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
therefore this routine can be used in a suid/sgid environment to test if the invoking user has th
access to *path*. *mode* should be `F_OK` to test the existence of *path*, or it can be the inclusive OR
of `R_OK`, `W_OK`, and `X_OK` to test permissions. Return `1` if access is allowed, `0` if not. See the UN
access(2) for more information. Availability: 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*. Th 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, Wi

`getcwdu`()
Return a Unicode object representing the current working directory. Availability: UNIX, Winc version 2.3.

`chroot`(*path*)
Change the root directory of the current process to *path*. Availability: UNIX. New in version 2

`chmod`(*path, mode*)
Change the mode of *path* to the numeric *mode*. *mode* may take one of the following values (as `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: UNIX, Windows.

`chown`(*path, uid, gid*)

Change the owner and group id of *path* to the numeric *uid* and *gid*. Availability: UNIX.

`lchown`(*path, uid, gid*)
Change the owner and group id of *path* to the numeric *uid* and gid. This function will not follo
links. Availability: UNIX. New in version 2.3.

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

`listdir`(*path*)
Return a list containing the names of the entries in the directory. The list is in arbitrary order.
include the special entries `'.'` and `'..'` even if they are present in the directory. Availability:
UNIX, Windows.

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

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

`mkfifo`(*path*[, *mode*])
Create a FIFO (a named pipe) named *path* with numeric mode *mode*. The default *mode* is `066`
current umask value is first masked out from the mode. Availability: UNIX.

FIFOs are pipes that can be accessed like regular files. FIFOs exist until they are deleted (for e
`os.unlink()`). Generally, FIFOs are used as rendezvous between ``client'' and ``server'' type
server opens the FIFO for reading, and the client opens it for writing. Note that `mkfifo()` doe
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* speci
permissions to use and the type of node to be created, being combined (bitwise OR) with one
S_IFCHR, S_IFBLK, and S_IFIFO (those constants are available in `stat`). For S_IFCHR and
device defines the newly created device special file (probably using `os.makedev()`), otherwis
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.

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

`makedirs`(*path*[, *mode*])
: Recursive directory creation function. Like `mkdir()`, but makes all intermediate-level directo contain the leaf directory. Throws an `error` exception if the leaf directory already exists or ca The default *mode* is `0777` (octal). This function does not properly handle UNC paths (only rel Windows systems; Universal Naming Convention paths are those that use the ``\\host\path`' version 1.5.2.

`pathconf`(*path, name*)
: Return system configuration information relevant to a named file. *name* specifies the configu retrieve; it may be a string which is the name of a defined system value; these names are spec number of standards (POSIX.1, UNIX 95, UNIX 98, and others). Some platforms define additi well. The names known to the host operating system are given in the `pathconf_names` dictio configuration variables not included in that mapping, passing an integer for *name* is also acce Availability: UNIX.

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

`pathconf_names`

Dictionary mapping names accepted by `pathconf()` and `fpathconf()` to the integer values d names by the host operating system. This can be used to determine the set of names known to Availability: UNIX.

`readlink`(*path*)

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

`remove`(*path*)

Remove the file *path*. If *path* is a directory, `OSError` is raised; see `rmdir()` below to remove is identical to the `unlink()` function documented below. On Windows, attempting to remove use causes an exception to be raised; on UNIX, the directory entry is removed but the storage file is not made available until the original file is no longer in use. Availability: Macintosh, U

`removedirs`(*path*)

Removes directories recursively. Works like `rmdir()` except that, if the leaf directory is succe removed, directories corresponding to rightmost path segments will be pruned way until eithe is consumed or an error is raised (which is ignored, because it generally means that a parent d empty). Throws an `error` exception if the leaf directory could not be successfully removed. N 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, is a file, it will be removed silently if the user has permission. The operation may fail on some *src* and *dst* are on different filesystems. If successful, the renaming will be an atomic operatio POSIX requirement). On Windows, if *dst* already exists, `OSError` will be raised even if it is a be no way to implement an atomic rename when *dst* names an existing file. Availability: Mac Windows.

`renames`(*old, new*)

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

Note: this function can fail with the new directory structure made if you lack permissions nee the leaf directory or file. New in version 1.5.2.

`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 attribute:
the members of the `stat` structure, namely: `st_mode` (protection bits), `st_ino` (inode number
(device), `st_nlink` (number of hard links), `st_uid` (user ID of owner), `st_gid` (group ID of
`st_size` (size of file, in bytes), `st_atime` (time of most recent access), `st_mtime` (time of mc
content modification), `st_ctime` (time of most recent content modification or metadata chang

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

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

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

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

For backward compatibility, the return value of `stat()` is also accessible as a tuple of at least
giving the most important (and portable) members of the `stat` structure, in the order `st_mode`
`st_dev`, `st_nlink`, `st_uid`, `st_gid`, `st_size`, `st_atime`, `st_mtime`, `st_ctime`. More items ı
the end by some implementations. The standard module `stat` defines functions and constants
for extracting information from a `stat` structure. (On Windows, some items are filled with du
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, fu
() return floats, if it is False, future calls return ints. If newval is omitted, return the current set

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

Future Python releases will change the default of this setting; applications that cannot deal wi
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__* n
should never change this setting. If an application uses a library that works incorrectly if float
stamps are processed, this application should turn the feature off until the library has been cor

`statvfs`(*path*)

Perform a `statvfs()` system call on the given path. The return value is an object whose attrib the filesystem on the given path, and correspond to the members of the `statvfs` structure, nar `f_blocks`, `f_bfree`, `f_bavail`, `f_files`, `f_ffree`, `f_favail`, `f_flag`, `f_namemax`. Availabi

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

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 abs names a potential directory entry in the directory *dir* or a common location for temporary files omitted or `None`. If given and not `None`, *prefix* is used to provide a short prefix to the filename are responsible for properly creating and managing files created using paths returned by `temp` automatic cleanup is provided. On UNIX, the environment variable TMPDIR overrides *dir*, wl Windows the TMP is used. The specific behavior of this function depends on the C library im some aspects are underspecified in system documentation. **Warning:** Use of `tempnam()` is vu symlink attacks; consider using `tmpfile()` instead. Availability: UNIX, Windows.

`tmpnam`()

Return a unique path name that is reasonable for creating a temporary file. This will be an abs names a potential directory entry in a common location for temporary files. Applications are r properly creating and managing files created using paths returned by `tmpnam()`; no automatic provided. **Warning:** Use of `tmpnam()` is vulnerable to symlink attacks; consider using `tmpfi` Availability: UNIX, Windows. This function probably shouldn't be used on Windows, though: implementation of `tmpnam()` always creates a name in the root directory of the current drive, generally a poor location for a temp file (depending on privileges, you may not even be able t 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 traditi 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 modified times are set to the current time. Otherwise, *times* must be a 2-tuple of numbers, of t (*`atime, mtime`*) which is used to set the access and modified times, respectively. Changed ir 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 bot each directory in the tree rooted at directory *top* (including *top* itself), it yields a 3-tuple (*`dir`* *`dirnames, filenames`*).

dirpath is a string, the path to the directory. *dirnames* is a list of the names of the subdirectori (excluding `'.'` and `'..'`). *filenames* is a list of the names of the non-directory files in *dirpath* names in the lists contain no path components. To get a full path (which begins with *top*) to a in *dirpath*, do `os.path.join(`*`dirpath, name`*`)`.

If optional argument *topdown* is true or not specified, the triple for a directory is generated be for any of its subdirectories (directories are generated top down). If *topdown* is false, the triple 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 s assignment), and `walk()` will only recurse into the subdirectories whose names remain in *dirn* be used to prune the search, impose a specific order of visiting, or even to inform `walk()` abo the caller creates or renames before it resumes `walk()` again. Modifying *dirnames* when *topd* ineffective, because in bottom-up mode the directories in *dirnames* are generated before *dirna* generated.

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

Note: If you pass a relative pathname, don't change the current working directory between res `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 will not visit them (infinite loops are hard to avoid when following symbolic links). To visit li directories, you can identify them with `os.path.islink(`*`path`*`)`, and invoke `walk(`*`path`*`)` on

This example displays the number of bytes taken by non-directory files in each directory unde 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 before the directory is empty:

```
import os
from os.path import join
# Delete everything reachable from the directory named in 'top'.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(join(root, name))
    for name in dirs:
        os.rmdir(join(root, name))
```

New in version 2.3.