st_mode`, `st_ino`, `st_dev`, `st_nlink`, `st_uid`, `st_gid`, `st_size`, `st_atime`, `st_mtime`, `st_ctime`. More items may be added at the end by some implementations. For compatibility with older Python versions, accessing `stat_result` as a tuple always returns integers.

> *Changed in version 3.5:* Windows now returns the file index as `st_ino` when available.

> *Changed in version 3.7:* Added the `st_fstype` member to Solaris/derivatives.

> *Changed in version 3.8:* Added the `st_reparse_tag` member on Windows.

> *Changed in version 3.8:* On Windows, the `st_mode` member now identifies special files as `S_IFCHR`, `S_IFIFO` or `S_IFBLK` as appropriate.

> *Changed in version 3.12:* On Windows, `st_ctime` is deprecated. Eventually, it will contain the last metadata change time, for consistency with other platforms, but for now still contains creation time. Use `st_birthtime` for the creation time.
>
> On Windows, `st_ino` may now be up to 128 bits, depending on the file system. Previously it would not be above 64 bits, and larger file identifiers would be arbitrarily packed.
>
> On Windows, `st_rdev` no longer returns a value. Previously it would contain the same as `st_dev`, which was incorrect.
>
> Added the `st_birthtime` member on Windows.

`os.`**statvfs**(*path*)

Perform a `statvfs()` system call on the given path. The return value is an object whose attributes describe the filesystem on the given path, and correspond to the members of the `statvfs` structure, namely: `f_bsize`, `f_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`, `f_fsid`.

Two module-level constants are defined for the `f_flag` attribute's bit-flags: if `ST_RDONLY` is set, the filesystem is mounted read-only, and if `ST_NOSUID` is set, the semantics of setuid/setgid bits are disabled

or not supported.

Additional module-level constants are defined for GNU/glibc based systems. These are `ST_NODEV` (disallow access to device special files), `ST_NOEXEC` (disallow program execution), `ST_SYNCHRONOUS` (writes are synced at once), `ST_MANDLOCK` (allow mandatory locks on an FS), `ST_WRITE` (write on file/directory/symlink), `ST_APPEND` (append-only file), `ST_IMMUTABLE` (immutable file), `ST_NOATIME` (do not update access times), `ST_NODIRATIME` (do not update directory access times), `ST_RELATIME` (update atime relative to mtime/ctime).

This function can support specifying a file descriptor.

Availability: Unix.

> *Changed in version 3.2:* The `ST_RDONLY` and `ST_NOSUID` constants were added.

> *Changed in version 3.3:* Added support for specifying *path* as an open file descriptor.

> *Changed in version 3.4:* The `ST_NODEV`, `ST_NOEXEC`, `ST_SYNCHRONOUS`, `ST_MANDLOCK`, `ST_WRITE`, `ST_APPEND`, `ST_IMMUTABLE`, `ST_NOATIME`, `ST_NODIRATIME`, and `ST_RELATIME` constants were added.

> *Changed in version 3.6:* Accepts a path-like object.

> *Changed in version 3.7:* Added the `f_fsid` attribute.

os.supports_dir_fd

A `set` object indicating which functions in the `os` module accept an open file descriptor for their *dir_fd* parameter. Different platforms provide different features, and the underlying functionality Python uses to implement the *dir_fd* parameter is not available on all platforms Python supports. For consistency's sake, functions that may support *dir_fd* always allow specifying the parameter, but will throw an exception if the functionality is used when it's not locally available. (Specifying `None` for *dir_fd* is always supported on all platforms.)

To check whether a particular function accepts an open file descriptor for its *dir_fd* parameter, use the `in` operator on `supports_dir_fd`. As an example, this expression evaluates to `True` if `os.stat()` accepts open file descriptors for *dir_fd* on the local platform:

```
os.stat in os.supports_dir_fd
```

Currently *dir_fd* parameters only work on Unix platforms; none of them work on Windows.

> *Added in version 3.3.*

os.supports_effective_ids

A `set` object indicating whether `os.access()` permits specifying `True` for its *effective_ids* parameter on the local platform. (Specifying `False` for *effective_ids* is always supported on all platforms.) If the local platform supports it, the collection will contain `os.access()`; otherwise it will be empty.

This expression evaluates to `True` if `os.access()` supports `effective_ids=True` on the local platform:

```
os.access in os.supports_effective_ids
```

Currently *effective_ids* is only supported on Unix platforms; it does not work on Windows.

> *Added in version 3.3.*

os.**supports_fd**

A set object indicating which functions in the os module permit specifying their *path* parameter as an open file descriptor on the local platform. Different platforms provide different features, and the underlying functionality Python uses to accept open file descriptors as *path* arguments is not available on all platforms Python supports.

To determine whether a particular function permits specifying an open file descriptor for its *path* parameter, use the `in` operator on `supports_fd`. As an example, this expression evaluates to `True` if `os.chdir()` accepts open file descriptors for *path* on your local platform:

```
os.chdir in os.supports_fd
```

> *Added in version 3.3.*

os.**supports_follow_symlinks**

A set object indicating which functions in the os module accept `False` for their *follow_symlinks* parameter on the local platform. Different platforms provide different features, and the underlying functionality Python uses to implement *follow_symlinks* is not available on all platforms Python supports. For consistency's sake, functions that may support *follow_symlinks* always allow specifying the parameter, but will throw an exception if the functionality is used when it's not locally available. (Specifying `True` for *follow_symlinks* is always supported on all platforms.)

To check whether a particular function accepts `False` for its *follow_symlinks* parameter, use the `in` operator on `supports_follow_symlinks`. As an example, this expression evaluates to `True` if you may specify `follow_symlinks=False` when calling `os.stat()` on the local platform:

```
os.stat in os.supports_follow_symlinks
```

> *Added in version 3.3.*

os.**symlink**(*src*, *dst*, *target_is_directory=False*, ***, *dir_fd=None*)

Create a symbolic link pointing to *src* named *dst*.

On Windows, a symlink represents either a file or a directory, and does not morph to the target dynamically. If the target is present, the type of the symlink will be created to match. Otherwise, the symlink will be created as a directory if *target_is_directory* is `True` or a file symlink (the default) otherwise. On non-Windows platforms, *target_is_directory* is ignored.

This function can support paths relative to directory descriptors.

> **Note:** On newer versions of Windows 10, unprivileged accounts can create symlinks if Developer Mode is enabled. When Developer Mode is not available/enabled, the *SeCreateSymbolicLinkPrivilege* privilege is required, or the process must be run as an administrator.
>
> `OSError` is raised when the function is called by an unprivileged user.

Raises an auditing event `os.symlink` with arguments `src`, `dst`, `dir_fd`.

Availability: Unix, Windows.

The function is limited on WASI, see WebAssembly platforms for more information.

> *Changed in version 3.2:* Added support for Windows 6.0 (Vista) symbolic links.

> *Changed in version 3.3:* Added the *dir_fd* parameter, and now allow *target_is_directory* on non-Windows platforms.

> *Changed in version 3.6:* Accepts a path-like object for *src* and *dst*.

> *Changed in version 3.8:* Added support for unelevated symlinks on Windows with Developer Mode.

os.**sync**()

Force write of everything to disk.

Availability: Unix.

> *Added in version 3.3.*

os.**truncate**(*path*, *length*)

Truncate the file corresponding to *path*, so that it is at most *length* bytes in size.

This function can support specifying a file descriptor.

Raises an auditing event `os.truncate` with arguments `path`, `length`.

Availability: Unix, Windows.

> *Added in version 3.3.*

> *Changed in version 3.5:* Added support for Windows

> *Changed in version 3.6:* Accepts a path-like object.

os.**unlink**(*path*, ***, *dir_fd=None*)

Remove (delete) the file *path*. This function is semantically identical to `remove()`; the `unlink` name is its traditional Unix name. Please see the documentation for `remove()` for further information.

Raises an auditing event `os.remove` with arguments `path`, `dir_fd`.

> *Changed in version 3.3:* Added the *dir_fd* parameter.

> *Changed in version 3.6:* Accepts a path-like object.

os.**utime**(*path*, *times=None*, ***, [*ns*,]*dir_fd=None*, *follow_symlinks=True*)

Set the access and modified times of the file specified by *path*.

`utime()` takes two optional parameters, *times* and *ns*. These specify the times set on *path* and are used as follows:

- If *ns* is specified, it must be a 2-tuple of the form `(atime_ns, mtime_ns)` where each member is an int expressing nanoseconds.
- If *times* is not `None`, it must be a 2-tuple of the form `(atime, mtime)` where each member is an int or float expressing seconds.

- If *times* is `None` and *ns* is unspecified, this is equivalent to specifying `ns=(atime_ns, mtime_ns)` where both times are the current time.

It is an error to specify tuples for both *times* and *ns*.

Note that the exact times you set here may not be returned by a subsequent `stat()` call, depending on the resolution with which your operating system records access and modification times; see `stat()`. The best way to preserve exact times is to use the *st_atime_ns* and *st_mtime_ns* fields from the `os.stat()` result object with the *ns* parameter to `utime()`.

This function can support specifying a file descriptor, paths relative to directory descriptors and not following symlinks.

Raises an auditing event `os.utime` with arguments `path`, `times`, `ns`, `dir_fd`.

> *Changed in version 3.3:* Added support for specifying *path* as an open file descriptor, and the *dir_fd*, *follow_symlinks*, and *ns* parameters.

> *Changed in version 3.6:* Accepts a path-like object.

`os.`**`walk`**`(`*`top, topdown=True, onerror=None, followlinks=False`*`)`

Generate the file names in a directory tree by walking the tree either top-down or bottom-up. For each directory in the tree rooted at directory *top* (including *top* itself), it yields a 3-tuple `(dirpath, dirnames, filenames)`.

dirpath is a string, the path to the directory. *dirnames* is a list of the names of the subdirectories in *dirpath* (including symlinks to directories, and excluding `'.'` and `'..'`). *filenames* is a list of the names of the non-directory files in *dirpath*. Note that the names in the lists contain no path components. To get a full path (which begins with *top*) to a file or directory in *dirpath*, do `os.path.join(dirpath, name)`. Whether or not the lists are sorted depends on the file system. If a file is removed from or added to the *dirpath* directory during generating the lists, whether a name for that file be included is unspecified.

If optional argument *topdown* is `True` or not specified, the triple for a directory is generated before the triples for any of its subdirectories (directories are generated top-down). If *topdown* is `False`, the triple for a directory is generated after the triples for all of its subdirectories (directories are generated bottom-up). No matter the value of *topdown*, the list of subdirectories is retrieved before the tuples for the directory and its subdirectories are generated.

When *topdown* is `True`, the caller can modify the *dirnames* list in-place (perhaps using `del` or slice assignment), and `walk()` will only recurse into the subdirectories whose names remain in *dirnames*; this can be used to prune the search, impose a specific order of visiting, or even to inform `walk()` about directories the caller creates or renames before it resumes `walk()` again. Modifying *dirnames* when *topdown* is `False` has no effect on the behavior of the walk, because in bottom-up mode the directories in *dirnames* are generated before *dirpath* itself is generated.

By default, errors from the `scandir()` call are ignored. If optional argument *onerror* is specified, it should be a function; it will be called with one argument, an `OSError` instance. It can report the error to continue with the walk, or raise the exception to abort the walk. Note that the filename is available as the `filename` attribute of the exception object.

By default, `walk()` will not walk down into symbolic links that resolve to directories. Set *followlinks* to `True` to visit directories pointed to by symlinks, on systems that support them.

> **Note:** Be aware that setting *followlinks* to `True` can lead to infinite recursion if a link points to a parent directory of itself. `walk()` does not keep track of the directories it visited already.

> **Note:** If you pass a relative pathname, don't change the current working directory between resumptions of `walk()`. `walk()` never changes the current directory, and assumes that its caller doesn't either.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn't look under any `__pycache__` subdirectory:

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/xml'):
    print(root, "consumes", end=" ")
    print(sum(getsize(join(root, name)) for name in files), end=" ")
    print("bytes in", len(files), "non-directory files")
    if '__pycache__' in dirs:
        dirs.remove('__pycache__')  # don't visit __pycache__ directories
```

In the next example (simple implementation of `shutil.rmtree()`), walking the tree bottom-up is essential, `rmdir()` doesn't allow deleting a directory before the directory is empty:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
os.rmdir(top)
```

Raises an auditing event `os.walk` with arguments `top`, `topdown`, `onerror`, `followlinks`.

> *Changed in version 3.5:* This function now calls `os.scandir()` instead of `os.listdir()`, making it faster by reducing the number of calls to `os.stat()`.

> *Changed in version 3.6:* Accepts a path-like object.

os.**fwalk**(*top='.', topdown=True, onerror=None, *, follow_symlinks=False, dir_fd=None*)

This behaves exactly like `walk()`, except that it yields a 4-tuple `(dirpath, dirnames, filenames, dirfd)`, and it supports `dir_fd`.

dirpath, *dirnames* and *filenames* are identical to `walk()` output, and *dirfd* is a file descriptor referring to the directory *dirpath*.

This function always supports paths relative to directory descriptors and not following symlinks. Note however that, unlike other functions, the `fwalk()` default value for *follow_symlinks* is `False`.

Note: Since `fwalk()` yields file descriptors, those are only valid until the next iteration step, so you should duplicate them (e.g. with `dup()`) if you want to keep them longer.

This example displays the number of bytes taken by non-directory files in each directory under the starting directory, except that it doesn't look under any `__pycache__` subdirectory:

```
import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/xml'):
    print(root, "consumes", end=" ")
    print(sum([os.stat(name, dir_fd=rootfd).st_size for name in files]),
          end=" ")
    print("bytes in", len(files), "non-directory files")
    if '__pycache__' in dirs:
        dirs.remove('__pycache__')  # don't visit __pycache__ directories
```

In the next example, walking the tree bottom-up is essential: `rmdir()` doesn't allow deleting a directory before the directory is empty:

```
# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files, rootfd in os.fwalk(top, topdown=False):
    for name in files:
        os.unlink(name, dir_fd=rootfd)
    for name in dirs:
        os.rmdir(name, dir_fd=rootfd)
```

Raises an auditing event `os.fwalk` with arguments `top`, `topdown`, `onerror`, `follow_symlinks`, `dir_fd`.

Availability: Unix.

> *Added in version 3.3.*

> *Changed in version 3.6:* Accepts a path-like object.

> *Changed in version 3.7:* Added support for `bytes` paths.

os.**memfd_create**(*name*[, *flags=os.MFD_CLOEXEC*])

Create an anonymous file and return a file descriptor that refers to it. *flags* must be one of the `os.MFD_*` constants available on the system (or a bitwise ORed combination of them). By default, the new file descriptor is non-inheritable.

The name supplied in *name* is used as a filename and will be displayed as the target of the corresponding symbolic link in the directory `/proc/self/fd/`. The displayed name is always prefixed with `memfd:` and serves only for debugging purposes. Names do not affect the behavior of the file descriptor, and as such multiple files can have the same name without any side effects.

Availability: Linux >= 3.17 with glibc >= 2.27.

> *Added in version 3.8.*

os.**MFD_CLOEXEC**
os.**MFD_ALLOW_SEALING**

os.**MFD_HUGETLB**
os.**MFD_HUGE_SHIFT**
os.**MFD_HUGE_MASK**
os.**MFD_HUGE_64KB**
os.**MFD_HUGE_512KB**
os.**MFD_HUGE_1MB**
os.**MFD_HUGE_2MB**
os.**MFD_HUGE_8MB**
os.**MFD_HUGE_16MB**
os.**MFD_HUGE_32MB**
os.**MFD_HUGE_256MB**
os.**MFD_HUGE_512MB**
os.**MFD_HUGE_1GB**
os.**MFD_HUGE_2GB**
os.**MFD_HUGE_16GB**

These flags can be passed to `memfd_create()`.

Availability: Linux >= 3.17 with glibc >= 2.27

The `MFD_HUGE*` flags are only available since Linux 4.14.

> *Added in version 3.8.*

os.**eventfd**(*initval*[, *flags=os.EFD_CLOEXEC*])

Create and return an event file descriptor. The file descriptors supports raw `read()` and `write()` with a buffer size of 8, `select()`, `poll()` and similar. See man page *eventfd(2)* for more information. By default, the new file descriptor is non-inheritable.

initval is the initial value of the event counter. The initial value must be a 32 bit unsigned integer. Please note that the initial value is limited to a 32 bit unsigned int although the event counter is an unsigned 64 bit integer with a maximum value of 2^{64}-2.

flags can be constructed from `EFD_CLOEXEC`, `EFD_NONBLOCK`, and `EFD_SEMAPHORE`.

If `EFD_SEMAPHORE` is specified and the event counter is non-zero, `eventfd_read()` returns 1 and decrements the counter by one.

If `EFD_SEMAPHORE` is not specified and the event counter is non-zero, `eventfd_read()` returns the current event counter value and resets the counter to zero.

If the event counter is zero and `EFD_NONBLOCK` is not specified, `eventfd_read()` blocks.

`eventfd_write()` increments the event counter. Write blocks if the write operation would increment the counter to a value larger than 2^{64}-2.

Example:

```
import os

# semaphore with start value '1'
fd = os.eventfd(1, os.EFD_SEMAPHORE | os.EFC_CLOEXEC)
try:
    # acquire semaphore
    v = os.eventfd_read(fd)
    try:
```

```
        do_work()
    finally:
        # release semaphore
        os.eventfd_write(fd, v)
finally:
    os.close(fd)
```

Availability: Linux >= 2.6.27 with glibc >= 2.8

> *Added in version 3.10.*

os.**eventfd_read**(*fd*)

Read value from an `eventfd()` file descriptor and return a 64 bit unsigned int. The function does not verify that *fd* is an `eventfd()`.

Availability: Linux >= 2.6.27

> *Added in version 3.10.*

os.**eventfd_write**(*fd*, *value*)

Add value to an `eventfd()` file descriptor. *value* must be a 64 bit unsigned int. The function does not verify that *fd* is an `eventfd()`.

Availability: Linux >= 2.6.27

> *Added in version 3.10.*

os.**EFD_CLOEXEC**

Set close-on-exec flag for new `eventfd()` file descriptor.

Availability: Linux >= 2.6.27

> *Added in version 3.10.*

os.**EFD_NONBLOCK**

Set `O_NONBLOCK` status flag for new `eventfd()` file descriptor.

Availability: Linux >= 2.6.27

> *Added in version 3.10.*

os.**EFD_SEMAPHORE**

Provide semaphore-like semantics for reads from an `eventfd()` file descriptor. On read the internal counter is decremented by one.

Availability: Linux >= 2.6.30

> *Added in version 3.10.*

Timer File Descriptors

> *Added in version 3.13.*

These functions provide support for Linux's *timer file descriptor* API. Naturally, they are all only available on Linux.

os.**timerfd_create**(*clockid*, /, *, *flags=0*)

Create and return a timer file descriptor (*timerfd*).

The file descriptor returned by `timerfd_create()` supports:

- `read()`
- `select()`
- `poll()`

The file descriptor's `read()` method can be called with a buffer size of 8. If the timer has already expired one or more times, `read()` returns the number of expirations with the host's endianness, which may be converted to an `int` by `int.from_bytes(x, byteorder=sys.byteorder)`.

`select()` and `poll()` can be used to wait until timer expires and the file descriptor is readable.

clockid must be a valid clock ID, as defined in the `time` module:

- `time.CLOCK_REALTIME`
- `time.CLOCK_MONOTONIC`
- `time.CLOCK_BOOTTIME` (Since Linux 3.15 for timerfd_create)

If *clockid* is `time.CLOCK_REALTIME`, a settable system-wide real-time clock is used. If system clock is changed, timer setting need to be updated. To cancel timer when system clock is changed, see `TFD_TIMER_CANCEL_ON_SET`.

If *clockid* is `time.CLOCK_MONOTONIC`, a non-settable monotonically increasing clock is used. Even if the system clock is changed, the timer setting will not be affected.

If *clockid* is `time.CLOCK_BOOTTIME`, same as `time.CLOCK_MONOTONIC` except it includes any time that the system is suspended.

The file descriptor's behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the `|` operator):

- `TFD_NONBLOCK`
- `TFD_CLOEXEC`

If `TFD_NONBLOCK` is not set as a flag, `read()` blocks until the timer expires. If it is set as a flag, `read()` doesn't block, but If there hasn't been an expiration since the last call to read, `read()` raises `OSError` with `errno` is set to `errno.EAGAIN`.

`TFD_CLOEXEC` is always set by Python automatically.

The file descriptor must be closed with `os.close()` when it is no longer needed, or else the file descriptor will be leaked.

See also: The *timerfd_create(2)* man page.

Availability: Linux >= 2.6.27 with glibc >= 2.8

> *Added in version 3.13.*

os.**timerfd_settime**(*fd*, *, *, *flags=flags*, *initial=0.0*, *interval=0.0*)

Alter a timer file descriptor's internal timer. This function operates the same interval timer as `timerfd_settime_ns()`.

fd must be a valid timer file descriptor.

The timer's behaviour can be modified by specifying a *flags* value. Any of the following variables may used, combined using bitwise OR (the | operator):

- `TFD_TIMER_ABSTIME`
- `TFD_TIMER_CANCEL_ON_SET`

The timer is disabled by setting *initial* to zero (`0`). If *initial* is equal to or greater than zero, the timer is enabled. If *initial* is less than zero, it raises an `OSError` exception with `errno` set to `errno.EINVAL`

By default the timer will fire when *initial* seconds have elapsed. (If *initial* is zero, timer will fire immediately.)

However, if the `TFD_TIMER_ABSTIME` flag is set, the timer will fire when the timer's clock (set by *clockid* in `timerfd_create()`) reaches *initial* seconds.

The timer's interval is set by the *interval* `float`. If *interval* is zero, the timer only fires once, on the initial expiration. If *interval* is greater than zero, the timer fires every time *interval* seconds have elapsed since the previous expiration. If *interval* is less than zero, it raises `OSError` with `errno` set to `errno.EINVAL`

If the `TFD_TIMER_CANCEL_ON_SET` flag is set along with `TFD_TIMER_ABSTIME` and the clock for this timer is `time.CLOCK_REALTIME`, the timer is marked as cancelable if the real-time clock is changed discontinuously. Reading the descriptor is aborted with the error ECANCELED.

Linux manages system clock as UTC. A daylight-savings time transition is done by changing time offset only and doesn't cause discontinuous system clock change.

Discontinuous system clock change will be caused by the following events:

- `settimeofday`
- `clock_settime`
- set the system date and time by `date` command

Return a two-item tuple of (`next_expiration`, `interval`) from the previous timer state, before this function executed.

> **See also:** *timerfd_create(2)*, *timerfd_settime(2)*, *settimeofday(2)*, *clock_settime(2)*, and *date(1)*.

Availability: Linux >= 2.6.27 with glibc >= 2.8

> *Added in version 3.13.*

os.**timerfd_settime_ns**(*fd*, *, *, *flags=0*, *initial=0*, *interval=0*)

Similar to `timerfd_settime()`, but use time as nanoseconds. This function operates the same interval timer as `timerfd_settime()`.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**timerfd_gettime**(*fd*, */*)

Return a two-item tuple of floats (`next_expiration`, `interval`).

`next_expiration` denotes the relative time until next the timer next fires, regardless of if the `TFD_TIMER_ABSTIME` flag is set.

`interval` denotes the timer's interval. If zero, the timer will only fire once, after `next_expiration` seconds have elapsed.

See also: *timerfd_gettime(2)*

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**timerfd_gettime_ns**(*fd*, */*)

Similar to `timerfd_gettime()`, but return time as nanoseconds.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_NONBLOCK**

A flag for the `timerfd_create()` function, which sets the `O_NONBLOCK` status flag for the new timer file descriptor. If `TFD_NONBLOCK` is not set as a flag, `read()` blocks.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_CLOEXEC**

A flag for the `timerfd_create()` function, If `TFD_CLOEXEC` is set as a flag, set close-on-exec flag for new file descriptor.

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_TIMER_ABSTIME**

A flag for the `timerfd_settime()` and `timerfd_settime_ns()` functions. If this flag is set, *initial* is interpreted as an absolute value on the timer's clock (in UTC seconds or nanoseconds since the Unix Epoch).

Availability: Linux >= 2.6.27 with glibc >= 2.8

Added in version 3.13.

os.**TFD_TIMER_CANCEL_ON_SET**

A flag for the `timerfd_settime()` and `timerfd_settime_ns()` functions along with `TFD_TIMER_ABSTIME`. The timer is cancelled when the time of the underlying clock changes discontinuously.

Availability: Linux >= 2.6.27 with glibc >= 2.8

> *Added in version 3.13.*

Linux extended attributes

> *Added in version 3.3.*

These functions are all available on Linux only.

os.**getxattr**(*path*, *attribute*, *, *follow_symlinks=True*)

Return the value of the extended filesystem attribute *attribute* for *path*. *attribute* can be bytes or str (directly or indirectly through the `PathLike` interface). If it is str, it is encoded with the filesystem encoding.

This function can support specifying a file descriptor and not following symlinks.

Raises an auditing event `os.getxattr` with arguments `path`, `attribute`.

> *Changed in version 3.6:* Accepts a path-like object for *path* and *attribute*.

os.**listxattr**(*path=None*, *, *follow_symlinks=True*)

Return a list of the extended filesystem attributes on *path*. The attributes in the list are represented as strings decoded with the filesystem encoding. If *path* is `None`, `listxattr()` will examine the current directory.

This function can support specifying a file descriptor and not following symlinks.

Raises an auditing event `os.listxattr` with argument `path`.

> *Changed in version 3.6:* Accepts a path-like object.

os.**removexattr**(*path*, *attribute*, *, *follow_symlinks=True*)

Removes the extended filesystem attribute *attribute* from *path*. *attribute* should be bytes or str (directly or indirectly through the `PathLike` interface). If it is a string, it is encoded with the filesystem encoding and error handler.

This function can support specifying a file descriptor and not following symlinks.

Raises an auditing event `os.removexattr` with arguments `path`, `attribute`.

> *Changed in version 3.6:* Accepts a path-like object for *path* and *attribute*.

os.**setxattr**(*path*, *attribute*, *value*, *flags=0*, *, *follow_symlinks=True*)

Set the extended filesystem attribute *attribute* on *path* to *value*. *attribute* must be a bytes or str with no embedded NULs (directly or indirectly through the `PathLike` interface). If it is a str, it is encoded with the filesystem encoding and error handler. *flags* may be `XATTR_REPLACE` or `XATTR_CREATE`. If `XATTR_REPLACE` is given and the attribute does not exist, `ENODATA` will be raised. If `XATTR_CREATE` is given and the attribute already exists, the attribute will not be created and `EEXISTS` will be raised.

This function can support specifying a file descriptor and not following symlinks.

> **Note:** A bug in Linux kernel versions less than 2.6.39 caused the flags argument to be ignored on some filesystems.

Raises an auditing event `os.setxattr` with arguments `path`, `attribute`, `value`, `flags`.

> *Changed in version 3.6:* Accepts a path-like object for *path* and *attribute*.

os.XATTR_SIZE_MAX
The maximum size the value of an extended attribute can be. Currently, this is 64 KiB on Linux.

os.XATTR_CREATE
This is a possible value for the flags argument in `setxattr()`. It indicates the operation must create an attribute.

os.XATTR_REPLACE
This is a possible value for the flags argument in `setxattr()`. It indicates the operation must replace an existing attribute.

Process Management

These functions may be used to create and manage processes.

The various `exec*` functions take a list of arguments for the new program loaded into the process. In each case, the first of these arguments is passed to the new program as its own name rather than as an argument a user may have typed on a command line. For the C programmer, this is the `argv[0]` passed to a program's `main()`. For example, `os.execv('/bin/echo', ['foo', 'bar'])` will only print `bar` on standard output; `foo` will seem to be ignored.

os.abort()
Generate a `SIGABRT` signal to the current process. On Unix, the default behavior is to produce a core dump; on Windows, the process immediately returns an exit code of `3`. Be aware that calling this function will not call the Python signal handler registered for `SIGABRT` with `signal.signal()`.

os.add_dll_directory(*path*)
Add a path to the DLL search path.

This search path is used when resolving dependencies for imported extension modules (the module itself is resolved through `sys.path`), and also by `ctypes`.

Remove the directory by calling **close()** on the returned object or using it in a `with` statement.

See the Microsoft documentation for more information about how DLLs are loaded.

Raises an auditing event `os.add_dll_directory` with argument `path`.

Availability: Windows.

> *Added in version 3.8:* Previous versions of CPython would resolve DLLs using the default behavior for the current process. This led to inconsistencies, such as only sometimes searching `PATH` or the current working directory, and OS functions such as `AddDllDirectory` having no effect.
>
> In 3.8, the two primary ways DLLs are loaded now explicitly override the process-wide behavior to ensure consistency. See the porting notes for information on updating libraries.

os.execl(*path*, *arg0*, *arg1*, ...)
os.execle(*path*, *arg0*, *arg1*, ..., *env*)

os.**execlp**(*file*, *arg0*, *arg1*, ...)
os.**execlpe**(*file*, *arg0*, *arg1*, ..., *env*)
os.**execv**(*path*, *args*)
os.**execve**(*path*, *args*, *env*)
os.**execvp**(*file*, *args*)
os.**execvpe**(*file*, *args*, *env*)

These functions all execute a new program, replacing the current process; they do not return. On Unix, the new executable is loaded into the current process, and will have the same process id as the caller. Errors will be reported as `OSError` exceptions.

The current process is replaced immediately. Open file objects and descriptors are not flushed, so if there may be data buffered on these open files, you should flush them using `sys.stdout.flush()` or `os.fsync()` before calling an `exec*` function.

The "l" and "v" variants of the `exec*` functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the `execl*()` functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process should start with the name of the command being run, but this is not enforced.

The variants which include a "p" near the end (`execlp()`, `execlpe()`, `execvp()`, and `execvpe()`) will use the `PATH` environment variable to locate the program *file*. When the environment is being replaced (using one of the `exec*e` variants, discussed in the next paragraph), the new environment is used as the source of the `PATH` variable. The other variants, `execl()`, `execle()`, `execv()`, and `execve()`, will not use the `PATH` variable to locate the executable; *path* must contain an appropriate absolute or relative path. Relative paths must include at least one slash, even on Windows, as plain names will not be resolved.

For `execle()`, `execlpe()`, `execve()`, and `execvpe()` (note that these all end in "e"), the *env* parameter must be a mapping which is used to define the environment variables for the new process (these are used instead of the current process' environment); the functions `execl()`, `execlp()`, `execv()`, and `execvp()` all cause the new process to inherit the environment of the current process.

For `execve()` on some platforms, *path* may also be specified as an open file descriptor. This functionality may not be supported on your platform; you can check whether or not it is available using `os.supports_fd`. If it is unavailable, using it will raise a `NotImplementedError`.

Raises an auditing event `os.exec` with arguments `path`, `args`, `env`.

Availability: Unix, Windows, not WASI, not Android, not iOS.

> *Changed in version 3.3:* Added support for specifying *path* as an open file descriptor for `execve()`.

> *Changed in version 3.6:* Accepts a path-like object.

os.**_exit**(*n*)

Exit the process with status *n*, without calling cleanup handlers, flushing stdio buffers, etc.

> **Note:** The standard way to exit is `sys.exit(n)`. `_exit()` should normally only be used in the child process after a `fork()`.

The following exit codes are defined and can be used with `_exit()`, although they are not required. These are typically used for system programs written in Python, such as a mail server's external command delivery program.

Note: Some of these may not be available on all Unix platforms, since there is some variation. These constants are defined where they are defined by the underlying platform.

os.**EX_OK**
Exit code that means no error occurred. May be taken from the defined value of `EXIT_SUCCESS` on some platforms. Generally has a value of zero.

Availability: Unix, Windows.

os.**EX_USAGE**
Exit code that means the command was used incorrectly, such as when the wrong number of arguments are given.

Availability: Unix, not WASI.

os.**EX_DATAERR**
Exit code that means the input data was incorrect.

Availability: Unix, not WASI.

os.**EX_NOINPUT**
Exit code that means an input file did not exist or was not readable.

Availability: Unix, not WASI.

os.**EX_NOUSER**
Exit code that means a specified user did not exist.

Availability: Unix, not WASI.

os.**EX_NOHOST**
Exit code that means a specified host did not exist.

Availability: Unix, not WASI.

os.**EX_UNAVAILABLE**
Exit code that means that a required service is unavailable.

Availability: Unix, not WASI.

os.**EX_SOFTWARE**
Exit code that means an internal software error was detected.

Availability: Unix, not WASI.

os.**EX_OSERR**
Exit code that means an operating system error was detected, such as the inability to fork or create a pipe.

Availability: Unix, not WASI.

os.**EX_OSFILE**

Exit code that means some system file did not exist, could not be opened, or had some other kind of error.

Availability: Unix, not WASI.

os.**EX_CANTCREAT**

Exit code that means a user specified output file could not be created.

Availability: Unix, not WASI.

os.**EX_IOERR**

Exit code that means that an error occurred while doing I/O on some file.

Availability: Unix, not WASI.

os.**EX_TEMPFAIL**

Exit code that means a temporary failure occurred. This indicates something that may not really be an error, such as a network connection that couldn't be made during a retryable operation.

Availability: Unix, not WASI.

os.**EX_PROTOCOL**

Exit code that means that a protocol exchange was illegal, invalid, or not understood.

Availability: Unix, not WASI.

os.**EX_NOPERM**

Exit code that means that there were insufficient permissions to perform the operation (but not intended for file system problems).

Availability: Unix, not WASI.

os.**EX_CONFIG**

Exit code that means that some kind of configuration error occurred.

Availability: Unix, not WASI.

os.**EX_NOTFOUND**

Exit code that means something like "an entry was not found".

Availability: Unix, not WASI.

os.**fork**()

Fork a child process. Return `0` in the child and the child's process id in the parent. If an error occurs `OSError` is raised.

Note that some platforms including FreeBSD <= 6.3 and Cygwin have known issues when using `fork()` from a thread.

Raises an auditing event `os.fork` with no arguments.

> **Warning:** If you use TLS sockets in an application calling `fork()`, see the warning in the `ssl` documentation.

Warning: On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using `urllib.request`.

> *Changed in version 3.8:* Calling `fork()` in a subinterpreter is no longer supported (`RuntimeError` is raised).

> *Changed in version 3.12:* If Python is able to detect that your process has multiple threads, `os.fork()` now raises a `DeprecationWarning`.
>
> We chose to surface this as a warning, when detectable, to better inform developers of a design problem that the POSIX platform specifically notes as not supported. Even in code that *appears* to work, it has never been safe to mix threading with `os.fork()` on POSIX platforms. The CPython runtime itself has always made API calls that are not safe for use in the child process when threads existed in the parent (such as `malloc` and `free`).
>
> Users of macOS or users of libc or malloc implementations other than those typically found in glibc to date are among those already more likely to experience deadlocks running such code.
>
> See this discussion on fork being incompatible with threads for technical details of why we're surfacing this longstanding platform compatibility problem to developers.

Availability: POSIX, not WASI, not Android, not iOS.

os.forkpty()

Fork a child process, using a new pseudo-terminal as the child's controlling terminal. Return a pair of `(pid, fd)`, where *pid* is `0` in the child, the new child's process id in the parent, and *fd* is the file descriptor of the master end of the pseudo-terminal. For a more portable approach, use the `pty` module. If an error occurs `OSError` is raised.

Raises an auditing event `os.forkpty` with no arguments.

Warning: On macOS the use of this function is unsafe when mixed with using higher-level system APIs, and that includes using `urllib.request`.

> *Changed in version 3.8:* Calling `forkpty()` in a subinterpreter is no longer supported (`RuntimeError` is raised).

> *Changed in version 3.12:* If Python is able to detect that your process has multiple threads, this now raises a `DeprecationWarning`. See the longer explanation on `os.fork()`.

Availability: Unix, not WASI, not Android, not iOS.

os.kill(*pid*, *sig*, /)

Send signal *sig* to the process *pid*. Constants for the specific signals available on the host platform are defined in the `signal` module.

Windows: The `signal.CTRL_C_EVENT` and `signal.CTRL_BREAK_EVENT` signals are special signals which can only be sent to console processes which share a common console window, e.g., some subprocesses. Any other value for *sig* will cause the process to be unconditionally killed by the TerminateProcess API, and the exit code will be set to *sig*.

See also `signal.pthread_kill()`.

Raises an auditing event `os.kill` with arguments `pid`, `sig`.

Availability: Unix, Windows, not WASI, not iOS.

> *Changed in version 3.2:* Added Windows support.

os.**killpg**(*pgid, sig, /*)

Send the signal *sig* to the process group *pgid*.

Raises an auditing event `os.killpg` with arguments `pgid`, `sig`.

Availability: Unix, not WASI, not iOS.

os.**nice**(*increment, /*)

Add *increment* to the process's "niceness". Return the new niceness.

Availability: Unix, not WASI.

os.**pidfd_open**(*pid, flags=0*)

Return a file descriptor referring to the process *pid* with *flags* set. This descriptor can be used to perform process management without races and signals.

See the *pidfd_open(2)* man page for more details.

Availability: Linux >= 5.3, Android >= `build-time` API level 31

> *Added in version 3.9.*

os.**PIDFD_NONBLOCK**

This flag indicates that the file descriptor will be non-blocking. If the process referred to by the file descriptor has not yet terminated, then an attempt to wait on the file descriptor using *waitid(2)* will immediately return the error `EAGAIN` rather than blocking.

Availability: Linux >= 5.10

> *Added in version 3.12.*

os.**plock**(*op, /*)

Lock program segments into memory. The value of *op* (defined in `<sys/lock.h>`) determines which segments are locked.

Availability: Unix, not WASI, not iOS.

os.**popen**(*cmd, mode='r', buffering=-1*)

Open a pipe to or from command *cmd*. The return value is an open file object connected to the pipe, which can be read or written depending on whether *mode* is `'r'` (default) or `'w'`. The *buffering* argument have the same meaning as the corresponding argument to the built-in `open()` function. The returned file object reads or writes text strings rather than bytes.

The `close` method returns `None` if the subprocess exited successfully, or the subprocess's return code if there was an error. On POSIX systems, if the return code is positive it represents the return value of the

process left-shifted by one byte. If the return code is negative, the process was terminated by the signal given by the negated value of the return code. (For example, the return value might be `-signal.SIGKILL` if the subprocess was killed.) On Windows systems, the return value contains the signed integer return code from the child process.

On Unix, `waitstatus_to_exitcode()` can be used to convert the `close` method result (exit status) into an exit code if it is not `None`. On Windows, the `close` method result is directly the exit code (or `None`).

This is implemented using `subprocess.Popen`; see that class's documentation for more powerful ways to manage and communicate with subprocesses.

Availability: not WASI, not Android, not iOS.

> **Note:** The Python UTF-8 Mode affects encodings used for *cmd* and pipe contents.
>
> `popen()` is a simple wrapper around `subprocess.Popen`. Use `subprocess.Popen` or `subprocess.run()` to control options like encodings.

> *Deprecated since version 3.14:* The function is soft deprecated and should no longer be used to write new code. The `subprocess` module is recommended instead.

os.**posix_spawn**(*path*, *argv*, *env*, *, *file_actions=None*, *setpgroup=None*, *resetids=False*, *setsid=False*, *setsigmask=()*, *setsigdef=()*, *scheduler=None*)

Wraps the `posix_spawn()` C library API for use from Python.

Most users should use `subprocess.run()` instead of `posix_spawn()`.

The positional-only arguments *path*, *args*, and *env* are similar to `execve()`. *env* is allowed to be `None`, in which case current process' environment is used.

The *path* parameter is the path to the executable file. The *path* should contain a directory. Use `posix_spawnp()` to pass an executable file without directory.

The *file_actions* argument may be a sequence of tuples describing actions to take on specific file descriptors in the child process between the C library implementation's `fork()` and `exec()` steps. The first item in each tuple must be one of the three type indicator listed below describing the remaining tuple elements:

os.**POSIX_SPAWN_OPEN**

(`os.POSIX_SPAWN_OPEN`, *fd*, *path*, *flags*, *mode*)

Performs `os.dup2(os.open(path, flags, mode), fd)`.

os.**POSIX_SPAWN_CLOSE**

(`os.POSIX_SPAWN_CLOSE`, *fd*)

Performs `os.close(fd)`.

os.**POSIX_SPAWN_DUP2**

(`os.POSIX_SPAWN_DUP2`, *fd*, *new_fd*)

Performs `os.dup2(fd, new_fd)`.

`os.`**`POSIX_SPAWN_CLOSEFROM`**

(`os.POSIX_SPAWN_CLOSEFROM`, *fd*)

Performs `os.closerange(fd, INF)`.

These tuples correspond to the C library `posix_spawn_file_actions_addopen()`, `posix_spawn_file_actions_addclose()`, `posix_spawn_file_actions_adddup2()`, and `posix_spawn_file_actions_addclosefrom_np()` API calls used to prepare for the `posix_spawn()` call itself.

The *setpgroup* argument will set the process group of the child to the value specified. If the value specified is 0, the child's process group ID will be made the same as its process ID. If the value of *setpgroup* is not set, the child will inherit the parent's process group ID. This argument corresponds to the C library `POSIX_SPAWN_SETPGROUP` flag.

If the *resetids* argument is `True` it will reset the effective UID and GID of the child to the real UID and GID of the parent process. If the argument is `False`, then the child retains the effective UID and GID of the parent. In either case, if the set-user-ID and set-group-ID permission bits are enabled on the executable file, their effect will override the setting of the effective UID and GID. This argument corresponds to the C library `POSIX_SPAWN_RESETIDS` flag.

If the *setsid* argument is `True`, it will create a new session ID for `posix_spawn`. *setsid* requires `POSIX_SPAWN_SETSID` or `POSIX_SPAWN_SETSID_NP` flag. Otherwise, `NotImplementedError` is raised.

The *setsigmask* argument will set the signal mask to the signal set specified. If the parameter is not used, then the child inherits the parent's signal mask. This argument corresponds to the C library `POSIX_SPAWN_SETSIGMASK` flag.

The *sigdef* argument will reset the disposition of all signals in the set specified. This argument corresponds to the C library `POSIX_SPAWN_SETSIGDEF` flag.

The *scheduler* argument must be a tuple containing the (optional) scheduler policy and an instance of `sched_param` with the scheduler parameters. A value of `None` in the place of the scheduler policy indicates that is not being provided. This argument is a combination of the C library `POSIX_SPAWN_SETSCHEDPARAM` and `POSIX_SPAWN_SETSCHEDULER` flags.

Raises an auditing event `os.posix_spawn` with arguments `path`, `argv`, `env`.

> *Added in version 3.8.*

> *Changed in version 3.13:* *env* parameter accepts `None`. `os.POSIX_SPAWN_CLOSEFROM` is available on platforms where `posix_spawn_file_actions_addclosefrom_np()` exists.

Availability: Unix, not WASI, not Android, not iOS.

`os.`**`posix_spawnp`**`(`*`path, argv, env, *, file_actions=None, setpgroup=None, resetids=False, setsid=False, setsigmask=(), setsigdef=(), scheduler=None`*`)`

Wraps the `posix_spawnp()` C library API for use from Python.

Similar to `posix_spawn()` except that the system searches for the *executable* file in the list of directories specified by the `PATH` environment variable (in the same way as for `execvp(3)`).

Raises an auditing event `os.posix_spawn` with arguments `path`, `argv`, `env`.

> *Added in version 3.8.*

Availability: POSIX, not WASI, not Android, not iOS.

See `posix_spawn()` documentation.

os.register_at_fork(*, *before=None*, *after_in_parent=None*, *after_in_child=None*)

Register callables to be executed when a new child process is forked using `os.fork()` or similar process cloning APIs. The parameters are optional and keyword-only. Each specifies a different call point.

- *before* is a function called before forking a child process.
- *after_in_parent* is a function called from the parent process after forking a child process.
- *after_in_child* is a function called from the child process.

These calls are only made if control is expected to return to the Python interpreter. A typical `subprocess` launch will not trigger them as the child is not going to re-enter the interpreter.

Functions registered for execution before forking are called in reverse registration order. Functions registered for execution after forking (either in the parent or in the child) are called in registration order.

Note that `fork()` calls made by third-party C code may not call those functions, unless it explicitly calls `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` and `PyOS_AfterFork_Child()`.

There is no way to unregister a function.

Availability: Unix, not WASI, not Android, not iOS.

> *Added in version 3.7.*

os.spawnl(*mode*, *path*, *...*)
os.spawnle(*mode*, *path*, *...*, *env*)
os.spawnlp(*mode*, *file*, *...*)
os.spawnlpe(*mode*, *file*, *...*, *env*)
os.spawnv(*mode*, *path*, *args*)
os.spawnve(*mode*, *path*, *args*, *env*)
os.spawnvp(*mode*, *file*, *args*)
os.spawnvpe(*mode*, *file*, *args*, *env*)

Execute the program *path* in a new process.

(Note that the `subprocess` module provides more powerful facilities for spawning new processes and retrieving their results; using that module is preferable to using these functions. Check especially the Replacing Older Functions with the subprocess Module section.)

If *mode* is `P_NOWAIT`, this function returns the process id of the new process; if *mode* is `P_WAIT`, returns the process's exit code if it exits normally, or `-signal`, where *signal* is the signal that killed the process. On Windows, the process id will actually be the process handle, so can be used with the `waitpid()` function.

Note on VxWorks, this function doesn't return `-signal` when the new process is killed. Instead it raises OSError exception.

The "l" and "v" variants of the `spawn*` functions differ in how command-line arguments are passed. The "l" variants are perhaps the easiest to work with if the number of parameters is fixed when the code is written; the individual parameters simply become additional parameters to the `spawnl*()` functions. The "v" variants are good when the number of parameters is variable, with the arguments being passed in a list or tuple as the *args* parameter. In either case, the arguments to the child process must start with the name of the command being run.

The variants which include a second "p" near the end (`spawnlp()`, `spawnlpe()`, `spawnvp()`, and `spawnvpe()`) will use the `PATH` environment variable to locate the program *file*. When the environment is being replaced (using one of the `spawn*e` variants, discussed in the next paragraph), the new environment is used as the source of the `PATH` variable. The other variants, `spawnl()`, `spawnle()`, `spawnv()`, and `spawnve()`, will not use the `PATH` variable to locate the executable; *path* must contain an appropriate absolute or relative path.

For `spawnle()`, `spawnlpe()`, `spawnve()`, and `spawnvpe()` (note that these all end in "e"), the *env* parameter must be a mapping which is used to define the environment variables for the new process (they are used instead of the current process' environment); the functions `spawnl()`, `spawnlp()`, `spawnv()`, and `spawnvp()` all cause the new process to inherit the environment of the current process. Note that keys and values in the *env* dictionary must be strings; invalid keys or values will cause the function to fail, with a return value of `127`.

As an example, the following calls to `spawnlp()` and `spawnvpe()` are equivalent:

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

Raises an auditing event `os.spawn` with arguments `mode`, `path`, `args`, `env`.

Availability: Unix, Windows, not WASI, not Android, not iOS.

`spawnlp()`, `spawnlpe()`, `spawnvp()` and `spawnvpe()` are not available on Windows. `spawnle()` and `spawnve()` are not thread-safe on Windows; we advise you to use the `subprocess` module instead.

> *Changed in version 3.6:* Accepts a path-like object.

> *Deprecated since version 3.14:* These functions are soft deprecated and should no longer be used to write new code. The `subprocess` module is recommended instead.

os.**P_NOWAIT**
os.**P_NOWAITO**

Possible values for the *mode* parameter to the `spawn*` family of functions. If either of these values is given, the `spawn*` functions will return as soon as the new process has been created, with the process id as the return value.

Availability: Unix, Windows.

os.**P_WAIT**

Possible value for the *mode* parameter to the `spawn*` family of functions. If this is given as *mode*, the `spawn*` functions will not return until the new process has run to completion and will return the exit code of the process the run is successful, or `-signal` if a signal kills the process.

Availability: Unix, Windows.

os.**P_DETACH**
os.**P_OVERLAY**

Possible values for the *mode* parameter to the `spawn*` family of functions. These are less portable than those listed above. `P_DETACH` is similar to `P_NOWAIT`, but the new process is detached from the console of the calling process. If `P_OVERLAY` is used, the current process will be replaced; the `spawn*` function will not return.

Availability: Windows.

os.**startfile**(*path*[, *operation*][, *arguments*][, *cwd*][, *show_cmd*])

Start a file with its associated application.

When *operation* is not specified, this acts like double-clicking the file in Windows Explorer, or giving the file name as an argument to the **start** command from the interactive command shell: the file is opened with whatever application (if any) its extension is associated.

When another *operation* is given, it must be a "command verb" that specifies what should be done with the file. Common verbs documented by Microsoft are `'open'`, `'print'` and `'edit'` (to be used on files) as well as `'explore'` and `'find'` (to be used on directories).

When launching an application, specify *arguments* to be passed as a single string. This argument may have no effect when using this function to launch a document.

The default working directory is inherited, but may be overridden by the *cwd* argument. This should be an absolute path. A relative *path* will be resolved against this argument.

Use *show_cmd* to override the default window style. Whether this has any effect will depend on the application being launched. Values are integers as supported by the Win32 `ShellExecute()` function.

`startfile()` returns as soon as the associated application is launched. There is no option to wait for the application to close, and no way to retrieve the application's exit status. The *path* parameter is relative to the current directory or *cwd*. If you want to use an absolute path, make sure the first character is not a slash (`'/'`) Use `pathlib` or the `os.path.normpath()` function to ensure that paths are properly encoded for Win32.

To reduce interpreter startup overhead, the Win32 `ShellExecute()` function is not resolved until this function is first called. If the function cannot be resolved, `NotImplementedError` will be raised.

Raises an auditing event `os.startfile` with arguments `path`, `operation`.

Raises an auditing event `os.startfile/2` with arguments `path`, `operation`, `arguments`, `cwd`, `show_cmd`.

Availability: Windows.

> *Changed in version 3.10:* Added the *arguments*, *cwd* and *show_cmd* arguments, and the `os.startfile/2` audit event.

`os.`**`system`**`(`*`command`*`)`

Execute the command (a string) in a subshell. This is implemented by calling the Standard C function `system()`, and has the same limitations. Changes to `sys.stdin`, etc. are not reflected in the environment of the executed command. If *command* generates any output, it will be sent to the interpreter standard output stream. The C standard does not specify the meaning of the return value of the C function, so the return value of the Python function is system-dependent.

On Unix, the return value is the exit status of the process encoded in the format specified for `wait()`.

On Windows, the return value is that returned by the system shell after running *command*. The shell is given by the Windows environment variable `COMSPEC`: it is usually **cmd.exe**, which returns the exit status of the command run; on systems using a non-native shell, consult your shell documentation.

The `subprocess` module provides more powerful facilities for spawning new processes and retrieving their results; using that module is recommended to using this function. See the Replacing Older Functions with the subprocess Module section in the `subprocess` documentation for some helpful recipes.

On Unix, `waitstatus_to_exitcode()` can be used to convert the result (exit status) into an exit code. On Windows, the result is directly the exit code.

Raises an auditing event `os.system` with argument `command`.

Availability: Unix, Windows, not WASI, not Android, not iOS.

`os.`**`times`**`()`

Returns the current global process times. The return value is an object with five attributes:

- `user` - user time
- `system` - system time
- `children_user` - user time of all child processes
- `children_system` - system time of all child processes
- `elapsed` - elapsed real time since a fixed point in the past

For backwards compatibility, this object also behaves like a five-tuple containing `user`, `system`, `children_user`, `children_system`, and `elapsed` in that order.

See the Unix manual page *times(2)* and times(3) manual page on Unix or the GetProcessTimes MSDN on Windows. On Windows, only `user` and `system` are known; the other attributes are zero.

Availability: Unix, Windows.

> *Changed in version 3.3:* Return type changed from a tuple to a tuple-like object with named attributes.

`os.`**`wait`**`()`

Wait for completion of a child process, and return a tuple containing its pid and exit status indication: a 16-bit number, whose low byte is the signal number that killed the process, and whose high byte is the exit status (if the signal number is zero); the high bit of the low byte is set if a core file was produced.

If there are no children that could be waited for, `ChildProcessError` is raised.

`waitstatus_to_exitcode()` can be used to convert the exit status into an exit code.

Availability: Unix, not WASI, not Android, not iOS.

> **See also:** The other `wait*()` functions documented below can be used to wait for the completion of a specific child process and have more options. `waitpid()` is the only one also available on Windows.

`os.`**`waitid`**`(`*`idtype, id, options, /`*`)`

Wait for the completion of a child process.

idtype can be `P_PID`, `P_PGID`, `P_ALL`, or (on Linux) `P_PIDFD`. The interpretation of *id* depends on it; see their individual descriptions.

options is an OR combination of flags. At least one of `WEXITED`, `WSTOPPED` or `WCONTINUED` is required; `WNOHANG` and `WNOWAIT` are additional optional flags.

The return value is an object representing the data contained in the `siginfo_t` structure with the following attributes:

- `si_pid` (process ID)
- `si_uid` (real user ID of the child)
- `si_signo` (always `SIGCHLD`)
- `si_status` (the exit status or signal number, depending on `si_code`)
- `si_code` (see `CLD_EXITED` for possible values)

If `WNOHANG` is specified and there are no matching children in the requested state, `None` is returned. Otherwise, if there are no matching children that could be waited for, `ChildProcessError` is raised.

Availability: Unix, not WASI, not Android, not iOS.

> *Added in version 3.3.*

> *Changed in version 3.13:* This function is now available on macOS as well.

`os.`**`waitpid`**`(`*`pid, options, /`*`)`

The details of this function differ on Unix and Windows.

On Unix: Wait for completion of a child process given by process id *pid*, and return a tuple containing its process id and exit status indication (encoded as for `wait()`). The semantics of the call are affected by the value of the integer *options*, which should be `0` for normal operation.

If *pid* is greater than `0`, `waitpid()` requests status information for that specific process. If *pid* is `0`, the request is for the status of any child in the process group of the current process. If *pid* is `-1`, the request pertains to any child of the current process. If *pid* is less than `-1`, status is requested for any process in the process group `-pid` (the absolute value of *pid*).

options is an OR combination of flags. If it contains `WNOHANG` and there are no matching children in the requested state, `(0, 0)` is returned. Otherwise, if there are no matching children that could be waited for, `ChildProcessError` is raised. Other options that can be used are `WUNTRACED` and `WCONTINUED`.

On Windows: Wait for completion of a process given by process handle *pid*, and return a tuple containing *pid*, and its exit status shifted left by 8 bits (shifting makes cross-platform use of the function easier). A *pid* less than or equal to `0` has no special meaning on Windows, and raises an exception. The value of integer *options* has no effect. *pid* can refer to any process whose id is known, not necessarily a child process. The `spawn*` functions called with `P_NOWAIT` return suitable process handles.

`waitstatus_to_exitcode()` can be used to convert the exit status into an exit code.

Availability: Unix, Windows, not WASI, not Android, not iOS.

> *Changed in version 3.5:* If the system call is interrupted and the signal handler does not raise an exception, the function now retries the system call instead of raising an `InterruptedError` exception (see **PEP 475** for the rationale).

os.**wait3**(*options*)

Similar to `waitpid()`, except no process id argument is given and a 3-element tuple containing the child's process id, exit status indication, and resource usage information is returned. Refer to `resource.getrusage()` for details on resource usage information. The *options* argument is the same as that provided to `waitpid()` and `wait4()`.

`waitstatus_to_exitcode()` can be used to convert the exit status into an exitcode.

Availability: Unix, not WASI, not Android, not iOS.

os.**wait4**(*pid, options*)

Similar to `waitpid()`, except a 3-element tuple, containing the child's process id, exit status indication, and resource usage information is returned. Refer to `resource.getrusage()` for details on resource usage information. The arguments to `wait4()` are the same as those provided to `waitpid()`.

`waitstatus_to_exitcode()` can be used to convert the exit status into an exitcode.

Availability: Unix, not WASI, not Android, not iOS.

os.**P_PID**
os.**P_PGID**
os.**P_ALL**
os.**P_PIDFD**

These are the possible values for *idtype* in `waitid()`. They affect how *id* is interpreted:

- `P_PID` - wait for the child whose PID is *id*.
- `P_PGID` - wait for any child whose progress group ID is *id*.
- `P_ALL` - wait for any child; *id* is ignored.
- `P_PIDFD` - wait for the child identified by the file descriptor *id* (a process file descriptor created with `pidfd_open()`).

Availability: Unix, not WASI, not Android, not iOS.

> **Note:** `P_PIDFD` is only available on Linux >= 5.4.

> *Added in version 3.3.*

> *Added in version 3.9:* The `P_PIDFD` constant.

os.**WCONTINUED**

This *options* flag for `waitpid()`, `wait3()`, `wait4()`, and `waitid()` causes child processes to be reported if they have been continued from a job control stop since they were last reported.

Availability: Unix, not WASI, not Android, not iOS.

os.**WEXITED**

This *options* flag for `waitid()` causes child processes that have terminated to be reported.

The other `wait*` functions always report children that have terminated, so this option is not available for them.

Availability: Unix, not WASI, not Android, not iOS.

> *Added in version 3.3.*

os.**WSTOPPED**

This *options* flag for `waitid()` causes child processes that have been stopped by the delivery of a signal to be reported.

This option is not available for the other `wait*` functions.

Availability: Unix, not WASI, not Android, not iOS.

> *Added in version 3.3.*

os.**WUNTRACED**

This *options* flag for `waitpid()`, `wait3()`, and `wait4()` causes child processes to also be reported if they have been stopped but their current state has not been reported since they were stopped.

This option is not available for `waitid()`.

Availability: Unix, not WASI, not Android, not iOS.

os.**WNOHANG**

This *options* flag causes `waitpid()`, `wait3()`, `wait4()`, and `waitid()` to return right away if no child process status is available immediately.

Availability: Unix, not WASI, not Android, not iOS.

os.**WNOWAIT**

This *options* flag causes `waitid()` to leave the child in a waitable state, so that a later `wait*()` call can be used to retrieve the child status information again.

This option is not available for the other `wait*` functions.

Availability: Unix, not WASI, not Android, not iOS.

os.**CLD_EXITED**
os.**CLD_KILLED**
os.**CLD_DUMPED**
os.**CLD_TRAPPED**
os.**CLD_STOPPED**
os.**CLD_CONTINUED**

These are the possible values for `si_code` in the result returned by `waitid()`.

Availability: Unix, not WASI, not Android, not iOS.

> *Added in version 3.3.*

> *Changed in version 3.9:* Added `CLD_KILLED` and `CLD_STOPPED` values.

os.**waitstatus_to_exitcode**(*status*)

Convert a wait status to an exit code.

On Unix:

- If the process exited normally (if `WIFEXITED(status)` is true), return the process exit status (return `WEXITSTATUS(status)`): result greater than or equal to 0.
- If the process was terminated by a signal (if `WIFSIGNALED(status)` is true), return `-signum` where *signum* is the number of the signal that caused the process to terminate (return `-WTERMSIG(status)`): result less than 0.
- Otherwise, raise a `ValueError`.

On Windows, return *status* shifted right by 8 bits.

On Unix, if the process is being traced or if `waitpid()` was called with `WUNTRACED` option, the caller must first check if `WIFSTOPPED(status)` is true. This function must not be called if `WIFSTOPPED(status)` is true.

> **See also:** `WIFEXITED()`, `WEXITSTATUS()`, `WIFSIGNALED()`, `WTERMSIG()`, `WIFSTOPPED()`, `WSTOPSIG()` functions.

Availability: Unix, Windows, not WASI, not Android, not iOS.

> *Added in version 3.9.*

The following functions take a process status code as returned by `system()`, `wait()`, or `waitpid()` as a parameter. They may be used to determine the disposition of a process.

os.**WCOREDUMP**(*status*, */*)

Return `True` if a core dump was generated for the process, otherwise return `False`.

This function should be employed only if `WIFSIGNALED()` is true.

Availability: Unix, not WASI, not Android, not iOS.

os.**WIFCONTINUED**(*status*)

Return `True` if a stopped child has been resumed by delivery of `SIGCONT` (if the process has been continued from a job control stop), otherwise return `False`.

See `WCONTINUED` option.

Availability: Unix, not WASI, not Android, not iOS.

os.**WIFSTOPPED**(*status*)

Return `True` if the process was stopped by delivery of a signal, otherwise return `False`.

`WIFSTOPPED()` only returns `True` if the `waitpid()` call was done using `WUNTRACED` option or when the process is being traced (see *ptrace(2)*).

Availability: Unix, not WASI, not Android, not iOS.

os.**WIFSIGNALED**(*status*)

Return `True` if the process was terminated by a signal, otherwise return `False`.

Availability: Unix, not WASI, not Android, not iOS.

os.**WIFEXITED**(*status*)

Return `True` if the process exited terminated normally, that is, by calling `exit()` or `_exit()`, or by returning from `main()`; otherwise return `False`.

Availability: Unix, not WASI, not Android, not iOS.

os.**WEXITSTATUS**(*status*)

Return the process exit status.

This function should be employed only if `WIFEXITED()` is true.

Availability: Unix, not WASI, not Android, not iOS.

os.**WSTOPSIG**(*status*)

Return the signal which caused the process to stop.

This function should be employed only if `WIFSTOPPED()` is true.

Availability: Unix, not WASI, not Android, not iOS.

os.**WTERMSIG**(*status*)

Return the number of the signal that caused the process to terminate.

This function should be employed only if `WIFSIGNALED()` is true.

Availability: Unix, not WASI, not Android, not iOS.

Interface to the scheduler

These functions control how a process is allocated CPU time by the operating system. They are only available on some Unix platforms. For more detailed information, consult your Unix manpages.

> *Added in version 3.3.*

The following scheduling policies are exposed if they are supported by the operating system.

os.**SCHED_OTHER**

The default scheduling policy.

os.**SCHED_BATCH**

Scheduling policy for CPU-intensive processes that tries to preserve interactivity on the rest of the computer.

os.**SCHED_DEADLINE**

Scheduling policy for tasks with deadline constraints.

> *Added in version 3.14.*

os.**SCHED_IDLE**

Scheduling policy for extremely low priority background tasks.

os.**SCHED_NORMAL**

Alias for `SCHED_OTHER`.

> *Added in version 3.14.*

os.**SCHED_SPORADIC**

Scheduling policy for sporadic server programs.

os.**SCHED_FIFO**

A First In First Out scheduling policy.

os.**SCHED_RR**

A round-robin scheduling policy.

os.**SCHED_RESET_ON_FORK**

This flag can be OR'ed with any other scheduling policy. When a process with this flag set forks, its child's scheduling policy and priority are reset to the default.

class os.**sched_param**(*sched_priority*)

This class represents tunable scheduling parameters used in `sched_setparam()`, `sched_setscheduler()`, and `sched_getparam()`. It is immutable.

At the moment, there is only one possible parameter:

sched_priority

The scheduling priority for a scheduling policy.

os.**sched_get_priority_min**(*policy*)

Get the minimum priority value for *policy*. *policy* is one of the scheduling policy constants above.

os.**sched_get_priority_max**(*policy*)

Get the maximum priority value for *policy*. *policy* is one of the scheduling policy constants above.

os.**sched_setscheduler**(*pid*, *policy*, *param*, /)

Set the scheduling policy for the process with PID *pid*. A *pid* of 0 means the calling process. *policy* is one of the scheduling policy constants above. *param* is a `sched_param` instance.

os.**sched_getscheduler**(*pid*, /)

Return the scheduling policy for the process with PID *pid*. A *pid* of 0 means the calling process. The result is one of the scheduling policy constants above.

os.**sched_setparam**(*pid*, *param*, /)

Set the scheduling parameters for the process with PID *pid*. A *pid* of 0 means the calling process. *param* is a `sched_param` instance.

os.**sched_getparam**(*pid*, /)

Return the scheduling parameters as a `sched_param` instance for the process with PID *pid*. A *pid* of 0 means the calling process.

os.**sched_rr_get_interval**(*pid*, /)

Return the round-robin quantum in seconds for the process with PID *pid*. A *pid* of 0 means the calling process.

os.**sched_yield**()

Voluntarily relinquish the CPU. See *sched_yield(2)* for details.

os.**sched_setaffinity**(*pid*, *mask*, /)

Restrict the process with PID *pid* (or the current process if zero) to a set of CPUs. *mask* is an iterable of integers representing the set of CPUs to which the process should be restricted.

os.**sched_getaffinity**(*pid*, /)

Return the set of CPUs the process with PID *pid* is restricted to.

If *pid* is zero, return the set of CPUs the calling thread of the current process is restricted to.

See also the `process_cpu_count()` function.

Miscellaneous System Information

os.**confstr**(*name*, /)

Return string-valued system configuration values. *name* specifies the configuration value to retrieve; it may be a string which is the name of a defined system value; these names are specified in a number of standards (POSIX, Unix 95, Unix 98, and others). Some platforms define additional names as well. The names known to the host operating system are given as the keys of the `confstr_names` dictionary. For configuration variables not included in that mapping, passing an integer for *name* is also accepted.

If the configuration value specified by *name* isn't defined, `None` is returned.

If *name* is a string and is not known, `ValueError` is raised. If a specific value for *name* is not supported by the host system, even if it is included in `confstr_names`, an `OSError` is raised with `errno.EINVAL` for the error number.

Availability: Unix.

os.**confstr_names**

Dictionary mapping names accepted by `confstr()` to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

os.**cpu_count**()

Return the number of logical CPUs in the **system**. Returns `None` if undetermined.

The `process_cpu_count()` function can be used to get the number of logical CPUs usable by the calling thread of the **current process**.

> *Added in version 3.4.*

> *Changed in version 3.13:* If `-X cpu_count` is given or `PYTHON_CPU_COUNT` is set, `cpu_count()` returns the overridden value *n*.

os.**getloadavg**()

Return the number of processes in the system run queue averaged over the last 1, 5, and 15 minutes or raises `OSError` if the load average was unobtainable.

Availability: Unix.

os.**process_cpu_count**()

Get the number of logical CPUs usable by the calling thread of the **current process**. Returns `None` if undetermined. It can be less than `cpu_count()` depending on the CPU affinity.

The `cpu_count()` function can be used to get the number of logical CPUs in the **system**.

If `-X cpu_count` is given or `PYTHON_CPU_COUNT` is set, `process_cpu_count()` returns the overridden value *n*.

See also the `sched_getaffinity()` function.

> *Added in version 3.13.*

os.**sysconf**(*name*, /)

Return integer-valued system configuration values. If the configuration value specified by *name* isn't defined, `-1` is returned. The comments regarding the *name* parameter for `confstr()` apply here as well; the dictionary that provides information on the known names is given by `sysconf_names`.

Availability: Unix.

os.**sysconf_names**

Dictionary mapping names accepted by `sysconf()` to the integer values defined for those names by the host operating system. This can be used to determine the set of names known to the system.

Availability: Unix.

> *Changed in version 3.11:* Add `'SC_MINSIGSTKSZ'` name.

The following data values are used to support path manipulation operations. These are defined for all platforms.

Higher-level operations on pathnames are defined in the `os.path` module.

os.**curdir**

The constant string used by the operating system to refer to the current directory. This is `'.'` for Windows and POSIX. Also available via `os.path`.

os.**pardir**

The constant string used by the operating system to refer to the parent directory. This is `'..'` for Windows and POSIX. Also available via `os.path`.

os.**sep**

The character used by the operating system to separate pathname components. This is `'/'` for POSIX and `'\\'` for Windows. Note that knowing this is not sufficient to be able to parse or concatenate pathnames — use `os.path.split()` and `os.path.join()` — but it is occasionally useful. Also available via `os.path`.

os.**altsep**

An alternative character used by the operating system to separate pathname components, or `None` if only one separator character exists. This is set to `'/'` on Windows systems where `sep` is a backslash. Also available via `os.path`.

os.**extsep**

The character which separates the base filename from the extension; for example, the `'.'` in `os.py`. Also available via `os.path`.

os.**pathsep**

The character conventionally used by the operating system to separate search path components (as in `PATH`), such as `':'` for POSIX or `';'` for Windows. Also available via `os.path`.

os.**defpath**

The default search path used by `exec*p*` and `spawn*p*` if the environment doesn't have a `'PATH'` key. Also available via `os.path`.

os.**linesep**

The string used to separate (or, rather, terminate) lines on the current platform. This may be a single character, such as `'\n'` for POSIX, or multiple characters, for example, `'\r\n'` for Windows. Do not use *os.linesep* as a line terminator when writing files opened in text mode (the default); use a single `'\n'` instead, on all platforms.

os.**devnull**

The file path of the null device. For example: `'/dev/null'` for POSIX, `'nul'` for Windows. Also available via `os.path`.

os.**RTLD_LAZY**
os.**RTLD_NOW**
os.**RTLD_GLOBAL**
os.**RTLD_LOCAL**
os.**RTLD_NODELETE**
os.**RTLD_NOLOAD**
os.**RTLD_DEEPBIND**

Flags for use with the `setdlopenflags()` and `getdlopenflags()` functions. See the Unix manual page *dlopen(3)* for what the different flags mean.

> *Added in version 3.3.*

Random numbers

os.**getrandom**(*size*, *flags=0*)

Get up to *size* random bytes. The function can return less bytes than requested.

These bytes can be used to seed user-space random number generators or for cryptographic purposes.

`getrandom()` relies on entropy gathered from device drivers and other sources of environmental noise. Unnecessarily reading large quantities of data will have a negative impact on other users of the

`/dev/random` and `/dev/urandom` devices.

The flags argument is a bit mask that can contain zero or more of the following values ORed together: `os.GRND_RANDOM` and `GRND_NONBLOCK`.

See also the Linux getrandom() manual page.

Availability: Linux >= 3.17.

> *Added in version 3.6.*

`os.`**`urandom`**`(`*`size`*`, /)`

Return a bytestring of *size* random bytes suitable for cryptographic use.

This function returns random bytes from an OS-specific randomness source. The returned data should be unpredictable enough for cryptographic applications, though its exact quality depends on the OS implementation.

On Linux, if the `getrandom()` syscall is available, it is used in blocking mode: block until the system urandom entropy pool is initialized (128 bits of entropy are collected by the kernel). See the **PEP 524** for the rationale. On Linux, the `getrandom()` function can be used to get random bytes in non-blocking mode (using the `GRND_NONBLOCK` flag) or to poll until the system urandom entropy pool is initialized.

On a Unix-like system, random bytes are read from the `/dev/urandom` device. If the `/dev/urandom` device is not available or not readable, the `NotImplementedError` exception is raised.

On Windows, it will use `BCryptGenRandom()`.

> **See also:** The `secrets` module provides higher level functions. For an easy-to-use interface to the random number generator provided by your platform, please see `random.SystemRandom`.

> *Changed in version 3.5:* On Linux 3.17 and newer, the `getrandom()` syscall is now used when available. On OpenBSD 5.6 and newer, the C `getentropy()` function is now used. These functions avoid the usage of an internal file descriptor.

> *Changed in version 3.5.2:* On Linux, if the `getrandom()` syscall blocks (the urandom entropy pool is not initialized yet), fall back on reading `/dev/urandom`.

> *Changed in version 3.6:* On Linux, `getrandom()` is now used in blocking mode to increase the security.

> *Changed in version 3.11:* On Windows, `BCryptGenRandom()` is used instead of `CryptGenRandom()` which is deprecated.

`os.`**`GRND_NONBLOCK`**

By default, when reading from `/dev/random`, `getrandom()` blocks if no random bytes are available, and when reading from `/dev/urandom`, it blocks if the entropy pool has not yet been initialized.

If the `GRND_NONBLOCK` flag is set, then `getrandom()` does not block in these cases, but instead immediately raises `BlockingIOError`.

> *Added in version 3.6.*

os.**GRND_RANDOM**

If this bit is set, then random bytes are drawn from the `/dev/random` pool instead of the `/dev/urandom` pool.

> *Added in version 3.6.*