

16.1. `os` — Miscellaneous operating system interfaces

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.
- An “Availability: Unix” note means that this function is commonly found on Unix systems. It does not make any claims about its existence on a specific operating system.
- If not separately noted, all functions that claim “Availability: Unix” are supported on Mac OS X, which builds on a Unix core.

Note: All functions in this module raise `OSError` 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', 'mac', 'os2', 'ce', '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.

16.1.1. 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 file system encoding to perform this conversion (see `sys.getfilesystemencoding()`).

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+DCxx 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 may raise `UnicodeErrors`.

16.1.2. 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.

`os.environ`

A [mapping](#) object representing the string 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.

If the platform supports the `putenv()` function, 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.

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

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

If `putenv()` is not provided, a modified copy of this mapping may be passed to the appropriate process-creation functions to cause child processes to use a modified environment.

If the platform supports the `unsetenv()` function, 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.

`os.environb`

Bytes version of `environ`: a *mapping* object representing the environment as byte strings. `environ` and `environb` are synchronized (modify `environb` updates `environ`, and vice versa).

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

New in version 3.2.

`os.chdir(path)`

`os.fchdir(fd)`

`os.getcwd()`

These functions are described in *Files and Directories*.

`os.fsencode(filename)`

Encode *filename* to the filesystem encoding with `'surrogateescape'` error handler, or `'strict'` on Windows; return `bytes` unchanged.

`fsdecode()` is the reverse function.

New in version 3.2.

`os.fsdecode(filename)`

Decode *filename* from the filesystem encoding with `'surrogateescape'` error handler, or `'strict'` on Windows; return `str` unchanged.

`fsencode()` is the reverse function.

New in version 3.2.

os.getenv(key, default=None)

Return the value of the environment variable *key* if it exists, or *default* if it doesn't. *key*, *default* and the result are str.

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: most flavors of Unix, Windows.

os.getenvb(key, default=None)

Return the value of the environment variable *key* if it exists, or *default* if it doesn't. *key*, *default* and the result are bytes.

Availability: most flavors of Unix.

New 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.

New 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.

os.geteuid()

Return the current process's effective user id.

Availability: Unix.

os.getgid()

Return the real group id of the current process.

Availability: Unix.

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*.

Availability: Unix.

New in version 3.3.

`os.getgroups()`

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

Availability: Unix.

Note: On Mac OS X, `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 the environment variables `LOGNAME` or `USERNAME` to find out who the user is, or `pwd.getpwuid(os.getuid())[0]` to get the login name of the currently effective user id.

Availability: Unix, Windows.

`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.

`os.getpgrp()`

Return the id of the current process group.

Availability: Unix.

`os.getpid()`

Return the current process id.

Availability: Unix, Windows.

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.

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.

New in version 3.3.

os.PRIO_PROCESS

os.PRIO_PGRP

os.PRIO_USER

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

Availability: Unix.

New in version 3.3.

os.getresuid()

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

Availability: Unix.

New 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.

New in version 3.2.

os.getuid()

Return the current process's user id.

Availability: Unix.

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.

New in version 3.2.

os.putenv(key, value)

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

Availability: most flavors of Unix, Windows.

Note: On some platforms, including FreeBSD and Mac OS X, setting `environ` may cause memory leaks. Refer to the system documentation for `putenv`.

When `putenv()` is supported, 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`.

os.setegid(egid)

Set the current process's effective group id.

Availability: Unix.

os.seteuid(euid)

Set the current process's effective user id.

Availability: Unix.

os.setgid(gid)

Set the current process' group id.

Availability: Unix.

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.

Note: On Mac OS X, 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.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.

`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.

`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

New in version 3.3.

`os.setregid(rgid, egid)`

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

Availability: Unix.

`os.setresgid(rgid, egid, sgid)`

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

Availability: Unix.

New in version 3.2.

`os.setresuid(ruid, euid, suid)`

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

Availability: Unix.

New in version 3.2.

`os.setreuid(ruid, euid)`

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

Availability: Unix.

`os.getsid(pid)`

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

Availability: Unix.

`os.setsid()`

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

Availability: Unix.

`os.setuid(uid)`

Set the current process's user id.

Availability: Unix.

`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.

Availability: Unix, Windows.

`os.supports_bytes_environ`

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

New in version 3.2.

`os.umask(mask)`

Set the current numeric umask and return the previous umask.

Availability: Unix, Windows.

`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())`.

Availability: recent flavors of 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()`.

When `unsetenv()` is supported, 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`.

Availability: most flavors of Unix, Windows.

16.1.3. File Object Creation

This function creates 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.

16.1.4. 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*.

Availability: Unix, Windows.

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
```

Availability: Unix, Windows.

`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`.

`os.dup(fd)`

Return a duplicate of file descriptor *fd*.

Availability: Unix, Windows.

`os.dup2(fd, fd2)`

Duplicate file descriptor *fd* to *fd2*, closing the latter first if necessary.

Availability: Unix, Windows.

`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)`.

Availability: Unix.

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)`.

Availability: Unix.

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*)

Return status for file descriptor *fd*, like `stat()`. As of Python 3.3, this is equivalent to `os.stat(fd)`.

Availability: Unix, Windows.

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)`.

Availability: Unix.

`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.

Availability: Unix.

New 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.

New in version 3.3.

`os.lseek(fd, pos, how)`

Set the current position of file descriptor *fd* to position *pos*, modified by *how*. `SEEK_SET` or 0 to set the position relative to the beginning of the file; `SEEK_CUR` or 1

to set it relative to the current position; `SEEK_END` or 2 to set it relative to the end of the file. Return the new cursor position in bytes, starting from the beginning.

Availability: Unix, Windows.

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

Parameters to the `lseek()` function. Their values are 0, 1, and 2, respectively.

Availability: Unix, Windows.

New in version 3.3: Some operating systems could support additional values, like `os.SEEK_HOLE` or `os.SEEK_DATA`.

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

Open the file *file* 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.

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 this module too (see [open\(\) flag constants](#)). In particular, on Windows adding `O_BINARY` is needed to open files in binary mode.

This function can support [paths relative to directory descriptors](#).

Availability: Unix, Windows.

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()`.

New in version 3.3: The *dir_fd* argument.

- os . **openpty**()

Open a new pseudo-terminal pair. Return a pair of file descriptors (*master*, *slave*) for the pty and the tty, respectively. For a (slightly) more portable approach, use the [pty](#) module.

Availability: some flavors of Unix.

- os . **pipe**()

Create a pipe. Return a pair of file descriptors (*r*, *w*) usable for reading and writing, respectively.

Availability: Unix, Windows.

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: some flavors of Unix.

New 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.

New 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.

New 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.

New in version 3.3.

os.**pread**(*fd*, *buffer_size*, *offset*)

Read from a file descriptor, *fd*, at a position of *offset*. It will read up to *buffer_size* number of bytes. The file offset remains unchanged.

Availability: Unix.

New in version 3.3.

`os.pwrite(fd, string, offset)`

Write *string* to a file descriptor, *fd*, from *offset*, leaving the file offset unchanged.

Availability: Unix.

New in version 3.3.

`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.

Availability: Unix, Windows.

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.

`os.sendfile(out, in, offset, nbytes)`

`os.sendfile(out, in, offset, nbytes, headers=None, trailers=None, flags=0)`

Copy *nbytes* bytes from file descriptor *in* to file descriptor *out* 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* and the position of *in* is updated.

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

On Mac OS X and FreeBSD, a value of 0 for *nbytes* specifies to send until the end of *in* is reached.

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

Availability: Unix.

New in version 3.3.

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

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

Availability: Unix.

New in version 3.3.

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

Read from a file descriptor *fd* into a number of mutable *bytes-like objects buffers*. `readv()` will 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. `readv()` returns the total number of bytes read (which may be less than the total capacity of all the objects).

Availability: Unix.

New 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.

- 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.

- 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.**write**(*fd*, *str*)

Write the bytestring in *str* to file descriptor *fd*. Return the number of bytes actually written.

Availability: Unix, Windows.

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.
```

os.writev(fd, buffers)

Write the contents of *buffers* to file descriptor *fd*. *buffers* must be a sequence of *bytes-like objects*. `writev()` writes the contents of each object to the file descriptor and returns the total number of bytes written.

Availability: Unix.

New in version 3.3.

16.1.4.1. open() flag constants

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
```

These 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_SHLOCK
os.O_EXLOCK
os.O_CLOEXEC
```

These 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
```

These constants are only available on Windows.

```
os.O_ASYNC
os.O_DIRECT
os.O_DIRECTORY
os.O_NOFOLLOW
os.O_NOATIME
```

These constants are GNU extensions and not present if they are not defined by the C library.

```
os.RTLD_LAZY
os.RTLD_NOW
os.RTLD_GLOBAL
os.RTLD_LOCAL
os.RTLD_NODELETE
os.RTLD_NOLOAD
os.RTLD_DEEPBIND
```

See the Unix manual page *dlopen(3)*.

New in version 3.3.

16.1.4.2. Querying the size of a terminal

New 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.

16.1.5. Files and Directories

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

- **specifying a file descriptor:** For some functions, the *path* argument can be not only a string giving a path name, but also a file descriptor. The function will then operate on the file referred to by the descriptor. (For POSIX systems, Python will call the `f...` version of the function.)

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

If the function also supports *dir_fd* or *follow_symlinks* arguments, it is 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 `...at` or `f...at` version of the function.)

You can check whether or not *dir_fd* is supported on your platform using `os.supports_dir_fd`. If it is 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 instead of the file the link points to. (For POSIX systems, Python will call the `l...` version of the function.)

You can check whether or not *follow_symlinks* is supported on your platform using `os.supports_follow_symlinks`. If it is 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`.

Availability: Unix, Windows.

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.

`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.

Availability: Unix, Windows.

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

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*.

Availability: Unix.

New in version 3.3: The *follow_symlinks* argument.

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*.

Availability: Unix, Windows.

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.

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

`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.

Availability: Unix.

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

`os.chroot(path)`

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

Availability: Unix.

`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)`.

Availability: Unix.

`os.getcwd()`

Return a string representing the current working directory.

Availability: Unix, Windows.

`os.getcwdb()`

Return a bytestring representing the current working directory.

Availability: Unix, 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)`.

Availability: Unix.

`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)`.

Availability: Unix.

`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)`.

Availability: Unix.

`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*.

Availability: Unix, Windows.

Changed in version 3.2: Added Windows support.

New in version 3.3: Added the `src_dir_fd`, `dst_dir_fd`, and `follow_symlinks` arguments.

`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.

path may be either of type `str` or of type `bytes`. If *path* is of type `bytes`, 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.

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

Availability: Unix, Windows.

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

New in version 3.3: Added support for specifying an open file descriptor for *path*.

`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. 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](#).

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

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

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

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

On some systems, *mode* is ignored. Where it is used, the current umask value is first masked out. If the directory already exists, `OSError` is raised.

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.

Availability: Unix, Windows.

New in version 3.3: The `dir_fd` argument.

`os.makedirs(path, mode=0o777, exist_ok=False)`

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

The default `mode` is `0o777` (octal). On some systems, `mode` is ignored. Where it is used, the current umask value is first masked out.

If `exist_ok` is `False` (the default), an `OSError` is raised if the target directory already exists. If `exist_ok` is `True` an `OSError` is still raised if the umask-masked `mode` is different from the existing mode, on systems where the mode is used. `OSError` will also be raised if the directory creation fails.

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

This function handles UNC paths correctly.

New in version 3.2: The `exist_ok` parameter.

`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.

New in version 3.3: The `dir_fd` argument.

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

Create a filesystem node (file, device special file or named pipe) named `filename`. `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*.

New in version 3.3: The *dir_fd* argument.

`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.

`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, the result will also be a string object, and the call may raise an `UnicodeDecodeError`. If the *path* is a bytes object, the result will be a bytes object.

This function can also support *paths relative to directory descriptors*.

Availability: Unix, Windows

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

New in version 3.3: The *dir_fd* argument.

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

Remove (delete) the file *path*. If *path* is a directory, `OSError` is raised. Use `rmdir()` to remove directories.

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 identical to `unlink()`.

Availability: Unix, Windows.

New in version 3.3: The *dir_fd* argument.

`os.removedirs(path)`

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.

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

Rename the file or directory *src* to *dst*. If *dst* is a directory, `OSError` will be raised. On Unix, if *dst* exists and is a file, it 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). On Windows, if *dst* already exists, `OSError` will be raised even if it is a file.

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()`.

Availability: Unix, Windows.

New in version 3.3: The `src_dir_fd` and `dst_dir_fd` arguments.

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.

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

Rename the file or directory *src* to *dst*. If *dst* is a 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*.

Availability: Unix, Windows.

New in version 3.3.

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

Remove (delete) the directory *path*. Only works when the directory is empty, otherwise, `OSError` is raised. In order to remove whole directory trees, `shutil.rmtree()` can be used.

This function can support *paths relative to directory descriptors*.

Availability: Unix, Windows.

New in version 3.3: The `dir_fd` parameter.

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

Perform the equivalent of a `stat()` system call on the given path. *path* may be specified as either a string or as an open file descriptor. (This function normally follows

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

The return value is an object whose attributes correspond roughly to the members of the `stat` structure, namely:

- `st_mode` - protection bits,
- `st_ino` - inode number,
- `st_dev` - device,
- `st_nlink` - number of hard links,
- `st_uid` - user id of owner,
- `st_gid` - group id of owner,
- `st_size` - size of file, in bytes,
- `st_atime` - time of most recent access expressed in seconds,
- `st_mtime` - time of most recent content modification expressed in seconds,
- `st_ctime` - platform dependent; time of most recent metadata change on Unix, or the time of creation on Windows, expressed in seconds
- `st_atime_ns` - time of most recent access expressed in nanoseconds as an integer,
- `st_mtime_ns` - time of most recent content modification expressed in nanoseconds as an integer,
- `st_ctime_ns` - platform dependent; time of most recent metadata change on Unix, or the time of creation on Windows, expressed in nanoseconds as an integer

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

- `st_blocks` - number of 512-byte blocks allocated for file
- `st_blksize` - filesystem blocksize for efficient file system I/O
- `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
- `st_birthtime` - time of file creation

On Mac OS systems, the following attributes may also be available:

- `st_rsize`
- `st_creator`
- `st_type`

Note: The exact meaning and resolution of the `st_atime`, `st_mtime`, and `st_ctime` attributes depend on the operating system and the file system. For example, on Windows systems using the FAT or 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`, and `st_ctime_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`, and `st_ctime` 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`, and `st_ctime_ns`.

For backward compatibility, the return value of `stat()` 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.

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

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.)

Example:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
posix.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=1297230,
st_mtime=1297230027, st_ctime=1297230027)
>>> statinfo.st_size
264
```

Availability: Unix, Windows.

New in version 3.3: Added the `dir_fd` and `follow_symlinks` arguments, specifying a file descriptor instead of a path, and the `st_atime_ns`, `st_mtime_ns`, and `st_ctime_ns` members.

`os.stat_float_times([newvalue])`

Determine whether `stat_result` represents time stamps as float objects. If *newvalue* is `True`, future calls to `stat()` return floats, if it is `False`, future calls return ints. If *newvalue* is omitted, return the current setting.

For compatibility with older Python versions, accessing `stat_result` as a tuple always returns integers.

Python now returns float values by default. Applications which do not work correctly with floating point time stamps can use this function to restore the old behaviour.

The resolution of the timestamps (that is the smallest possible fraction) depends on the system. Some systems only support second resolution; on these systems, the fraction will always be zero.

It is recommended that this setting is only changed at program startup time in the `__main__` module; libraries should never change this setting. If an application uses a library that works incorrectly if floating point time stamps are processed, this application should turn the feature off until the library has been corrected.

Deprecated since version 3.3.

`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`.

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.

This function can support [specifying a file descriptor](#).

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

Availability: Unix.

New in version 3.3: Added support for specifying an open file descriptor for `path`.

`os.supports_dir_fd`

A `Set` object indicating which functions in the `os` module permit use of their `dir_fd` parameter. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency's sakes, functions that support `dir_fd` always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits use of its `dir_fd` parameter, use the `in` operator on `supports_dir_fd`. As an example, this expression determines whether the `dir_fd` parameter of `os.stat()` is locally available:

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


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

New in version 3.3.

`os.supports_effective_ids`

A `Set` object indicating which functions in the `os` module permit use of the `effective_ids` parameter for `os.access()`. If the local platform supports it, the collection will contain `os.access()`, otherwise it will be empty.

To check whether you can use the `effective_ids` parameter for `os.access()`, use the `in` operator on `supports_dir_fd`, like so:

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

Currently `effective_ids` only works on Unix platforms; it does not work on Windows.

New 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. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency's sakes, functions that support `fd` always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check 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 determines whether `os.chdir()` accepts open file descriptors when called on your local platform:

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

New in version 3.3.

`os.supports_follow_symlinks`

A `Set` object indicating which functions in the `os` module permit use of their `follow_symlinks` parameter. Different platforms provide different functionality, and an option that might work on one might be unsupported on another. For consistency's sakes, functions that support `follow_symlinks` always allow specifying the parameter, but will raise an exception if the functionality is not actually available.

To check whether a particular function permits use of its `follow_symlinks` parameter, use the `in` operator on `supports_follow_symlinks`. As an example, this expression determines whether the `follow_symlinks` parameter of `os.stat()` is

locally available:

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

New in version 3.3.

`os.symlink(source, link_name, target_is_directory=False, *, dir_fd=None)`

Create a symbolic link pointing to *source* named *link_name*.

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.

Symbolic link support was introduced in Windows 6.0 (Vista). `symlink()` will raise a `NotImplementedError` on Windows versions earlier than 6.0.

This function can support *paths relative to directory descriptors*.

Note: On Windows, the `SeCreateSymbolicLinkPrivilege` is required in order to successfully create symlinks. This privilege is not typically granted to regular users but is available to accounts which can escalate privileges to the administrator level. Either obtaining the privilege or running your application as an administrator are ways to successfully create symlinks.

`OSError` is raised when the function is called by an unprivileged user.

Availability: Unix, Windows.

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

New in version 3.3: Added the *dir_fd* argument, and now allow *target_is_directory* on non-Windows platforms.

`os.sync()`

Force write of everything to disk.

Availability: Unix.

New 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*.

Availability: Unix.

New in version 3.3.

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

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

Availability: Unix, Windows.

New in version 3.3: The *dir_fd* parameter.

`os.utime(path, times=None, *, ns=None, 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 not `None`, 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* and *ns* are both `None`, 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*.

Whether a directory can be given for *path* depends on whether the operating system implements directories as files (for example, Windows does not). 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*.

Availability: Unix, Windows.

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

`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* (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)`.

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).

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` is ineffective, because in bottom-up mode the directories in *dirnames* are generated before *dirpath* itself is generated.

By default, errors from the `listdir()` 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 CVS subdirectory:

```
import os
```

```

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

```

import os
for root, dirs, files, rootfd in os.fwalk('python/Lib/email'):
    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 'CVS' in dirs:
    dirs.remove('CVS')    # don't visit CVS 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)
```

Availability: Unix.

New in version 3.3.

16.1.5.1. Linux extended attributes

New 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. If it is str, it is encoded with the filesystem encoding.

This function can support [specifying a file descriptor](#) and [not following symlinks](#).

`os.listdirxattr(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](#).

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

Removes the extended filesystem attribute *attribute* from *path*. *attribute* should be bytes or str. If it is a string, it is encoded with the filesystem encoding.

This function can support [specifying a file descriptor](#) and [not following symlinks](#).

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. If it is a str, it is encoded with the filesystem encoding. *flags* may be `XATTR_REPLACE` or `XATTR_CREATE`. If `XATTR_REPLACE` is given and the attribute does not exist, `EXISTS` will be raised. If `XATTR_CREATE` is given and the attribute already exists, the attribute will not be created and `ENODATA` 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.

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.

16.1.6. 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()`.

Availability: Unix, Windows.


```
os.execl(path, arg0, arg1, ...)  
os.execle(path, arg0, arg1, ..., env)  
os.execlp(file, arg0, arg1, ...)  
os.execipe(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()`, `execipe()`, `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.

For `execle()`, `execipe()`, `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`.

Availability: Unix, Windows.

New in version 3.3: Added support for specifying an open file descriptor for *path* for `execve()`.

os.**_exit**(*n*)

Exit the process with status *n*, without calling cleanup handlers, flushing stdio buffers, etc.

Availability: Unix, Windows.

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.

Availability: Unix.

os.**EX_USAGE**

Exit code that means the command was used incorrectly, such as when the wrong number of arguments are given.

Availability: Unix.

os.**EX_DATAERR**

Exit code that means the input data was incorrect.

Availability: Unix.

os.**EX_NOINPUT**

Exit code that means an input file did not exist or was not readable.

Availability: Unix.

os.**EX_NOUSER**

Exit code that means a specified user did not exist.

Availability: Unix.

os . **EX_NOHOST**

Exit code that means a specified host did not exist.

Availability: Unix.

os . **EX_UNAVAILABLE**

Exit code that means that a required service is unavailable.

Availability: Unix.

os . **EX_SOFTWARE**

Exit code that means an internal software error was detected.

Availability: Unix.

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.

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.

os . **EX_CANTCREAT**

Exit code that means a user specified output file could not be created.

Availability: Unix.

os . **EX_IOERR**

Exit code that means that an error occurred while doing I/O on some file.

Availability: Unix.

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.

os . **EX_PROTOCOL**

Exit code that means that a protocol exchange was illegal, invalid, or not understood.

Availability: Unix.

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.

os . **EX_CONFIG**

Exit code that means that some kind of configuration error occurred.

Availability: Unix.

os . **EX_NOTFOUND**

Exit code that means something like “an entry was not found”.

Availability: Unix.

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, Cygwin and OS/2 EMX have known issues when using `fork()` from a thread.

Warning: See [ssl](#) for applications that use the SSL module with `fork()`.

Availability: Unix.

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.

Availability: some flavors of Unix.

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`. The Windows version of `kill()` additionally takes process

handles to be killed.

See also `signal.thread_kill()`.

New in version 3.2: Windows support.

os.**killpg**(*pgid*, *sig*)

Send the signal *sig* to the process group *pgid*.

Availability: Unix.

os.**nice**(*increment*)

Add *increment* to the process's "niceness". Return the new niceness.

Availability: Unix.

os.**plock**(*op*)

Lock program segments into memory. The value of *op* (defined in `<sys/lock.h>`) determines which segments are locked.

Availability: Unix.

os.**popen**(...)

Run child processes, returning opened pipes for communications. These functions are described in section [File Object Creation](#).

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.

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)
```

Availability: Unix, Windows. `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.

`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*])

Start a file with its associated application.

When *operation* is not specified or `'open'`, 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 `'print'` and `'edit'` (to be used on files) as well as `'explore'` and `'find'` (to be used on directories).

`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. If you want to use an absolute path, make sure the first character is not a slash (`'/'`); the underlying Win32 `ShellExecute()` function doesn’t work if it is. Use the `os.path.normpath()` function to ensure that the path is properly encoded for Win32.

Availability: Windows.

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.

On Unix, the return value is the exit status of the process encoded in the format specified for `wait()`. Note that POSIX does not specify the meaning of the return value of the C `system()` function, so the return value of the Python function is system-dependent.

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 preferable to using this function. See the [Replacing Older Functions with the subprocess Module](#) section in the `subprocess` documentation for some helpful recipes.

Availability: Unix, Windows.

`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)* or the corresponding Windows Platform API documentation. On Windows, only `user` and `system` are known; the other attributes are zero. On OS/2, only `elapsed` is 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.

Availability: Unix.

os.**waitid**(*idtype*, *id*, *options*)

Wait for the completion of one or more child processes. *idtype* can be `P_PID`, `P_PGID` or `P_ALL`. *id* specifies the pid to wait on. *options* is constructed from the ORing of one or more of `WEXITED`, `WSTOPPED` or `WCONTINUED` and additionally may be ORed with `WNOHANG` or `WNOWAIT`. The return value is an object representing the data contained in the `siginfo_t` structure, namely: `si_pid`, `si_uid`, `si_signo`, `si_status`, `si_code` or `None` if `WNOHANG` is specified and there are no children in a waitable state.

Availability: Unix.

New in version 3.3.

os.**P_PID**

os.**P_PGID**

os.**P_ALL**

These are the possible values for *idtype* in `waitid()`. They affect how *id* is interpreted.

Availability: Unix.

New in version 3.3.

os.**WEXITED**

os.**WSTOPPED**

os.**WNOWAIT**

Flags that can be used in *options* in `waitid()` that specify what child signal to wait for.

Availability: Unix.

New in version 3.3.

os.**CLD_EXITED**

os.**CLD_DUMPED**

os.**CLD_TRAPPED**

os.**CLD_CONTINUED**

These are the possible values for `si_code` in the result returned by `waitid()`.

Availability: Unix.

New in version 3.3.

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*).

An `OSError` is raised with the value of `errno` when the syscall returns -1.

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.

`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 option argument is the same as that provided to `waitpid()` and `wait4()`.

Availability: Unix.

`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()`.

Availability: Unix.

`os.WNOHANG`

The option for `waitpid()` to return immediately if no child process status is available immediately. The function returns `(0, 0)` in this case.

Availability: Unix.

`os.WCONTINUED`

This option causes child processes to be reported if they have been continued from a job control stop since their status was last reported.

Availability: some Unix systems.

`os.WUNTRACED`

This option causes child processes to be reported if they have been stopped but their current state has not been reported since they were stopped.

Availability: Unix.

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`.

Availability: Unix.

`os.WIFCONTINUED(status)`

Return `True` if the process has been continued from a job control stop, otherwise return `False`.

Availability: Unix.

`os.WIFSTOPPED(status)`

Return `True` if the process has been stopped, otherwise return `False`.

Availability: Unix.

`os.WIFSIGNALED(status)`

Return `True` if the process exited due to a signal, otherwise return `False`.

Availability: Unix.

`os.WIFEXITED(status)`

Return `True` if the process exited using the `exit(2)` system call, otherwise return `False`.

Availability: Unix.

`os.WEXITSTATUS(status)`

If `WIFEXITED(status)` is true, return the integer parameter to the `exit(2)` system call. Otherwise, the return value is meaningless.

Availability: Unix.

○ **os.WSTOPSIG(status)**

Return the signal which caused the process to stop.

Availability: Unix.

○ **os.WTERMSIG(status)**

Return the signal which caused the process to exit.

Availability: Unix.

16.1.7. 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.

New 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_IDLE**

Scheduling policy for extremely low priority background tasks.

○ **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 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 a 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.

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* (or the current process if zero) is restricted to.

See also: `multiprocessing.cpu_count()` returns the number of CPUs in the system.

16.1.8. 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.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.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.

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`.

16.1.9. Miscellaneous Functions

`os.urandom(n)`

Return a string of *n* 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 a Unix-like system this will query `/dev/urandom`, and on Windows it will use `CryptGenRandom()`. If a randomness source is not found, `NotImplementedError` will be raised.

For an easy-to-use interface to the random number generator provided by your platform, please see `random.SystemRandom`.