10. File and Text Operations

safaribooksonline.com/library/view/python-in-a/9781491913833/ch10.html

Filesystem Operations

Using the `os` module, you can manipulate the filesystem in a variety of ways: creating, copying, and deleting files and directories; comparing files; and examining filesystem information about files and directories. This section documents the attributes and methods of the `os` module that you use for these purposes, and covers some related modules that operate on the filesystem.

Path-String Attributes of the os Module

A file or directory is identified by a string, known as its *path*, whose syntax depends on the platform. On both Unix-like and Windows platforms, Python accepts Unix syntax for paths, with a slash (`/`) as the directory separator. On non-Unix-like platforms, Python also accepts platform-specific path syntax. On Windows, in particular, you may use a backslash (`\`) as the separator. However, you then need to double-up each backslash as `\\` in string literals, or use raw-string syntax as covered in "Literals"; you needlessly lose portability. Unix path syntax is handier and usable everywhere, so we strongly recommend that you *always* use it. In the rest of this chapter, we assume Unix path syntax in both explanations and examples.

The `os` module supplies attributes that provide details about path strings on the current platform. You should typically use the higher-level path manipulation operations covered in "The os.path Module" rather than lower-level string operations based on these attributes. However, the attributes may be useful at times.

`curdir`

> The string that denotes the current directory (`'.'` on Unix and Windows)

`defpath`

> The default search path for programs, used if the environment lacks a `PATH` environment variable

`linesep`

> The string that terminates text lines (`'\n'` on Unix; `'\r\n'` on Windows)

`extsep`

> The string that separates the extension part of a file's name from the rest of the name (`'.'` on Unix and Windows)

`pardir`

> The string that denotes the parent directory (`'..'` on Unix and Windows)

`pathsep`

> The separator between paths in lists of paths, such as those used for the environment variable `PATH` (`':'` on Unix; `';'` on Windows)

`sep`

> The separator of path components (`'/'` on Unix; `'\\'` on Windows)

Permissions

Unix-like platforms associate nine bits with each file or directory: three each for the file's owner, its group, and anybody else (AKA "the world"), indicating whether the file or directory can be read, written, and executed by the given subject. These nine bits are known as the file's *permission bits*, and are part of the file's *mode* (a bit string that includes other bits that describe the file). These bits are often displayed in octal notation, since that groups three bits per digit. For example, mode 0o664 indicates a file that can be read and written by its owner and group, and read—but not written—by anybody else. When any process on a Unix-like system creates a file or directory, the operating system applies to the specified mode a bit mask known as the process's *umask*, which can remove some of the permission bits.

Non-Unix-like platforms handle file and directory permissions in very different ways. However, the os functions that deal with file permissions accept a mode argument according to the Unix-like approach described in the previous paragraph. Each platform maps the nine permission bits in a way appropriate for it. For example, on versions of Windows that distinguish only between read-only and read/write files and do not distinguish file ownership, a file's permission bits show up as either 0o666 (read/write) or 0o444 (read-only). On such a platform, when creating a file, the implementation looks only at bit 0o200, making the file read/write when that bit is 1, read-only when 0.

File and Directory Functions of the os Module

The os module supplies several functions to query and set file and directory status. In all versions and platforms, the argument path to any of these functions can be a string giving the path of the file or directory involved. In v3 only, on some Unix platforms, some of the functions also support as argument path a *file descriptor* (AKA *fd*), an int denoting a file (as returned, for example, by os.open). In this case, the module attribute os.supports_fd is the set of functions of the os module that do support a file descriptor as argument path (the module attribute is missing in v2, and in v3 on platforms lacking such support).

In v3 only, on some Unix platforms, some functions support the optional, keyword-only argument follow_symlinks, defaulting to True. When true, and always in v2, if path indicates a symbolic link, the function follows it to reach an actual file or directory; when false, the function operates on the symbolic link itself. The module attribute os.supports_follow_symlinks, if present, is the set of functions of the os module that do support this argument.

In v3 only, on some Unix platforms, some functions support the optional, keyword-only argument dir_fd, defaulting to None. When present, path (if relative) is taken as being relative to the directory open at that file descriptor; when missing, and always in v2, path (if relative) is taken as relative to the current working directory. The module attribute os.supports_dir_fd, if present, is the set of functions of the os module that do support this argument.

Table 10-3. os module functions

access

`access(path, mode)`

Returns `True` when the file `path` has all of the permissions encoded in integer `mode`; otherwise, `False`. `mode` can be `os.F_OK` to test for file existence, or one or more of the constant integers named `os.R_OK`, `os.W_OK`, and `os.X_OK` (with the bitwise-OR operator `|` joining them, if more than one) to test permissions to read, write, and execute the file.

`access` does not use the standard interpretation for its `mode` argument, covered in "Permissions". Rather, `access` tests only if this specific process's real user and group identifiers have the requested permissions on the file. If you need to study a file's permission bits in more detail, see the function `stat`, covered in Table 10-4.

In v3 only, `access` supports an optional, keyword-only argument `effective_ids`, defaulting to `False`. When this argument is passed as true, `access` uses effective rather than real user and group identifiers.

chdir

`chdir(path)`

Sets the current working directory of the process to `path`.

chmod

`chmod(path, mode)`

Changes the permissions of the file `path`, as encoded in integer `mode`. `mode` can be zero or more of `os.R_OK`, `os.W_OK`, and `os.X_OK` (with the bitwise-OR operator `|` joining them, if more than one) for read, write, and execute (respectively). On Unix-like platforms, `mode` can be a richer bit pattern (as covered in "Permissions") to specify different permissions for user, group, and other, as well as other special bits defined in the module `stat` and listed in the online docs.

getcwd

`getcwd()`

Returns a `str`, the path of the current working directory. In v3, `getcwdb` returns the same value as `bytes`; in v2, `getcwdu` returns the same value as `unicode`.

link

`link(`*`src`*`,`*`dst`*`)`

Create a hard link named `dst`, pointing to `src`. In v2, this is only available on Unix platforms; in v3, it's also available on Windows.

listdir

`listdir(path)`

Returns a list whose items are the names of all files and subdirectories in the directory `path`. The list is in arbitrary order and does *not* include the special directory names `'.'` (current directory) and `'..'` (parent directory). See also the v3-only alternative function `scandir`, covered later in this table, which can offer performance improvements in some cases.

The v2-only `dircache` module also supplies a function named `listdir`, which works like `os.listdir`, with two enhancements. `dircache.listdir` returns a sorted list; and `dircache` caches the list, so that repeated requests for the same directory are faster if the directory's contents don't change. `dircache` automatically detects changes: when you call `dircache.listdir`, you get a list of the directory's contents at that time.

makedirs, mkdir	=0777) makedirs(path, modemkdir(path, mode=0777) makedirs creates all directories that are part of path and do not yet exist. mkdir creates only the rightmost directory of path and raises OSError if any of the previous directories in path do not exist. Both functions use mode as permission bits of directories they create. Both raise OSError when creation fails, or when a file or directory named path already exists.
remove, unlink	) remove(pathunlink(path) Removes the file named path (see rmdir in this table to remove a directory). unlink is a synonym of remove.
removedirs	removedirs(path) Loops from right to left over the directories that are part of path, removing each one. The loop ends when a removal attempt raises an exception, generally because a directory is not empty. removedirs does not propagate the exception, as long as it has removed at least one directory.
rename	rename(source, dest) Renames (i.e., moves) the file or directory named source to dest. If dest already exists, rename may either replace dest, or raise an exception; in v3 only, to guarantee replacement rather than exception, call, instead, the function os.replace.
renames	renames(source, dest) Like rename, except that renames tries to create all intermediate directories needed for dest. Also, after renaming, renames tries to remove empty directories from the path source using removedirs. It does not propagate any resulting exception; it's not an error if the starting directory of source does not become empty after the renaming.
rmdir	rmdir(path) Removes the empty directory named path (raises OSError if the removal fails, and, in particular, if the directory is not empty).

scandir `scandir(path)` *v3 only*

Returns an iterator over `os.DirEntry` instances representing each item in the path; using `scandir`, and calling each resulting instance's methods to determine its characteristics, can offer performance improvements compared to using `listdir` and `stat`, depending on the underlying platform.

> `class`
> `DirEntry`
>
> An instance `d` of class `DirEntry` supplies string attributes `name` and `path`, holding the item's base name and full path, respectively; and several methods, of which the most frequently used are the no-arguments, bool-returning methods `is_dir`, `is_file`, and `is_symlink`—`is_dir` and `is_file` by default follow symbolic links: pass named-only argument `follow_symlinks=False` to avoid this behavior. For more complete information, see the online docs. `d` avoids system calls as much as feasible, and, when it needs one, it caches the results; if you need information that's guaranteed to be up to date, call `os.stat(`*`d`*`.path)` and use the `stat_result` instance it returns (however, this sacrifices `scandir`'s potential performance improvements).

stat stat(path)

Returns a value x of type stat_result, which provides 10 items of information about the file or subdirectory path. Accessing those items by their numeric indices is possible but generally not advisable, because the resulting code is not very readable; use the corresponding attribute names instead. Table 10-4 lists the attributes of a stat_result instance and the meaning of corresponding items.

Table 10-4. Items (attributes) of a stat_result instance

Item index	Attribute name	Meaning
0	st_mode	Protection and other mode bits
1	st_ino	Inode number
2	st_dev	Device ID
3	st_nlink	Number of hard links
4	st_uid	User ID of owner
5	st_gid	Group ID of owner
6	st_size	Size in bytes
7	st_atime	Time of last access
8	st_mtime	Time of last modification
9	st_ctime	Time of last status change

For example, to print the size in bytes of file path, you can use any of:

```
import os
print(os.path.getsize(path))
print(os.stat(path)[6])
print(os.stat(path).st_size)
```

Time values are in seconds since the epoch, as covered in Chapter 12 (int on most platforms). Platforms unable to give a meaningful value for an item use a dummy value.

tempnam, tmpnam =None) tmpnam(
tempnam(dir=None, prefix)

Returns an absolute path usable as the name of a new temporary file.

Note: tempnam and tmpnam are weaknesses in your program's security. Avoid these functions and use instead the standard library module tempfile, covered in "The tempfile Module".

utime `utime(path, times=None)`

Sets the accessed and modified times of file or directory `path`. If `times` is `None`, `utime` uses the current time. Otherwise, `times` must be a pair of numbers (in seconds since the epoch, as covered in Chapter 12) in the order `(accessed, modified)`.

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

A generator yielding an item for each directory in the tree whose root is directory *`top`*. When *`topdown`* is `True`, the default, `walk` visits directories from the tree's root downward; when *`topdown`* is `False`, `walk` visits directories from the tree's leaves upward. When *`onerror`* is `None`, `walk` catches and ignores any `OSError` exception raised during the tree-walk. Otherwise, *`onerror`* must be a function; `walk` catches any `OSError` exception raised during the tree-walk and passes it as the only argument in a call to *`onerror`*, which may process it, ignore it, or `raise` it to terminate the tree-walk and propagate the exception.

Each item `walk` yields is a tuple of three subitems: *`dirpath`*, a string that is the directory's path; *`dirnames`*, a list of names of subdirectories that are immediate children of the directory (special directories '.' and '..' are *not* included); and `filenames`, a list of names of files that are directly in the directory. If *`topdown`* is `True`, you can alter list *`dirnames`* in-place, removing some items and/or reordering others, to affect the tree-walk of the subtree rooted at *`dirpath`*; `walk` iterates only on subdirectories left in *`dirnames`*, in the order in which they're left. Such alterations have no effect if *`topdown`* is `False` (in this case, `walk` has already visited all subdirectories by the time it visits the current directory and yields its item).

By default, `walk` does not walk down symbolic links that resolve to directories. To get such extra walking, pass `followlinks` as true, but beware: this can cause infinite looping if a symbolic link resolves to a directory that is its ancestor—`walk` doesn't take precautions against this anomaly.

The os.path Module

The `os.path` module supplies functions to analyze and transform path strings. To use this module, you can `import os.path`; however, even if you just `import os`, you can also access the `os.path` module and all of its attributes. The most commonly useful functions from the module are listed here:

abspath `abspath(path)`

Returns a normalized absolute path string equivalent to `path`, just like:

```
os.path.normpath(os.path.join(os.getcwd(), path))
```

For example, `os.path.abspath(os.curdir)` is the same as `os.getcwd()`.

basename `basename(path)`

Returns the base name part of `path`, just like `os.path.split(path)[1]`. For example, `os.path.basename('b/c/d.e')` returns `'d.e'`.

commonprefix

`commonprefix(list)`

Accepts a list of strings and returns the longest string that is a prefix of all items in the list. Unlike all other functions in `os.path`, `commonprefix` works on arbitrary strings, not just on paths. For example, `os.path.commonprefix('foobar', 'foolish')` returns `'foo'`.

In v3 only, function `os.path.commonpath` works similarly, but returns, specifically, only common prefix *paths*, not arbitrary *string* prefixes.

dirname

`dirname(path)`

Returns the directory part of `path`, just like `os.path.split(path)[0]`. For example, `os.path.dirname('b/c/d.e')` returns `'b/c'`.

exists, lexists

`)`
`exists(pathlexists(`*path*`)`

Returns `True` when `path` names an existing file or directory; otherwise, `False`. In other words,

`,`
`os.path.exists(x)` is the same as `os.access(xos.F_OK)` . `lexists` is the same, but also returns `True` when `path` names a symbolic link that indicates a nonexisting file or directory (sometimes known as a *broken symlink*), while `exists` returns `False` in such cases.

expandvars, expanduser

`)`
`expandvars(pathexpanduser(`*path*`)`

Returns a copy of string `path`, where each substring of the form `$name` or `${name}` is replaced with the value of environment variable `name`. For example, if environment variable `HOME` is set to `/u/alex`, the following code:

```
import os
print(os.path.expandvars('$HOME/foo/'))
```

emits `/u/alex/foo/`.

`os.path.expanduser` expands a leading `~` to the path of the home directory of the current user.

getatime, getmtime, getctime, getsize

`)` `)` `getctime(`*path*`)`
`getatime(pathgetmtime( pathgetsize( path)`

Each of these functions returns an attribute from the result of `os.stat(path)`: respectively, `st_atime`, `st_mtime`, `st_ctime`, and `st_size`. See Table 10-4 for more details about these attributes.

isabs

`isabs(path)`

Returns `True` when `path` is absolute. (A path is absolute when it starts with a slash (`/`), or, on some non-Unix-like platforms, with a drive designator followed by `os.sep`.) When `path` is not absolute, `isabs` returns `False`.

isfile

`isfile(path)`

Returns `True` when `path` names an existing regular file (however, `isfile` also follows symbolic links); otherwise, `False`.

isdir

`isdir(path)`

Returns `True` when `path` names an existing directory (however, `isdir` also follows symbolic links); otherwise, `False`.

islink

`islink(path)`

Returns `True` when `path` names a symbolic link; otherwise, `False`.

ismount

`ismount(path)`

Returns `True` when `path` names a mount point; otherwise, `False`.

join

`join(path, *paths)`

Returns a string that joins the argument strings with the appropriate path separator for the current platform. For example, on Unix, exactly one slash character `/` separates adjacent path components. If any argument is an absolute path, `join` ignores previous components. For example:

```
print(os.path.join('a/b', 'c/d', 'e/f'))# on Unix prints: a/b/c/d/e/f
print(os.path.join('a/b', '/c/d', 'e/f'))# on Unix prints: /c/d/e/f
```

The second call to `os.path.join` ignores its first argument `'a/b'`, since its second argument `'/c/d'` is an absolute path.

normcase

`normcase(path)`

Returns a copy of `path` with case normalized for the current platform. On case-sensitive filesystems (as is typical in Unix-like systems), `path` is returned unchanged. On case-insensitive filesystems (as is typical in Windows), all letters in the returned string are lowercase. On Windows, `normcase` also converts each `/` to a `\`.

normpath

`normpath(path)`

Returns a normalized pathname equivalent to `path`, removing redundant separators and path-navigation aspects. For example, on Unix, `normpath` returns `'a/b'` when `path` is any of `'a//b'`, `'a/./b'`, or `'a/c/../b'`. `normpath` makes path separators appropriate for the current platform. For example, on Windows, separators become `\`.

realpath

`realpath(path)`

Returns the actual path of the specified file or directory, resolving symlinks along the way.

relpath

`relpath(path, start=os.curdir)`

Returns a relative path to the specified file or directory, relative to directory `start` (by default, the process's current working directory).

samefile	samefile(*path1*, *path2*) Returns True if both arguments refer to the same file or directory.
sameopenfile	sameopenfile(*fd1*, *fd2*) Returns True if both file descriptor arguments refer to the same open file or directory.
samestat	samestat(*stat1*, *stat2*) Returns True if both arguments, instances of os.stat_result (typically, results of os.stat calls), refer to the same file or directory.
split	split(path) Returns a pair of strings (dir, base) such that join(dir, base) equals path. base is the last component and never contains a path separator. If path ends in a separator, base is ''. dir is the leading part of path, up to the last separator, shorn of trailing separators. For example, os.path.split('a/b/c/d') returns ('a/b/c', 'd').
splitdrive	splitdrive(path) Returns a pair of strings (*drv*,*pth*) such that *drv*+*pth* equals path. drv is either a drive specification, or ''—always '' on platforms not supporting drive specifications, such as Unix-like systems. On Windows, os.path.splitdrive('c:d/e') returns ('c:', 'd/e').
splitext	splitext(path) Returns a pair (root, ext) such that root+ext equals path. ext is either '' or starts with a '.' and has no other '.' or path separator. For example, os.path.splitext('a.a/b.c.d') returns the pair ('a.a/b.c', '.d').
walk	walk(path, func, arg) (v2 only) Calls func(arg, dirpath, namelist) for each directory in the tree whose root is the directory path, starting with path itself. This function is hard to use and obsolete; use, instead, generator os.walk, covered in Table 10-4, on both v2 and v3.

The stat Module

The function os.stat (covered in Table 10-4) returns instances of stat_result, whose item indices, attribute names, and meaning are also covered there. The stat module supplies attributes with names like those of stat_result's attributes, turned into uppercase, and corresponding values that are the corresponding item indices.

The more interesting contents of the stat module are functions to examine the st_mode attribute of a stat_result instance and determine the kind of file. os.path also supplies functions for such tasks, which operate directly on the file's path. The functions supplied by stat shown in the following list are faster when you perform several tests on the same file: they require only one os.stat call at the start of a series of tests, while the functions in os.path implicitly ask the operating system for the same information at each test. Each function returns True when mode denotes a file of the given kind; otherwise, False.

`S_ISDIR(mode)`

Is the file a directory?

`S_ISCHR(mode)`

Is the file a special device-file of the character kind?

`S_ISBLK(mode)`

Is the file a special device-file of the block kind?

`S_ISREG(mode)`

Is the file a normal file (not a directory, special device-file, and so on)?

`S_ISFIFO(mode)`

Is the file a FIFO (also known as a “named pipe”)?

`S_ISLNK(mode)`

Is the file a symbolic link?

`S_ISSOCK(mode)`

Is the file a Unix-domain socket?

Several of these functions are meaningful only on Unix-like systems, since other platforms do not keep special files such as devices and sockets in the namespace for regular files, as Unix-like systems do.

The `stat` module supplies two more functions that extract relevant parts of a file’s `mode` (`x.st_mode`, for some result `x` of function `os.stat`):

S_IFMT `S_IFMT(mode)`

Returns those bits of `mode` that describe the kind of file (i.e., those bits that are examined by functions `S_ISDIR`, `S_ISREG`, etc.).

S_IMODE `S_IMODE(mode)`

Returns those bits of `mode` that can be set by the function `os.chmod` (i.e., the permission bits and, on Unix-like platforms, a few other special bits such as the `set-user-id` flag).

The filecmp Module

The `filecmp` module supplies the following functions to compare files and directories:

cmp `cmp(f1, f2, shallow=True)`

Compares the files named by path strings `f1` and `f2`. If the files are deemed to be equal, `cmp` returns `True`; otherwise, `False`. If `shallow` is true, files are deemed to be equal if their `stat` tuples are. When `shallow` is `False`, `cmp` reads and compares the contents of files whose `stat` tuples are equal.

cmpfiles `cmpfiles(dir1, dir2, common, shallow=True)`

Loops on the sequence `common`. Each item of `common` is a string that names a file present in both directories `dir1` and `dir2`. `cmpfiles` returns a tuple whose items are three lists of strings: `(equal, diff, errs)`. `equal` is the list of names of files that are equal in both directories, `diff` is the list of names of files that differ between directories, and `errs` is the list of names of files that could not be compared (because they do not exist in both directories, or there is no permission to read them). The argument `shallow` is the same as for `cmp`.

dircmp `=('RCS', 'CVS',` `=('.',`
`class dircmp(dir1, dir2, ignore'tags'),` `hide'..'))`

Creates a new directory-comparison instance object, comparing directories named `dir1` and `dir2`, ignoring names listed in `ignore`, and hiding names listed in `hide`. (In v3, the default value for `ignore` lists more files, and is supplied by attribute `DEFAULT_IGNORE` of module `filecmp`; at the `['RCS', 'CVS', 'tags', '.git', '.hg',`
time of this writing, `'.bzr', '_darcs', '__pycache__']` .)

A `dircmp` instance `d` supplies three methods:

`d.report()`

Outputs to `sys.stdout` a comparison between `dir1` and `dir2`

`d.report_partial_closure()`

Outputs to `sys.stdout` a comparison between `dir1` and `dir2` and their common immediate subdirectories

`d.report_full_closure()`

Outputs to `sys.stdout` a comparison between `dir1` and `dir2` and all their common subdirectories, recursively

In addition, `d` supplies several attributes, covered in the next section.

dircmp instance attributes

A `dircmp` instance `d` supplies several attributes, computed “just in time” (i.e., only if and when needed, thanks to a `__getattr__` special method) so that using a `dircmp` instance suffers no unnecessary overhead:

`d.common`

Files and subdirectories that are in both `dir1` and `dir2`

`d.common_dirs`

Subdirectories that are in both `dir1` and `dir2`

`d.common_files`

Files that are in both `dir1` and `dir2`

`d.common_funny`

Names that are in both `dir1` and `dir2` for which `os.stat` reports an error or returns different kinds for the versions in the two directories

d.diff_files

Files that are in both dir1 and dir2 but with different contents

d.funny_files

Files that are in both dir1 and dir2 but could not be compared

d.left_list

Files and subdirectories that are in dir1

d.left_only

Files and subdirectories that are in dir1 and not in dir2

d.right_list

Files and subdirectories that are in dir2

d.right_only

Files and subdirectories that are in dir2 and not in dir1

d.same_files

Files that are in both dir1 and dir2 with the same contents

d.subdirs

A dictionary whose keys are the strings in common_dirs; the corresponding values are instances of dircmp for each subdirectory

The fnmatch Module

The fnmatch module (an abbreviation for *filename match*) matches filename strings with patterns that resemble the ones used by Unix shells:

*

Matches any sequence of characters

?

Matches any single character

[*chars*]

Matches any one of the characters in *chars*

[!*chars*]

Matches any one character not among those in *chars*

fnmatch does *not* follow other conventions of Unix shells' pattern matching, such as treating a slash / or a leading dot `.` specially. It also does not allow escaping special characters: rather, to literally match a special character, enclose it in brackets. For example, to match a filename that's a single closed bracket, use the pattern `'[]]'`.

The `fnmatch` module supplies the following functions:

filter `filter(`*`names`*`,`*`pattern`*`)`

Returns the list of items of `names` (a sequence of strings) that match `pattern`.

fnmatch `fnmatch(`*`filename`*`,`*`pattern`*`)`

Returns `True` when string `filename` matches `pattern`; otherwise, `False`. The match is case-sensitive when the platform is, for example, any Unix-like system, and otherwise (for example, on Windows) case-insensitive; beware of that, if you're dealing with a filesystem whose case-sensitivity doesn't match your platform (for example, macOS is Unix-like, however, its typical filesystems are case-insensitive).

fnmatchcase `fnmatchcase(`*`filename`*`,`*`pattern`*`)`

Returns `True` when string `filename` matches `pattern`; otherwise, `False`. The match is always case-sensitive on any platform.

translate `translate(`*`pattern`*`)`

Returns the *regular expression* pattern (as covered in "Pattern-String Syntax") equivalent to the `fnmatch` pattern `pattern`. This can be quite useful, for example, to perform matches that are always case-insensitive on any platform—a functionality `fnmatch` doesn't supply:

```
import fnmatch, re

def fnmatchnocase(filename, pattern):
    re_pat = fnmatch.translate(pattern)
    return re.match(re_pat, filename, re.IGNORECASE
)
```

The glob Module

The `glob` module lists (in arbitrary order) the path names of files that match a path *pattern* using the same rules as `fnmatch`; in addition, it does treat a leading dot `.` specially, like Unix shells do.

glob `glob(`*`pathname`*`)`

Returns the list of path names of files that match pattern `pathname`. In v3 only, you can also optionally pass named argument *`recursive`*`=True` to have path component `**` recursively match zero or more levels of subdirectories.

iglob `iglob(`*`pathname`*`)`

Like `glob`, but returns an iterator yielding one relevant path name at a time.

The shutil Module

The `shutil` module (an abbreviation for *shell utilities*) supplies the following functions to copy and move files, and to remove an entire directory tree. In v3 only, on some Unix platforms, most of the functions support optional, keyword-only argument `follow_symlinks`, defaulting to `True`. When true, and always in v2, if a path indicates a symbolic link, the function follows it to reach an actual file or directory; when false, the function operates on the symbolic link itself.

copy

`copy(src, dst)`

Copies the contents of the file `src`, creating or overwriting the file `dst`. If `dst` is a directory, the target is a file with the same base name as `src`, but located in `dst`. `copy` also copies permission bits, but not last-access and modification times. In v3 only, returns the path to the destination file it has copied to (in v2, less usefully, `copy` returns `None`).

copy2

`copy2(src, dst)`

Like `copy`, but also copies times of last access and modification.

copyfile

`copyfile(src, dst)`

Copies just the contents (not permission bits, nor last-access and modification times) of the file `src`, creating or overwriting the file `dst`.

copyfileobj

`copyfileobj(fsrc, fdst, bufsize=16384)`

Copies all bytes from the “file” object `fsrc`, which must be open for reading, to “file” object `fdst`, which must be open for writing. Copies no more than `bufsize` bytes at a time if `bufsize` is greater than `0`. “File” objects are covered in “The io Module”.

copymode

`copymode(src, dst)`

Copies permission bits of the file or directory `src` to file or directory `dst`. Both `src` and `dst` must exist. Does not change `dst`’s contents, nor its status as being a file or a directory.

copystat

`copystat(src, dst)`

Copies permission bits and times of last access and modification of the file or directory `src` to the file or directory `dst`. Both `src` and `dst` must exist. Does not change `dst`’s contents, nor its status as being a file or a directory.

copytree

copytree(src, dst, symlinks=False, *ignore*=None)

Copies the directory tree rooted at src into the destination directory named by dst. dst must not already exist: copytree creates it (as well as creating any missing parent directory). copytree copies each file by using function copy2 (in v3 only, you can optionally pass a different file-copy function as named argument copy_function).

When symlinks is true, copytree creates symbolic links in the new tree when it finds symbolic links in the source tree. When symlinks is false, copytree follows each symbolic link it finds and copies the linked-to file with the link's name. On platforms that do not have the concept of a symbolic link, copytree ignores argument symlinks.

When ignore is not None, it must be a callable accepting two arguments (a directory path and a list of the directory's immediate children) and returning a list of such children to be ignored in the copy process. If present, ignore is usually the result of a call to shutil.ignore_patterns; for example:

```
import shutil
ignore = shutil.ignore_patterns('.*', '*.bak')
shutil.copytree('src', 'dst', ignore=ignore)
```

copies the tree rooted at directory src into a new tree rooted at directory dst, ignoring any file or subdirectory whose name starts with a dot, and any file or subdirectory whose name ends with '.bak'.

ignore_patterns

ignore_patterns(**patterns*)

Returns a callable picking out files and subdirectories matching patterns, like those used in the fnmatch module (see "The fnmatch Module"). The result is suitable for passing as the ignore argument to function copytree.

move

move(src, dst)

Moves the file or directory src to dst. First tries os.rename. Then, if that fails (because src and dst are on separate filesystems, or because dst already exists), copies src to dst (copy2 for a file, copytree for a directory; in v3 only, you can optionally pass a file-copy function other than copy2 as named argument copy_function), then removes src (os.unlink for a file, rmtree for a directory).

rmtree

rmtree(path, ignore_errors=False, onerror=None)

Removes directory tree rooted at path. When ignore_errors is True, rmtree ignores errors. When ignore_errors is False and onerror is None, errors raise exceptions. When onerror is not None, it must be callable with three parameters: func, path, and ex. func is the function raising the exception (os.remove or os.rmdir), path is the path passed to func, and ex the tuple of information sys.exc_info() returns. When onerror raises an exception, rmtree terminates, and the exception propagates.

Beyond offering functions that are directly useful, the source file *shutil.py* in the standard Python library is an excellent example of how to use many os functions.

File Descriptor Operations

The `os` module supplies, among many others, many functions to handle *file descriptors*, integers the operating system uses as opaque handles to refer to open files. Python “file” objects (covered in “The io Module”) are usually better for I/O tasks, but sometimes working at file-descriptor level lets you perform some operation faster, or (sacrificing portability) in ways not directly available with `io.open`. “File” objects and file descriptors are not interchangeable.

To get the file descriptor `n` of a Python “file” object `f`, call `n=f.fileno()`. To wrap a new Python “file” object `f` around an open file descriptor `fd`, call `f=os.fdopen(fd)`, or pass `fd` as the first argument of `io.open`. On Unix-like and Windows platforms, some file descriptors are pre-allocated when a process starts: `0` is the file descriptor for the process’s standard input, `1` for the process’s standard output, and `2` for the process’s standard error.

`os` provides many functions dealing with file descriptors; the most often used ones are listed in Table 10-5.

Table 10-5.

close	`close(fd)` Closes file descriptor `fd`.
closerange	`closerange(`*`fd_low`*`, `*`fd_high`*`)` Closes all file descriptors from `fd_low`, included, to `fd_high`, excluded, ignoring any errors that may occur.
dup	`dup(fd)` Returns a file descriptor that duplicates file descriptor `fd`.
dup2	`dup2(fd, fd2)` Duplicates file descriptor `fd` to file descriptor `fd2`. When file descriptor `fd2` is already open, `dup2` first closes `fd2`.
fdopen	`fdopen(`*`fd`*`, `*`*a`*`, `*`**k`*`)` Like `io.open`, except that `fd` *must* be an `int` that’s an open file descriptor.
fstat	`fstat(fd)` Returns a `stat_result` instance `x`, with info about the file open on file descriptor `fd`. Table 10-4 covers `x`’s contents.
lseek	`lseek(fd, pos, how)` Sets the current position of file descriptor `fd` to the signed integer byte offset `pos` and returns the resulting byte offset from the start of the file. `how` indicates the reference (point `0`). When `how` is `os.SEEK_SET`, the reference is the start of the file; when `os.SEEK_CUR`, the current position; when `os.SEEK_END`, the end of the file. In particular, `lseek(fd, 0, os.SEEK_CUR)` returns the current position’s byte offset from the start of the file without affecting the current position. Normal disk files support seeking; calling `lseek` on a file that does not support seeking (e.g., a file open for output to a terminal) raises an exception.

open `open(file, flags, mode=0o777)`

Returns a file descriptor, opening or creating a file named by string `file`. If `open` creates the file, it uses `mode` as the file's permission bits. `flags` is an `int`, normally the bitwise OR (with operator `|`) of one or more of the following attributes of `os`:

`O_RDONLY` `O_WRONLY`
`O_RDWR`

> Opens `file` for read-only, write-only, or read/write, respectively (mutually exclusive: exactly one of these attributes must be in `flags`)

`O_NDELAY`
`O_NONBLOCK`

> Opens `file` in nonblocking (no-delay) mode if the platform supports this

`O_APPEND`

> Appends any new data to `file`'s previous contents

`O_DSYNC` `O_RSYNC` `O_SYNC`
`O_NOCTTY`

> Sets synchronization mode accordingly if the platform supports this

`O_CREAT`

> Creates `file` if `file` does not already exist

`O_EXCL`

> Raises an exception if `file` already exists

`O_TRUNC`

> Throws away previous contents of `file` (incompatible with `O_RDONLY`)

`O_BINARY`

> Opens `file` in binary rather than text mode on non-Unix platforms (innocuous and without effect on Unix-like platforms)

pipe `pipe()`

Creates a pipe and returns a pair of file descriptors `(r, w)`, respectively open for reading and writing.

read `read(fd, n)`

Reads up to `n` bytes from file descriptor `fd` and returns them as a bytestring. Reads and returns `m< n` bytes when only `m` more bytes are currently available for reading from the file. In particular, returns the empty string when no more bytes are currently available from the file, typically because the file is finished.

write	`write(fd, s)` Writes all bytes from bytestring `s` to file descriptor `fd` and returns the number of bytes written (i.e., `len(s)`).