6.1.4 Files and Directories

access(*path, mode*)
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. Availability: Macintosh, UNIX, Windows.

F_OK
Value to pass as the *mode* parameter of `access()` to test the existence of *path*.

R_OK
Value to include in the *mode* parameter of `access()` to test the readability of *path*.

W_OK
Value to include in the *mode* parameter of `access()` to test the writability of *path*.

X_OK
Value to include in the *mode* parameter of `access()` to determine if *path* can be executed.

chdir(*path*)
Change the current working directory to *path*. Availability: Macintosh, UNIX, Windows.

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. Availability: UNIX. New in version 2.3.

getcwd()
Return a string representing the current working directory. Availability: Macintosh, UNIX, Windows.

getcwdu()
Return a Unicode object representing the current working directory. Availability: Macintosh, UNIX, Windows. New in version 2.3.

chroot(*path*)
Change the root directory of the current process to *path*. Availability: Macintosh, UNIX. New in version 2.2.

chmod(*path, mode*)
Change the mode of *path* to the numeric *mode*. *mode* may take one of the following values (as defined in the `stat` module):

- `S_ISUID`
- `S_ISGID`
- `S_ENFMT`
- `S_ISVTX`
- `S_IREAD`
- `S_IWRITE`
- `S_IEXEC`
- `S_IRWXU`
- `S_IRUSR`
- `S_IWUSR`
- `S_IXUSR`
- `S_IRWXG`
- `S_IRGRP`
- `S_IWGRP`

- S_IXGRP
- S_IRWXO
- S_IROTH
- S_IWOTH
- S_IXOTH

Availability: Macintosh, UNIX, Windows.

chown(*path, uid, gid*)
Change the owner and group id of *path* to the numeric *uid* and *gid*. Availability: Macintosh, UNIX.

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. Availability: Macintosh, UNIX. New in version 2.3.

link(*src, dst*)
Create a hard link pointing to *src* named *dst*. Availability: Macintosh, UNIX.

listdir(*path*)
Return a list containing the names of the entries in the directory. The list is in arbitrary order. It does not include the special entries `'.'` and `'..'` even if they are present in the directory. Availability: Macintosh, UNIX, Windows.

Changed in version 2.3: On Windows NT/2k/XP and Unix, if *path* is a Unicode object, the result will be a list of Unicode objects..

lstat(*path*)
Like `stat()`, but do not follow symbolic links. Availability: Macintosh, UNIX.

mkfifo(*path*[, *mode*])
Create a FIFO (a named pipe) named *path* with numeric mode *mode*. The default *mode* is `0666` (octal). The current umask value is first masked out from the mode. Availability: Macintosh, UNIX.

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.

mknod(*path*[, *mode=0600, device*])
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 S_IFREG, S_IFCHR, S_IFBLK, and S_IFIFO (those constants are available in `stat`). For S_IFCHR and S_IFBLK, *device* defines the newly created device special file (probably using `os.makedev()`), otherwise it is ignored. New in version 2.3.

major(*device*)
Extracts a device major number from a raw device number. New in version 2.3.

minor(*device*)
Extracts a device minor number from a raw device number. New in version 2.3.

makedev(*major, minor*)
Composes a raw device number from the major and minor device numbers. New in version 2.3.

mkdir(*path*[, *mode*])
Create a directory named *path* with numeric mode *mode*. The default *mode* is `0777` (octal). On some systems, *mode* is ignored. Where it is used, the current umask value is first masked out. Availability: Macintosh, UNIX, Windows.

makedirs(*path*[, *mode*])

Recursive directory creation function. Like `mkdir()`, but makes all intermediate-level directories needed to contain the leaf directory. Throws an `error` exception if the leaf directory already exists or cannot be created. The default *mode* is `0777` (octal). This function does not properly handle UNC paths (only relevant on Windows systems; Universal Naming Convention paths are those that use the `\\host\path' syntax). New in version 1.5.2.

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. Availability: Macintosh, UNIX.

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.

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: Macintosh, UNIX.

readlink(*path*)
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*`)`. Availability: Macintosh, UNIX.

remove(*path*)
Remove the file *path*. If *path* is a directory, `OSError` is raised; see `rmdir()` below to remove a directory. This is identical to the `unlink()` function documented below. 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. Availability: Macintosh, UNIX, Windows.

removedirs(*path*)
Removes directories recursively. Works like `rmdir()` except that, if the leaf directory is successfully removed, directories corresponding to rightmost path segments will be pruned way until either the whole path is consumed or an error is raised (which is ignored, because it generally means that a parent directory is not empty). Throws an `error` exception if the leaf directory could not be successfully removed. New in version 1.5.2.

rename(*src, dst*)
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 removed 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; there may be no way to implement an atomic rename when *dst* names an existing file. Availability: Macintosh, UNIX, Windows.

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()`. New in version 1.5.2.

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

rmdir(*path*)
Remove the directory *path*. Availability: Macintosh, UNIX, Windows.

stat(*path*)

Perform a `stat()` system call on the given path. The return value is an object whose attributes correspond 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), `st_mtime` (time of most recent content modification), `st_ctime` (platform dependent; time of most recent metadata change on UNIX, or the time of creation on Windows).

Changed in version 2.3: If `stat_float_times` returns true, the time values are floats, measuring seconds. Fractions of a second may be reported if the system supports that. On Mac OS, the times are always floats. See `stat_float_times` for further discussion. .

On some Unix systems (such as Linux), the following attributes may also be available: `st_blocks` (number of blocks allocated for file), `st_blksize` (filesystem blocksize), `st_rdev` (type of device if an inode device).

On Mac OS systems, the following attributes may also be available: `st_rsize`, `st_creator`, `st_type`.

On RISCOS systems, the following attributes are also available: `st_ftype` (file type), `st_attrs` (attributes), `st_obtype` (object type).

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. 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.)

Note: The exact meaning and resolution of the `st_atime`, `st_mtime`, and `st_ctime` members depends 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.

Availability: Macintosh, UNIX, Windows.

Changed in version 2.2: Added access to values as attributes of the returned object.

stat_float_times([*newvalue*])

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

For compatibility with older Python versions, accessing `stat_result` as a tuple always returns integers. For compatibility with Python 2.2, accessing the time stamps by field name also returns integers. Applications that want to determine the fractions of a second in a time stamp can use this function to have time stamps represented as floats. Whether they will actually observe non-zero fractions depends on the system.

Future Python releases will change the default of this setting; applications that cannot deal with floating point time stamps can then use this function to turn the feature off.

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.

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_frsize`, `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`. Availability: UNIX.

For backward compatibility, the return value is also accessible as a tuple whose values correspond to the attributes, in the order given above. The standard module `statvfs` defines constants that are useful for extracting information from a `statvfs` structure when accessing it as a sequence; this remains useful when writing code that needs to work with versions of Python that don't support accessing the fields as attributes.

Changed in version 2.2: Added access to values as attributes of the returned object.

symlink(*src, dst*)
Create a symbolic link pointing to *src* named *dst*. Availability: UNIX.

tempnam([*dir*[, *prefix*]])
Return a unique path name that is reasonable for creating a temporary file. This will be an absolute path that names a potential directory entry in the directory *dir* or a common location for temporary files if *dir* is omitted or `None`. If given and not `None`, *prefix* is used to provide a short prefix to the filename. Applications are responsible for properly creating and managing files created using paths returned by `tempnam()`; no automatic cleanup is provided. On UNIX, the environment variable TMPDIR overrides *dir*, while on Windows the TMP is used. The specific behavior of this function depends on the C library implementation; some aspects are underspecified in system documentation. **Warning:** Use of `tempnam()` is vulnerable to symlink attacks; consider using `tmpfile()` instead. Availability: Macintosh, UNIX, Windows.

tmpnam()
Return a unique path name that is reasonable for creating a temporary file. This will be an absolute path that names a potential directory entry in a common location for temporary files. Applications are responsible for properly creating and managing files created using paths returned by `tmpnam()`; no automatic cleanup is provided. **Warning:** Use of `tmpnam()` is vulnerable to symlink attacks; consider using `tmpfile()` instead. Availability: UNIX, Windows. This function probably shouldn't be used on Windows, though: Microsoft's implementation of `tmpnam()` always creates a name in the root directory of the current drive, and that's generally a poor location for a temp file (depending on privileges, you may not even be able to open a file using this name).

TMP_MAX
The maximum number of unique names that `tmpnam()` will generate before reusing names.

unlink(*path*)
Remove the file *path*. This is the same function as `remove()`; the `unlink()` name is its traditional UNIX name. Availability: Macintosh, UNIX, Windows.

utime(*path, times*)
Set the access and modified times of the file specified by *path*. If *times* is `None`, then the file's access and modified times are set to the current time. Otherwise, *times* must be a 2-tuple of numbers, of the form `(`*atime*`,` *mtime*`)` which is used to set the access and modified times, respectively. 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()`. Changed in version 2.0: Added support for `None` for *times*. Availability: Macintosh, UNIX, Windows.

walk(*top*[, *topdown*=`True` [, *onerror*=`None`]])
`walk()` generates 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 *dirnames* itself is generated.

By default errors from the `os.listdir()` call are ignored. If optional argument *onerror* is specified, it should be a function; it will be called with one argument, an os.error 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.

> **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.

> **Note:** On systems that support symbolic links, links to subdirectories appear in *dirnames* lists, but `walk()` will not visit them (infinite loops are hard to avoid when following symbolic links). To visit linked directories, you can identify them with `os.path.islink(`*path*`)`, and invoke `walk(`*path*`)` on each directly.

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",
    print sum(getsize(join(root, name)) for name in files),
    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))
```

New in version 2.3.

Release 2.4.1, documentation updated on 30 March 2005.
See About this document... for information on suggesting changes.