CHAPTER

ELEVEN

FILE AND DIRECTORY ACCESS

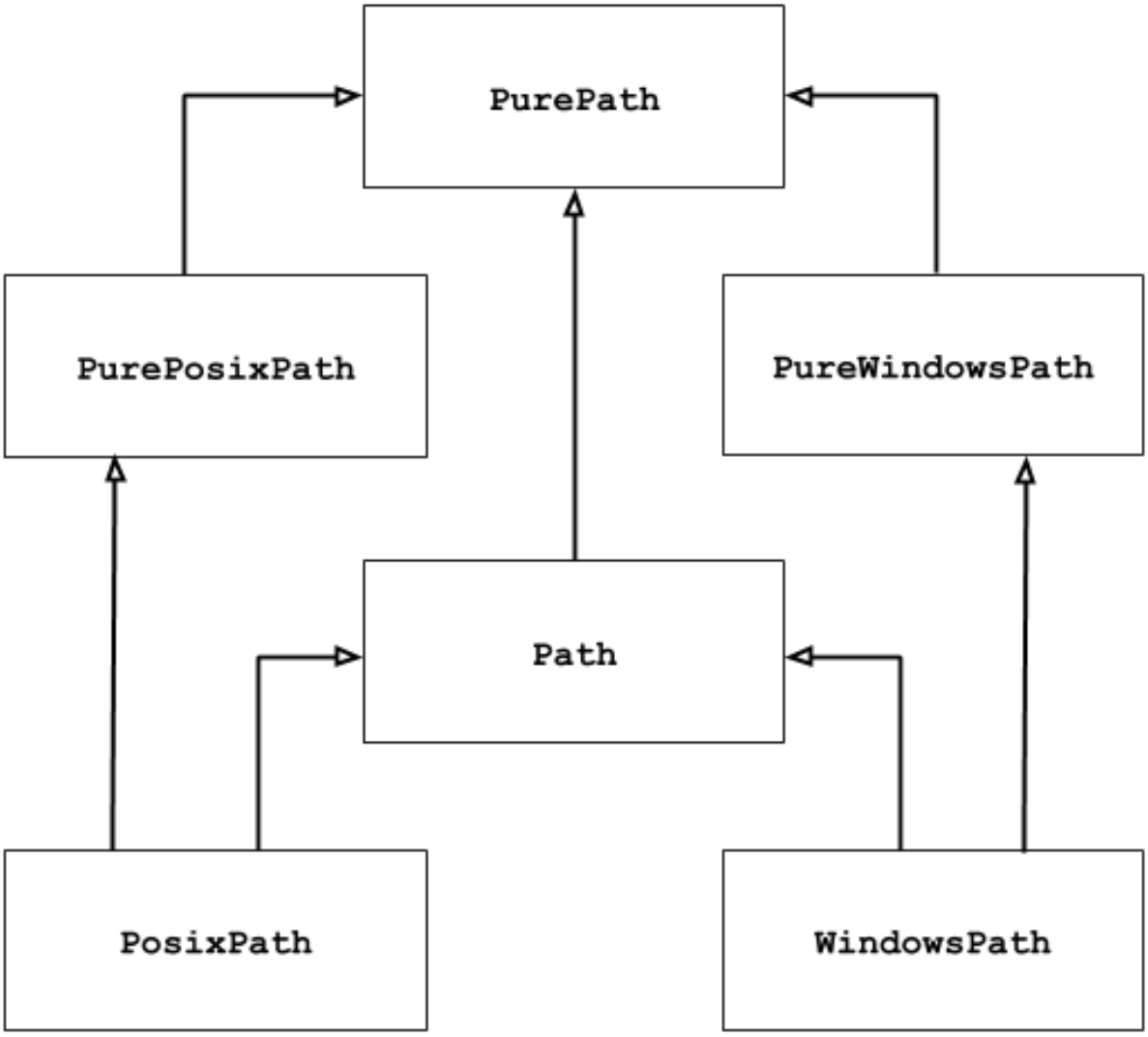
The modules described in this chapter deal with disk files and directories. For example, there are modules for reading the properties of files, manipulating paths in a portable way, and creating temporary files. The full list of modules in this chapter is:

11.1 `pathlib` — Object-oriented filesystem paths

New in version 3.4.

Source code: Lib/pathlib.py

This module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between *pure paths*, which provide purely computational operations without I/O, and *concrete paths*, which inherit from pure paths but also provide I/O operations.



If you've never used this module before or just aren't sure which class is right for your task, *Path* is most likely what you need. It instantiates a *concrete path* for the platform the code is running on.

Pure paths are useful in some special cases; for example:

1. If you want to manipulate Windows paths on a Unix machine (or vice versa). You cannot instantiate a *WindowsPath* when running on Unix, but you can instantiate *PureWindowsPath*.
2. You want to make sure that your code only manipulates paths without actually accessing the OS. In this case, instantiating one of the pure classes may be useful since those simply don't have any OS-accessing operations.

See also:

PEP 428: The pathlib module – object-oriented filesystem paths.

See also:

For low-level path manipulation on strings, you can also use the *os.path* module.

11.1.1 Basic use

Importing the main class:

```
>>> from pathlib import Path
```

Listing subdirectories:

```
>>> p = Path('.')
>>> [x for x in p.iterdir() if x.is_dir()]
[PosixPath('.hg'), PosixPath('docs'), PosixPath('dist'),
 PosixPath('__pycache__'), PosixPath('build')]
```

Listing Python source files in this directory tree:

```
>>> list(p.glob('**/*.py'))
[PosixPath('test_pathlib.py'), PosixPath('setup.py'),
 PosixPath('pathlib.py'), PosixPath('docs/conf.py'),
 PosixPath('build/lib/pathlib.py')]
```

Navigating inside a directory tree:

```
>>> p = Path('/etc')
>>> q = p / 'init.d' / 'reboot'
>>> q
PosixPath('/etc/init.d/reboot')
>>> q.resolve()
PosixPath('/etc/rc.d/init.d/halt')
```

Querying path properties:

```
>>> q.exists()
True
>>> q.is_dir()
False
```

Opening a file:

```
>>> with q.open() as f: f.readline()
...
'#!/bin/bash\n'
```

11.1.2 Pure paths

Pure path objects provide path-handling operations which don't actually access a filesystem. There are three ways to access these classes, which we also call *flavours*:

class pathlib.**PurePath**(**pathsegments*)

A generic class that represents the system's path flavour (instantiating it creates either a *PurePosixPath* or a *PureWindowsPath*):

```
>>> PurePath('setup.py')      # Running on a Unix machine
PurePosixPath('setup.py')
```

Each element of *pathsegments* can be either a string representing a path segment, an object implementing the *os.PathLike* interface which returns a string, or another path object:

```
>>> PurePath('foo', 'some/path', 'bar')
PurePosixPath('foo/some/path/bar')
>>> PurePath(Path('foo'), Path('bar'))
PurePosixPath('foo/bar')
```

When *pathsegments* is empty, the current directory is assumed:

```
>>> PurePath()
PurePosixPath('.')
```

When several absolute paths are given, the last is taken as an anchor (mimicking *os.path.join()*'s behaviour):

```
>>> PurePath('/etc', '/usr', 'lib64')
PurePosixPath('/usr/lib64')
>>> PureWindowsPath('c:/Windows', 'd:bar')
PureWindowsPath('d:bar')
```

However, in a Windows path, changing the local root doesn't discard the previous drive setting:

```
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

Spurious slashes and single dots are collapsed, but double dots (`'..'`) are not, since this would change the meaning of a path in the face of symbolic links:

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/./bar')
PurePosixPath('foo/bar')
>>> PurePath('foo/../bar')
PurePosixPath('foo/../bar')
```

(a naïve approach would make `PurePosixPath('foo/../bar')` equivalent to `PurePosixPath('bar')`, which is wrong if `foo` is a symbolic link to another directory)

Pure path objects implement the *os.PathLike* interface, allowing them to be used anywhere the interface is accepted.

Changed in version 3.6: Added support for the *os.PathLike* interface.

class pathlib.**PurePosixPath**(**pathsegments*)

A subclass of *PurePath*, this path flavour represents non-Windows filesystem paths:

```
>>> PurePosixPath('/etc')
PurePosixPath('/etc')
```

pathsegments is specified similarly to *PurePath*.

class pathlib.**PureWindowsPath**(**pathsegments*)

A subclass of *PurePath*, this path flavour represents Windows filesystem paths:

```
>>> PureWindowsPath('c:/Program Files/')
PureWindowsPath('c:/Program Files')
```

pathsegments is specified similarly to *PurePath*.

Regardless of the system you're running on, you can instantiate all of these classes, since they don't provide any operation that does system calls.

General properties

Paths are immutable and hashable. Paths of a same flavour are comparable and orderable. These properties respect the flavour's case-folding semantics:

```
>>> PurePosixPath('foo') == PurePosixPath('FOO')
False
>>> PureWindowsPath('foo') == PureWindowsPath('FOO')
True
>>> PureWindowsPath('FOO') in { PureWindowsPath('foo') }
True
>>> PureWindowsPath('C:') < PureWindowsPath('d:')
True
```

Paths of a different flavour compare unequal and cannot be ordered:

```
>>> PureWindowsPath('foo') == PurePosixPath('foo')
False
>>> PureWindowsPath('foo') < PurePosixPath('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'PureWindowsPath' and 'PurePosixPath'
```

Operators

The slash operator helps create child paths, similarly to *os.path.join()*:

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
```

A path object can be used anywhere an object implementing *os.PathLike* is accepted:

```
>>> import os
>>> p = PurePath('/etc')
>>> os.fspath(p)
'/etc'
```

The string representation of a path is the raw filesystem path itself (in native form, e.g. with backslashes under Windows), which you can pass to any function taking a file path as a string:

```
>>> p = PurePath('/etc')
>>> str(p)
'/etc'
>>> p = PureWindowsPath('c:/Program Files')
```

(continues on next page)

(continued from previous page)

```
>>> str(p)
'c:\\Program Files'
```

Similarly, calling *bytes* on a path gives the raw filesystem path as a bytes object, as encoded by *os.fsencode()*:

```
>>> bytes(p)
b'/etc'
```

Note: Calling *bytes* is only recommended under Unix. Under Windows, the unicode form is the canonical representation of filesystem paths.

Accessing individual parts

To access the individual "parts" (components) of a path, use the following property:

PurePath.parts

A tuple giving access to the path's various components:

```
>>> p = PurePath('/usr/bin/python3')
>>> p.parts
('/', 'usr', 'bin', 'python3')

>>> p = PureWindowsPath('c:/Program Files/PSF')
>>> p.parts
('c:\\', 'Program Files', 'PSF')
```

(note how the drive and local root are regrouped in a single part)

Methods and properties

Pure paths provide the following methods and properties:

PurePath.drive

A string representing the drive letter or name, if any:

```
>>> PureWindowsPath('c:/Program Files/').drive
'c:'
>>> PureWindowsPath('/Program Files/').drive
''
>>> PurePosixPath('/etc').drive
''
```

UNC shares are also considered drives:

```
>>> PureWindowsPath('//host/share/foo.txt').drive
'\\\\host\\share'
```

PurePath.root

A string representing the (local or global) root, if any:

```
>>> PureWindowsPath('c:/Program Files/').root
'\\'
>>> PureWindowsPath('c:Program Files/').root
''
>>> PurePosixPath('/etc').root
'/'
```

UNC shares always have a root:

```
>>> PureWindowsPath('//host/share').root
'\\'
```

PurePath.anchor

The concatenation of the drive and root:

```
>>> PureWindowsPath('c:/Program Files/').anchor
'c:\\'
>>> PureWindowsPath('c:Program Files/').anchor
'c:'
>>> PurePosixPath('/etc').anchor
'/'
>>> PureWindowsPath('//host/share').anchor
'\\\\host\\share\\'
```

PurePath.parents

An immutable sequence providing access to the logical ancestors of the path:

```
>>> p = PureWindowsPath('c:/foo/bar/setup.py')
>>> p.parents[0]
PureWindowsPath('c:/foo/bar')
>>> p.parents[1]
PureWindowsPath('c:/foo')
>>> p.parents[2]
PureWindowsPath('c:/')
```

PurePath.parent

The logical parent of the path:

```
>>> p = PurePosixPath('/a/b/c/d')
>>> p.parent
PurePosixPath('/a/b/c')
```

You cannot go past an anchor, or empty path:

```
>>> p = PurePosixPath('/')
>>> p.parent
PurePosixPath('/')
>>> p = PurePosixPath('.')
>>> p.parent
PurePosixPath('.')
```

Note: This is a purely lexical operation, hence the following behaviour:

```
>>> p = PurePosixPath('foo/..')
>>> p.parent
PurePosixPath('foo')
```

If you want to walk an arbitrary filesystem path upwards, it is recommended to first call *Path.resolve()* so as to resolve symlinks and eliminate "." components.

PurePath.name

A string representing the final path component, excluding the drive and root, if any:

```
>>> PurePosixPath('my/library/setup.py').name
'setup.py'
```

UNC drive names are not considered:

```
>>> PureWindowsPath('//some/share/setup.py').name
'setup.py'
>>> PureWindowsPath('//some/share').name
''
```

PurePath.suffix

The file extension of the final component, if any:

```
>>> PurePosixPath('my/library/setup.py').suffix
'.py'
>>> PurePosixPath('my/library.tar.gz').suffix
'.gz'
>>> PurePosixPath('my/library').suffix
''
```

PurePath.suffixes

A list of the path's file extensions:

```
>>> PurePosixPath('my/library.tar.gar').suffixes
['.tar', '.gar']
>>> PurePosixPath('my/library.tar.gz').suffixes
['.tar', '.gz']
>>> PurePosixPath('my/library').suffixes
[]
```

PurePath.stem

The final path component, without its suffix:

```
>>> PurePosixPath('my/library.tar.gz').stem
'library.tar'
>>> PurePosixPath('my/library.tar').stem
'library'
>>> PurePosixPath('my/library').stem
'library'
```

PurePath.as_posix()

Return a string representation of the path with forward slashes (/):

```
>>> p = PureWindowsPath('c:\\windows')
>>> str(p)
'c:\\windows'
>>> p.as_posix()
'c:/windows'
```

PurePath.as_uri()
Represent the path as a file URI. *ValueError* is raised if the path isn't absolute.

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.as_uri()
'file:///etc/passwd'
>>> p = PureWindowsPath('c:/Windows')
>>> p.as_uri()
'file:///c:/Windows'
```

PurePath.is_absolute()
Return whether the path is absolute or not. A path is considered absolute if it has both a root and (if the flavour allows) a drive:

```
>>> PurePosixPath('/a/b').is_absolute()
True
>>> PurePosixPath('a/b').is_absolute()
False

>>> PureWindowsPath('c:/a/b').is_absolute()
True
>>> PureWindowsPath('/a/b').is_absolute()
False
>>> PureWindowsPath('c:').is_absolute()
False
>>> PureWindowsPath('//some/share').is_absolute()
True
```

PurePath.is_reserved()
With *PureWindowsPath*, return True if the path is considered reserved under Windows, False otherwise. With *PurePosixPath*, False is always returned.

```
>>> PureWindowsPath('nul').is_reserved()
True
>>> PurePosixPath('nul').is_reserved()
False
```

File system calls on reserved paths can fail mysteriously or have unintended effects.

PurePath.joinpath(*other)
Calling this method is equivalent to combining the path with each of the *other* arguments in turn:

```
>>> PurePosixPath('/etc').joinpath('passwd')
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath(PurePosixPath('passwd'))
PurePosixPath('/etc/passwd')
>>> PurePosixPath('/etc').joinpath('init.d', 'apache2')
PurePosixPath('/etc/init.d/apache2')
```

(continues on next page)

(continued from previous page)

```
>>> PureWindowsPath('c:').joinpath('/Program Files')
PureWindowsPath('c:/Program Files')
```

PurePath.**match**(*pattern*)

Match this path against the provided glob-style pattern. Return `True` if matching is successful, `False` otherwise.

If *pattern* is relative, the path can be either relative or absolute, and matching is done from the right:

```
>>> PurePath('a/b.py').match('*.py')
True
>>> PurePath('/a/b/c.py').match('b/*.py')
True
>>> PurePath('/a/b/c.py').match('a/*.py')
False
```

If *pattern* is absolute, the path must be absolute, and the whole path must match:

```
>>> PurePath('/a.py').match('/*.py')
True
>>> PurePath('a/b.py').match('/*.py')
False
```

As with other methods, case-sensitivity is observed:

```
>>> PureWindowsPath('b.py').match('*.PY')
True
```

PurePath.**relative_to**(**other*)

Compute a version of this path relative to the path represented by *other*. If it's impossible, ValueError is raised:

```
>>> p = PurePosixPath('/etc/passwd')
>>> p.relative_to('/')
PurePosixPath('etc/passwd')
>>> p.relative_to('/etc')
PurePosixPath('passwd')
>>> p.relative_to('/usr')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 694, in relative_to
    .format(str(self), str(formatted)))
ValueError: '/etc/passwd' does not start with '/usr'
```

PurePath.**with_name**(*name*)

Return a new path with the *name* changed. If the original path doesn't have a name, ValueError is raised:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_name('setup.py')
PureWindowsPath('c:/Downloads/setup.py')
>>> p = PureWindowsPath('c:/')
>>> p.with_name('setup.py')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
  File "/home/antoine/cpython/default/Lib/pathlib.py", line 751, in with_name
    raise ValueError("%r has an empty name" % (self,))
ValueError: PureWindowsPath('c:/') has an empty name
```

PurePath.**with_suffix**(*suffix*)

Return a new path with the *suffix* changed. If the original path doesn't have a suffix, the new *suffix* is appended instead. If the *suffix* is an empty string, the original suffix is removed:

```
>>> p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')
>>> p.with_suffix('.bz2')
PureWindowsPath('c:/Downloads/pathlib.tar.bz2')
>>> p = PureWindowsPath('README')
>>> p.with_suffix('.txt')
PureWindowsPath('README.txt')
>>> p = PureWindowsPath('README.txt')
>>> p.with_suffix('')
PureWindowsPath('README')
```

11.1.3 Concrete paths

Concrete paths are subclasses of the pure path classes. In addition to operations provided by the latter, they also provide methods to do system calls on path objects. There are three ways to instantiate concrete paths:

class pathlib.**Path**(**pathsegments*)

A subclass of *PurePath*, this class represents concrete paths of the system's path flavour (instantiating it creates either a *PosixPath* or a *WindowsPath*):

```
>>> Path('setup.py')
PosixPath('setup.py')
```

pathsegments is specified similarly to *PurePath*.

class pathlib.**PosixPath**(**pathsegments*)

A subclass of *Path* and *PurePosixPath*, this class represents concrete non-Windows filesystem paths:

```
>>> PosixPath('/etc')
PosixPath('/etc')
```

pathsegments is specified similarly to *PurePath*.

class pathlib.**WindowsPath**(**pathsegments*)

A subclass of *Path* and *PureWindowsPath*, this class represents concrete Windows filesystem paths:

```
>>> WindowsPath('c:/Program Files/')
WindowsPath('c:/Program Files')
```

pathsegments is specified similarly to *PurePath*.

You can only instantiate the class flavour that corresponds to your system (allowing system calls on non-compatible path flavours could lead to bugs or failures in your application):

```
>>> import os
>>> os.name
'posix'
>>> Path('setup.py')
PosixPath('setup.py')
>>> PosixPath('setup.py')
PosixPath('setup.py')
>>> WindowsPath('setup.py')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 798, in __new__
    % (cls.__name__,))
NotImplementedError: cannot instantiate 'WindowsPath' on your system
```

Methods

Concrete paths provide the following methods in addition to pure paths methods. Many of these methods can raise an *OSError* if a system call fails (for example because the path doesn't exist):

classmethod Path.**cwd**()
Return a new path object representing the current directory (as returned by *os.getcwd()*):

```
>>> Path.cwd()
PosixPath('/home/antoine/pathlib')
```

classmethod Path.**home**()
Return a new path object representing the user's home directory (as returned by *os.path.expanduser()* with ~ construct):

```
>>> Path.home()
PosixPath('/home/antoine')
```

New in version 3.5.

Path.**stat**()
Return information about this path (similarly to *os.stat()*). The result is looked up at each call to this method.

```
>>> p = Path('setup.py')
>>> p.stat().st_size
956
>>> p.stat().st_mtime
1327883547.852554
```

Path.**chmod**(*mode*)
Change the file mode and permissions, like *os.chmod()*:

```
>>> p = Path('setup.py')
>>> p.stat().st_mode
33277
>>> p.chmod(0o444)
>>> p.stat().st_mode
33060
```

Path.exists()
Whether the path points to an existing file or directory:

```
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

Note: If the path points to a symlink, *exists()* returns whether the symlink *points to* an existing file or directory.

Path.expanduser()
Return a new path with expanded ~ and ~user constructs, as returned by *os.path.expanduser()*:

```
>>> p = PosixPath('~/films/Monty Python')
>>> p.expanduser()
PosixPath('/home/eric/films/Monty Python')
```

New in version 3.5.

Path.glob(*pattern*)
Glob the given *pattern* in the directory represented by this path, yielding all matching files (of any kind):

```
>>> sorted(Path('.').glob('*.py'))
[PosixPath('pathlib.py'), PosixPath('setup.py'), PosixPath('test_pathlib.py')]
>>> sorted(Path('.').glob('*/*.py'))
[PosixPath('docs/conf.py')]
```

The "**" pattern means "this directory and all subdirectories, recursively". In other words, it enables recursive globbing:

```
>>> sorted(Path('.').glob('**/*.py'))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

Note: Using the "**" pattern in large directory trees may consume an inordinate amount of time.

Path.group()
Return the name of the group owning the file. *KeyError* is raised if the file's gid isn't found in the system database.

Path.is_dir()
Return True if the path points to a directory (or a symbolic link pointing to a directory), False if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.is_file()`
Return `True` if the path points to a regular file (or a symbolic link pointing to a regular file), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.is_mount()`
Return `True` if the path is a *mount point*: a point in a file system where a different file system has been mounted. On POSIX, the function checks whether *path*'s parent, `path/..`, is on a different device than *path*, or whether `path/..` and *path* point to the same i-node on the same device — this should detect mount points for all Unix and POSIX variants. Not implemented on Windows.

New in version 3.7.

`Path.is_symlink()`
Return `True` if the path points to a symbolic link, `False` otherwise.

`False` is also returned if the path doesn't exist; other errors (such as permission errors) are propagated.

`Path.is_socket()`
Return `True` if the path points to a Unix socket (or a symbolic link pointing to a Unix socket), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.is_fifo()`
Return `True` if the path points to a FIFO (or a symbolic link pointing to a FIFO), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.is_block_device()`
Return `True` if the path points to a block device (or a symbolic link pointing to a block device), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.is_char_device()`
Return `True` if the path points to a character device (or a symbolic link pointing to a character device), `False` if it points to another kind of file.

`False` is also returned if the path doesn't exist or is a broken symlink; other errors (such as permission errors) are propagated.

`Path.iterdir()`
When the path points to a directory, yield path objects of the directory contents:

```
>>> p = Path('docs')
>>> for child in p.iterdir(): child
...
PosixPath('docs/conf.py')
PosixPath('docs/_templates')
PosixPath('docs/make.bat')
PosixPath('docs/index.rst')
PosixPath('docs/_build')
```

(continues on next page)

(continued from previous page)

```
PosixPath('docs/_static')
PosixPath('docs/Makefile')
```

Path.lchmod(*mode*)
Like *Path.chmod()* but, if the path points to a symbolic link, the symbolic link's mode is changed rather than its target's.

Path.lstat()
Like *Path.stat()* but, if the path points to a symbolic link, return the symbolic link's information rather than its target's.

Path.mkdir(*mode=0o777*, *parents=False*, *exist_ok=False*)
Create a new directory at this given path. If *mode* is given, it is combined with the process' `umask` value to determine the file mode and access flags. If the path already exists, *FileExistsError* is raised.

If *parents* is true, any missing parents of this path are created as needed; they are created with the default permissions without taking *mode* into account (mimicking the POSIX `mkdir -p` command).

If *parents* is false (the default), a missing parent raises *FileNotFoundError*.

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

If *exist_ok* is true, *FileExistsError* exceptions will be ignored (same behavior as the POSIX `mkdir -p` command), but only if the last path component is not an existing non-directory file.

Changed in version 3.5: The *exist_ok* parameter was added.

Path.open(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)
Open the file pointed to by the path, like the built-in *open()* function does:

```
>>> p = Path('setup.py')
>>> with p.open() as f:
...     f.readline()
...
'#!/usr/bin/env python3\n'
```

Path.owner()
Return the name of the user owning the file. *KeyError* is raised if the file's uid isn't found in the system database.

Path.read_bytes()
Return the binary contents of the pointed-to file as a bytes object:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

New in version 3.5.

Path.read_text(*encoding=None*, *errors=None*)
Return the decoded contents of the pointed-to file as a string:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
18
```

(continues on next page)

(continued from previous page)

```
>>> p.read_text()
'Text file contents'
```

The file is opened and then closed. The optional parameters have the same meaning as in *open()*.

New in version 3.5.

Path.rename(*target*)
Rename this file or directory to the given *target*. On Unix, if *target* exists and is a file, it will be replaced silently if the user has permission. *target* can be either a string or another path object:

```
>>> p = Path('foo')
>>> p.open('w').write('some text')
9
>>> target = Path('bar')
>>> p.rename(target)
>>> target.open().read()
'some text'
```

Path.replace(*target*)
Rename this file or directory to the given *target*. If *target* points to an existing file or directory, it will be unconditionally replaced.

Path.resolve(*strict=False*)
Make the path absolute, resolving any symlinks. A new path object is returned:

```
>>> p = Path()
>>> p
PosixPath('.')
>>> p.resolve()
PosixPath('/home/antoine/pathlib')
```

"`..`" components are also eliminated (this is the only method to do so):

```
>>> p = Path('docs/../setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
```

If the path doesn't exist and *strict* is `True`, *FileNotFoundError* is raised. If *strict* is `False`, the path is resolved as far as possible and any remainder is appended without checking whether it exists. If an infinite loop is encountered along the resolution path, *RuntimeError* is raised.

New in version 3.6: The *strict* argument.

Path.rglob(*pattern*)
This is like calling *Path.glob()* with "`**`" added in front of the given *pattern*:

```
>>> sorted(Path().rglob("*.py"))
[PosixPath('build/lib/pathlib.py'),
 PosixPath('docs/conf.py'),
 PosixPath('pathlib.py'),
 PosixPath('setup.py'),
 PosixPath('test_pathlib.py')]
```

Path.rmdir()
Remove this directory. The directory must be empty.

Path.**samefile**(*other_path*)
Return whether this path points to the same file as *other_path*, which can be either a Path object, or a string. The semantics are similar to *os.path.samefile()* and *os.path.samestat()*.

An *OSError* can be raised if either file cannot be accessed for some reason.

```
>>> p = Path('spam')
>>> q = Path('eggs')
>>> p.samefile(q)
False
>>> p.samefile('spam')
True
```

New in version 3.5.

Path.**symlink_to**(*target*, *target_is_directory=False*)
Make this path a symbolic link to *target*. Under Windows, *target_is_directory* must be true (default `False`) if the link's target is a directory. Under POSIX, *target_is_directory*'s value is ignored.

```
>>> p = Path('mylink')
>>> p.symlink_to('setup.py')
>>> p.resolve()
PosixPath('/home/antoine/pathlib/setup.py')
>>> p.stat().st_size
956
>>> p.lstat().st_size
8
```

Note: The order of arguments (link, target) is the reverse of *os.symlink()*'s.

Path.**touch**(*mode=0o666*, *exist_ok=True*)
Create a file at this given path. If *mode* is given, it is combined with the process' `umask` value to determine the file mode and access flags. If the file already exists, the function succeeds if *exist_ok* is true (and its modification time is updated to the current time), otherwise *FileExistsError* is raised.

Path.**unlink**()
Remove this file or symbolic link. If the path points to a directory, use *Path.rmdir()* instead.

Path.**write_bytes**(*data*)
Open the file pointed to in bytes mode, write *data* to it, and close the file:

```
>>> p = Path('my_binary_file')
>>> p.write_bytes(b'Binary file contents')
20
>>> p.read_bytes()
b'Binary file contents'
```

An existing file of the same name is overwritten.

New in version 3.5.

Path.**write_text**(*data*, *encoding=None*, *errors=None*)
Open the file pointed to in text mode, write *data* to it, and close the file:

```
>>> p = Path('my_text_file')
>>> p.write_text('Text file contents')
```

(continues on next page)

(continued from previous page)

```
18
>>> p.read_text()
'Text file contents'
```

New in version 3.5.

11.1.4 Correspondence to tools in the os module

Below is a table mapping various *os* functions to their corresponding *PurePath*/*Path* equivalent.

Note: Although *os.path.relpath()* and *PurePath.relative_to()* have some overlapping use-cases, their semantics differ enough to warrant not considering them equivalent.

os and os.path	pathlib
os.path.abspath()	*Path.resolve()*
os.chmod()	*Path.chmod()*
os.mkdir()	*Path.mkdir()*
os.rename()	*Path.rename()*
os.replace()	*Path.replace()*
os.rmdir()	*Path.rmdir()*
os.remove(), *os.unlink()*	*Path.unlink()*
os.getcwd()	*Path.cwd()*
os.path.exists()	*Path.exists()*
os.path.expanduser()	*Path.expanduser()* and *Path.home()*
os.path.isdir()	*Path.is_dir()*
os.path.isfile()	*Path.is_file()*
os.path.islink()	*Path.is_symlink()*
os.stat()	*Path.stat()*, *Path.owner()*, *Path.group()*
os.path.isabs()	*PurePath.is_absolute()*
os.path.join()	*PurePath.joinpath()*
os.path.basename()	*PurePath.name*
os.path.dirname()	*PurePath.parent*
os.path.samefile()	*Path.samefile()*
os.path.splitext()	*PurePath.suffix*

11.2 os.path — Common pathname manipulations

Source code: Lib/posixpath.py (for POSIX), Lib/ntpath.py (for Windows NT), and Lib/macpath.py (for Macintosh)

This module implements some useful functions on pathnames. To read or write files see *open()*, and for accessing the filesystem see the *os* module. The path parameters can be passed as either strings, or bytes. Applications are encouraged to represent file names as (Unicode) character strings. Unfortunately, some file names may not be representable as strings on Unix, so applications that need to support arbitrary file names on Unix should use bytes objects to represent path names. Vice versa, using bytes objects cannot represent all file names on Windows (in the standard `mbcs` encoding), hence Windows applications should use string objects to access all files.

Unlike a unix shell, Python does not do any *automatic* path expansions. Functions such as *expanduser()* and *expandvars()* can be invoked explicitly when an application desires shell-like path expansion. (See also the *glob* module.)

See also:

The *pathlib* module offers high-level path objects.

Note: All of these functions accept either only bytes or only string objects as their parameters. The result is an object of the same type, if a path or file name is returned.

Note: Since different operating systems have different path name conventions, there are several versions of this module in the standard library. The *os.path* module is always the path module suitable for the operating system Python is running on, and therefore usable for local paths. However, you can also import and use the individual modules if you want to manipulate a path that is *always* in one of the different formats. They all have the same interface:

- `posixpath` for UNIX-style paths
- `ntpath` for Windows paths
- *macpath* for old-style MacOS paths

`os.path.abspath(`*path*`)`
: Return a normalized absolutized version of the pathname *path*. On most platforms, this is equivalent to calling the function *normpath()* as follows: `normpath(join(os.getcwd(), path))`.

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

`os.path.basename(`*path*`)`
: Return the base name of pathname *path*. This is the second element of the pair returned by passing *path* to the function *split()*. Note that the result of this function is different from the Unix **basename** program; where **basename** for `'/foo/bar/'` returns `'bar'`, the *basename()* function returns an empty string (`''`).

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

`os.path.commonpath(`*paths*`)`
: Return the longest common sub-path of each pathname in the sequence *paths*. Raise ValueError if *paths* contains both absolute and relative pathnames, or if *paths* is empty. Unlike *commonprefix()*, this returns a valid path.

  Availability: Unix, Windows.

  New in version 3.5.

  Changed in version 3.6: Accepts a sequence of *path-like objects*.

`os.path.commonprefix(`*list*`)`
: Return the longest path prefix (taken character-by-character) that is a prefix of all paths in *list*. If *list* is empty, return the empty string (`''`).

  Note: This function may return invalid paths because it works a character at a time. To obtain a valid path, see *commonpath()*.

```
>>> os.path.commonprefix(['/usr/lib', '/usr/local/lib'])
'/usr/l'
```

(continues on next page)

(continued from previous page)

```
>>> os.path.commonpath(['/usr/lib', '/usr/local/lib'])
'/usr'
```

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

`os.path.dirname`(*path*)
Return the directory name of pathname *path*. This is the first element of the pair returned by passing *path* to the function *split()*.

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

`os.path.exists`(*path*)
Return `True` if *path* refers to an existing path or an open file descriptor. Returns `False` for broken symbolic links. On some platforms, this function may return `False` if permission is not granted to execute *os.stat()* on the requested file, even if the *path* physically exists.

Changed in version 3.3: *path* can now be an integer: `True` is returned if it is an open file descriptor, `False` otherwise.

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

`os.path.lexists`(*path*)
Return `True` if *path* refers to an existing path. Returns `True` for broken symbolic links. Equivalent to *exists()* on platforms lacking *os.lstat()*.

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

`os.path.expanduser`(*path*)
On Unix and Windows, return the argument with an initial component of `~` or `~user` replaced by that *user*'s home directory.

On Unix, an initial `~` is replaced by the environment variable `HOME` if it is set; otherwise the current user's home directory is looked up in the password directory through the built-in module *pwd*. An initial `~user` is looked up directly in the password directory.

On Windows, `HOME` and `USERPROFILE` will be used if set, otherwise a combination of `HOMEPATH` and `HOMEDRIVE` will be used. An initial `~user` is handled by stripping the last directory component from the created user path derived above.

If the expansion fails or if the path does not begin with a tilde, the path is returned unchanged.

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

`os.path.expandvars`(*path*)
Return the argument with environment variables expanded. Substrings of the form `$name` or `${name}` are replaced by the value of environment variable *name*. Malformed variable names and references to non-existing variables are left unchanged.

On Windows, `%name%` expansions are supported in addition to `$name` and `${name}`.

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

`os.path.getatime`(*path*)
Return the time of last access of *path*. The return value is a floating point number giving the number of seconds since the epoch (see the *time* module). Raise *OSError* if the file does not exist or is inaccessible.

`os.path.getmtime`(*path*)
Return the time of last modification of *path*. The return value is a floating point number giving the number of seconds since the epoch (see the *time* module). Raise *OSError* if the file does not exist or is inaccessible.

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

os.path.getctime(*path*)
Return the system's ctime which, on some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time for *path*. The return value is a number giving the number of seconds since the epoch (see the *time* module). Raise *OSError* if the file does not exist or is inaccessible.

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

os.path.getsize(*path*)
Return the size, in bytes, of *path*. Raise *OSError* if the file does not exist or is inaccessible.

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

os.path.isabs(*path*)
Return `True` if *path* is an absolute pathname. On Unix, that means it begins with a slash, on Windows that it begins with a (back)slash after chopping off a potential drive letter.

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

os.path.isfile(*path*)
Return `True` if *path* is an *existing* regular file. This follows symbolic links, so both *islink()* and *isfile()* can be true for the same path.

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

os.path.isdir(*path*)
Return `True` if *path* is an *existing* directory. This follows symbolic links, so both *islink()* and *isdir()* can be true for the same path.

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

os.path.islink(*path*)
Return `True` if *path* refers to an *existing* directory entry that is a symbolic link. Always `False` if symbolic links are not supported by the Python runtime.

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

os.path.ismount(*path*)
Return `True` if pathname *path* is a *mount point*: a point in a file system where a different file system has been mounted. On POSIX, the function checks whether *path*'s parent, `path/..`, is on a different device than *path*, or whether `path/..` and *path* point to the same i-node on the same device — this should detect mount points for all Unix and POSIX variants. On Windows, a drive letter root and a share UNC are always mount points, and for any other path `GetVolumePathName` is called to see if it is different from the input path.

New in version 3.4: Support for detecting non-root mount points on Windows.

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

os.path.join(*path*, **paths*)
Join one or more path components intelligently. The return value is the concatenation of *path* and any members of **paths* with exactly one directory separator (`os.sep`) following each non-empty part except the last, meaning that the result will only end in a separator if the last part is empty. If a component is an absolute path, all previous components are thrown away and joining continues from the absolute path component.

On Windows, the drive letter is not reset when an absolute path component (e.g., `r'\foo'`) is encountered. If a component contains a drive letter, all previous components are thrown away and the drive letter is reset. Note that since there is a current directory for each drive, `os.path.join("c:", "foo")` represents a path relative to the current directory on drive `C:` (`c:foo`), not `c:\foo`.

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

os.path.**normcase**(*path*)
Normalize the case of a pathname. On Unix and Mac OS X, this returns the path unchanged; on case-insensitive filesystems, it converts the path to lowercase. On Windows, it also converts forward slashes to backward slashes. Raise a *TypeError* if the type of *path* is not `str` or `bytes` (directly or indirectly through the *os.PathLike* interface).

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

os.path.**normpath**(*path*)
Normalize a pathname by collapsing redundant separators and up-level references so that `A//B`, `A/B/`, `A/./B` and `A/foo/../B` all become `A/B`. This string manipulation may change the meaning of a path that contains symbolic links. On Windows, it converts forward slashes to backward slashes. To normalize case, use *normcase()*.

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

os.path.**realpath**(*path*)
Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path (if they are supported by the operating system).

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

os.path.**relpath**(*path*, *start=os.curdir*)
Return a relative filepath to *path* either from the current directory or from an optional *start* directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of *path* or *start*.

start defaults to *os.curdir*.

Availability: Unix, Windows.

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

os.path.**samefile**(*path1*, *path2*)
Return `True` if both pathname arguments refer to the same file or directory. This is determined by the device number and i-node number and raises an exception if an *os.stat()* call on either pathname fails.

Availability: Unix, Windows.

Changed in version 3.2: Added Windows support.

Changed in version 3.4: Windows now uses the same implementation as all other platforms.

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

os.path.**sameopenfile**(*fp1*, *fp2*)
Return `True` if the file descriptors *fp1* and *fp2* refer to the same file.

Availability: Unix, Windows.

Changed in version 3.2: Added Windows support.

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

os.path.**samestat**(*stat1*, *stat2*)
Return `True` if the stat tuples *stat1* and *stat2* refer to the same file. These structures may have been returned by *os.fstat()*, *os.lstat()*, or *os.stat()*. This function implements the underlying comparison used by *samefile()* and *sameopenfile()*.

Availability: Unix, Windows.

Changed in version 3.4: Added Windows support.

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

os.path.split(*path*)
: Split the pathname *path* into a pair, `(head, tail)` where *tail* is the last pathname component and *head* is everything leading up to that. The *tail* part will never contain a slash; if *path* ends in a slash, *tail* will be empty. If there is no slash in *path*, *head* will be empty. If *path* is empty, both *head* and *tail* are empty. Trailing slashes are stripped from *head* unless it is the root (one or more slashes only). In all cases, `join(head, tail)` returns a path to the same location as *path* (but the strings may differ). Also see the functions *dirname()* and *basename()*.

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

os.path.splitdrive(*path*)
: Split the pathname *path* into a pair `(drive, tail)` where *drive* is either a mount point or the empty string. On systems which do not use drive specifications, *drive* will always be the empty string. In all cases, `drive + tail` will be the same as *path*.

  On Windows, splits a pathname into drive/UNC sharepoint and relative path.

  If the path contains a drive letter, drive will contain everything up to and including the colon. e.g. `splitdrive("c:/dir")` returns `("c:", "/dir")`

  If the path contains a UNC path, drive will contain the host name and share, up to but not including the fourth separator. e.g. `splitdrive("//host/computer/dir")` returns `("//host/computer", "/dir")`

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

os.path.splitext(*path*)
: Split the pathname *path* into a pair `(root, ext)` such that `root + ext == path`, and *ext* is empty or begins with a period and contains at most one period. Leading periods on the basename are ignored; `splitext('.cshrc')` returns `('.cshrc', '')`.

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

os.path.supports_unicode_filenames
: `True` if arbitrary Unicode strings can be used as file names (within limitations imposed by the file system).

11.3 `fileinput` — Iterate over lines from multiple input streams

Source code: Lib/fileinput.py

This module implements a helper class and functions to quickly write a loop over standard input or a list of files. If you just want to read or write one file see *open()*.

The typical use is:

```
import fileinput
for line in fileinput.input():
    process(line)
```

This iterates over the lines of all files listed in `sys.argv[1:]`, defaulting to `sys.stdin` if the list is empty. If a filename is `'-'`, it is also replaced by `sys.stdin`. To specify an alternative list of filenames, pass it as the first argument to *input()*. A single file name is also allowed.

All files are opened in text mode by default, but you can override this by specifying the *mode* parameter in the call to *input()* or *FileInput*. If an I/O error occurs during opening or reading a file, *OSError* is raised.

Changed in version 3.3: *IOError* used to be raised; it is now an alias of *OSError*.

If `sys.stdin` is used more than once, the second and further use will return no lines, except perhaps for interactive use, or if it has been explicitly reset (e.g. using `sys.stdin.seek(0)`).

Empty files are opened and immediately closed; the only time their presence in the list of filenames is noticeable at all is when the last file opened is empty.

Lines are returned with any newlines intact, which means that the last line in a file may not have one.

You can control how files are opened by providing an opening hook via the *openhook* parameter to *fileinput.input()* or *FileInput()*. The hook must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. Two useful hooks are already provided by this module.

The following function is the primary interface of this module:

`fileinput.input`(*files=None*, *inplace=False*, *backup=''*, *bufsize=0*, *mode='r'*, *openhook=None*)
Create an instance of the *FileInput* class. The instance will be used as global state for the functions of this module, and is also returned to use during iteration. The parameters to this function will be passed along to the constructor of the *FileInput* class.

The *FileInput* instance can be used as a context manager in the `with` statement. In this example, *input* is closed after the `with` statement is exited, even if an exception occurs:

```
with fileinput.input(files=('spam.txt', 'eggs.txt')) as f:
    for line in f:
        process(line)
```

Changed in version 3.2: Can be used as a context manager.

Deprecated since version 3.6, will be removed in version 3.8: The *bufsize* parameter.

The following functions use the global state created by *fileinput.input()*; if there is no active state, *RuntimeError* is raised.

`fileinput.filename`()
Return the name of the file currently being read. Before the first line has been read, returns `None`.

`fileinput.fileno`()
Return the integer "file descriptor" for the current file. When no file is opened (before the first line and between files), returns `-1`.

`fileinput.lineno`()
Return the cumulative line number of the line that has just been read. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line.

`fileinput.filelineno`()
Return the line number in the current file. Before the first line has been read, returns `0`. After the last line of the last file has been read, returns the line number of that line within the file.

`fileinput.isfirstline`()
Returns true if the line just read is the first line of its file, otherwise returns false.

`fileinput.isstdin`()
Returns true if the last line was read from `sys.stdin`, otherwise returns false.

`fileinput.nextfile`()
Close the current file so that the next iteration will read the first line from the next file (if any); lines not read from the file will not count towards the cumulative line count. The filename is not changed until after the first line of the next file has been read. Before the first line has been read, this function has no effect; it cannot be used to skip the first file. After the last line of the last file has been read, this function has no effect.

`fileinput.close`()
Close the sequence.

The class which implements the sequence behavior provided by the module is available for subclassing as well:

class fileinput.FileInput(*files=None*, *inplace=False*, *backup=''*, *bufsize=0*, *mode='r'*, *openhook=None*)

Class *FileInput* is the implementation; its methods *filename()*, *fileno()*, *lineno()*, *filelineno()*, *isfirstline()*, *isstdin()*, *nextfile()* and *close()* correspond to the functions of the same name in the module. In addition it has a *readline()* method which returns the next input line, and a `__getitem__()` method which implements the sequence behavior. The sequence must be accessed in strictly sequential order; random access and *readline()* cannot be mixed.

With *mode* you can specify which file mode will be passed to *open()*. It must be one of `'r'`, `'rU'`, `'U'` and `'rb'`.

The *openhook*, when given, must be a function that takes two arguments, *filename* and *mode*, and returns an accordingly opened file-like object. You cannot use *inplace* and *openhook* together.

A *FileInput* instance can be used as a context manager in the `with` statement. In this example, *input* is closed after the `with` statement is exited, even if an exception occurs:

```
with FileInput(files=('spam.txt', 'eggs.txt')) as input:
    process(input)
```

Changed in version 3.2: Can be used as a context manager.

Deprecated since version 3.4: The `'rU'` and `'U'` modes.

Deprecated since version 3.6, will be removed in version 3.8: The *bufsize* parameter.

Optional in-place filtering: if the keyword argument `inplace=True` is passed to *fileinput.input()* or to the *FileInput* constructor, the file is moved to a backup file and standard output is directed to the input file (if a file of the same name as the backup file already exists, it will be replaced silently). This makes it possible to write a filter that rewrites its input file in place. If the *backup* parameter is given (typically as `backup='.<some extension>'`), it specifies the extension for the backup file, and the backup file remains around; by default, the extension is `'.bak'` and it is deleted when the output file is closed. In-place filtering is disabled when standard input is read.

The two following opening hooks are provided by this module:

fileinput.hook_compressed(*filename*, *mode*)

Transparently opens files compressed with gzip and bzip2 (recognized by the extensions `'.gz'` and `'.bz2'`) using the *gzip* and *bz2* modules. If the filename extension is not `'.gz'` or `'.bz2'`, the file is opened normally (ie, using *open()* without any decompression).

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_compressed)`

fileinput.hook_encoded(*encoding*, *errors=None*)

Returns a hook which opens each file with *open()*, using the given *encoding* and *errors* to read the file.

Usage example: `fi = fileinput.FileInput(openhook=fileinput.hook_encoded("utf-8", "surrogateescape"))`

Changed in version 3.6: Added the optional *errors* parameter.

11.4 stat — Interpreting stat() results

Source code: Lib/stat.py

The *stat* module defines constants and functions for interpreting the results of *os.stat()*, *os.fstat()* and *os.lstat()* (if they exist). For complete details about the `stat()`, `fstat()` and `lstat()` calls, consult the documentation for your system.

Changed in version 3.4: The stat module is backed by a C implementation.

The *stat* module defines the following functions to test for specific file types:

`stat.S_ISDIR`(*mode*)
Return non-zero if the mode is from a directory.

`stat.S_ISCHR`(*mode*)
Return non-zero if the mode is from a character special device file.

`stat.S_ISBLK`(*mode*)
Return non-zero if the mode is from a block special device file.

`stat.S_ISREG`(*mode*)
Return non-zero if the mode is from a regular file.

`stat.S_ISFIFO`(*mode*)
Return non-zero if the mode is from a FIFO (named pipe).

`stat.S_ISLNK`(*mode*)
Return non-zero if the mode is from a symbolic link.

`stat.S_ISSOCK`(*mode*)
Return non-zero if the mode is from a socket.

`stat.S_ISDOOR`(*mode*)
Return non-zero if the mode is from a door.

New in version 3.4.

`stat.S_ISPORT`(*mode*)
Return non-zero if the mode is from an event port.

New in version 3.4.

`stat.S_ISWHT`(*mode*)
Return non-zero if the mode is from a whiteout.

New in version 3.4.

Two additional functions are defined for more general manipulation of the file's mode:

`stat.S_IMODE`(*mode*)
Return the portion of the file's mode that can be set by *os.chmod()*—that is, the file's permission bits, plus the sticky bit, set-group-id, and set-user-id bits (on systems that support them).

`stat.S_IFMT`(*mode*)
Return the portion of the file's mode that describes the file type (used by the `S_IS*()` functions above).

Normally, you would use the `os.path.is*()` functions for testing the type of a file; the functions here are useful when you are doing multiple tests of the same file and wish to avoid the overhead of the `stat()` system call for each test. These are also useful when checking for information about a file that isn't handled by *os.path*, like the tests for block and character devices.

Example:

```
import os, sys
from stat import *

def walktree(top, callback):
    '''recursively descend the directory tree rooted at top,
```

(continues on next page)

(continued from previous page)

```
        calling the callback function for each regular file'''

    for f in os.listdir(top):
        pathname = os.path.join(top, f)
        mode = os.stat(pathname).st_mode
        if S_ISDIR(mode):
            # It's a directory, recurse into it
            walktree(pathname, callback)
        elif S_ISREG(mode):
            # It's a file, call the callback function
            callback(pathname)
        else:
            # Unknown file type, print a message
            print('Skipping %s' % pathname)

def visitfile(file):
    print('visiting', file)

if __name__ == '__main__':
    walktree(sys.argv[1], visitfile)
```

An additional utility function is provided to convert a file's mode in a human readable string:

stat.filemode(*mode*)
Convert a file's mode to a string of the form '-rwxrwxrwx'.

New in version 3.3.

Changed in version 3.4: The function supports *S_IFDOOR*, *S_IFPORT* and *S_IFWHT*.

All the variables below are simply symbolic indexes into the 10-tuple returned by *os.stat()*, *os.fstat()* or *os.lstat()*.

stat.ST_MODE
Inode protection mode.

stat.ST_INO
Inode number.

stat.ST_DEV
Device inode resides on.

stat.ST_NLINK
Number of links to the inode.

stat.ST_UID
User id of the owner.

stat.ST_GID
Group id of the owner.

stat.ST_SIZE
Size in bytes of a plain file; amount of data waiting on some special files.

stat.ST_ATIME
Time of last access.

stat.ST_MTIME
Time of last modification.

stat.ST_CTIME
: The "ctime" as reported by the operating system. On some systems (like Unix) is the time of the last metadata change, and, on others (like Windows), is the creation time (see platform documentation for details).

The interpretation of "file size" changes according to the file type. For plain files this is the size of the file in bytes. For FIFOs and sockets under most flavors of Unix (including Linux in particular), the "size" is the number of bytes waiting to be read at the time of the call to *os.stat()*, *os.fstat()*, or *os.lstat()*; this can sometimes be useful, especially for polling one of these special files after a non-blocking open. The meaning of the size field for other character and block devices varies more, depending on the implementation of the underlying system call.

The variables below define the flags used in the *ST_MODE* field.

Use of the functions above is more portable than use of the first set of flags:

stat.S_IFSOCK
: Socket.

stat.S_IFLNK
: Symbolic link.

stat.S_IFREG
: Regular file.

stat.S_IFBLK
: Block device.

stat.S_IFDIR
: Directory.

stat.S_IFCHR
: Character device.

stat.S_IFIFO
: FIFO.

stat.S_IFDOOR
: Door.

  New in version 3.4.

stat.S_IFPORT
: Event port.

  New in version 3.4.

stat.S_IFWHT
: Whiteout.

  New in version 3.4.

Note: *S_IFDOOR*, *S_IFPORT* or *S_IFWHT* are defined as 0 when the platform does not have support for the file types.

The following flags can also be used in the *mode* argument of *os.chmod()*:

stat.S_ISUID
: Set UID bit.

stat.S_ISGID
: Set-group-ID bit. This bit has several special uses. For a directory it indicates that BSD semantics is to be used for that directory: files created there inherit their group ID from the directory, not from the effective group ID of the creating process, and directories created there will also get the *S_ISGID* bit

set. For a file that does not have the group execution bit (*S_IXGRP*) set, the set-group-ID bit indicates mandatory file/record locking (see also *S_ENFMT*).

stat.S_ISVTX
Sticky bit. When this bit is set on a directory it means that a file in that directory can be renamed or deleted only by the owner of the file, by the owner of the directory, or by a privileged process.

stat.S_IRWXU
Mask for file owner permissions.

stat.S_IRUSR
Owner has read permission.

stat.S_IWUSR
Owner has write permission.

stat.S_IXUSR
Owner has execute permission.

stat.S_IRWXG
Mask for group permissions.

stat.S_IRGRP
Group has read permission.

stat.S_IWGRP
Group has write permission.

stat.S_IXGRP
Group has execute permission.

stat.S_IRWXO
Mask for permissions for others (not in group).

stat.S_IROTH
Others have read permission.

stat.S_IWOTH
Others have write permission.

stat.S_IXOTH
Others have execute permission.

stat.S_ENFMT
System V file locking enforcement. This flag is shared with *S_ISGID*: file/record locking is enforced on files that do not have the group execution bit (*S_IXGRP*) set.

stat.S_IREAD
Unix V7 synonym for *S_IRUSR*.

stat.S_IWRITE
Unix V7 synonym for *S_IWUSR*.

stat.S_IEXEC
Unix V7 synonym for *S_IXUSR*.

The following flags can be used in the *flags* argument of *os.chflags()*:

stat.UF_NODUMP
Do not dump the file.

stat.UF_IMMUTABLE
The file may not be changed.

stat.UF_APPEND
The file may only be appended to.

stat.UF_OPAQUE
The directory is opaque when viewed through a union stack.

stat.UF_NOUNLINK
The file may not be renamed or deleted.

stat.UF_COMPRESSED
The file is stored compressed (Mac OS X 10.6+).

stat.UF_HIDDEN
The file should not be displayed in a GUI (Mac OS X 10.5+).

stat.SF_ARCHIVED
The file may be archived.

stat.SF_IMMUTABLE
The file may not be changed.

stat.SF_APPEND
The file may only be appended to.

stat.SF_NOUNLINK
The file may not be renamed or deleted.

stat.SF_SNAPSHOT
The file is a snapshot file.

See the *BSD or Mac OS systems man page *chflags(2)* for more information.

On Windows, the following file attribute constants are available for use when testing bits in the st_file_attributes member returned by *os.stat()*. See the Windows API documentation for more detail on the meaning of these constants.

stat.FILE_ATTRIBUTE_ARCHIVE
stat.FILE_ATTRIBUTE_COMPRESSED
stat.FILE_ATTRIBUTE_DEVICE
stat.FILE_ATTRIBUTE_DIRECTORY
stat.FILE_ATTRIBUTE_ENCRYPTED
stat.FILE_ATTRIBUTE_HIDDEN
stat.FILE_ATTRIBUTE_INTEGRITY_STREAM
stat.FILE_ATTRIBUTE_NORMAL
stat.FILE_ATTRIBUTE_NOT_CONTENT_INDEXED
stat.FILE_ATTRIBUTE_NO_SCRUB_DATA
stat.FILE_ATTRIBUTE_OFFLINE
stat.FILE_ATTRIBUTE_READONLY
stat.FILE_ATTRIBUTE_REPARSE_POINT
stat.FILE_ATTRIBUTE_SPARSE_FILE
stat.FILE_ATTRIBUTE_SYSTEM
stat.FILE_ATTRIBUTE_TEMPORARY
stat.FILE_ATTRIBUTE_VIRTUAL
New in version 3.5.

11.5 filecmp — File and Directory Comparisons

Source code: Lib/filecmp.py

The *filecmp* module defines functions to compare files and directories, with various optional time/correctness trade-offs. For comparing files, see also the *difflib* module.

The *filecmp* module defines the following functions:

filecmp.cmp(*f1*, *f2*, *shallow=True*)
Compare the files named *f1* and *f2*, returning `True` if they seem equal, `False` otherwise.

If *shallow* is true, files with identical *os.stat()* signatures are taken to be equal. Otherwise, the contents of the files are compared.

Note that no external programs are called from this function, giving it portability and efficiency.

This function uses a cache for past comparisons and the results, with cache entries invalidated if the *os.stat()* information for the file changes. The entire cache may be cleared using *clear_cache()*.

filecmp.cmpfiles(*dir1*, *dir2*, *common*, *shallow=True*)
Compare the files in the two directories *dir1* and *dir2* whose names are given by *common*.

Returns three lists of file names: *match*, *mismatch*, *errors*. *match* contains the list of files that match, *mismatch* contains the names of those that don't, and *errors* lists the names of files which could not be compared. Files are listed in *errors* if they don't exist in one of the directories, the user lacks permission to read them or if the comparison could not be done for some other reason.

The *shallow* parameter has the same meaning and default value as for *filecmp.cmp()*.

For example, `cmpfiles('a', 'b', ['c', 'd/e'])` will compare `a/c` with `b/c` and `a/d/e` with `b/d/e`. `'c'` and `'d/e'` will each be in one of the three returned lists.

filecmp.clear_cache()
Clear the filecmp cache. This may be useful if a file is compared so quickly after it is modified that it is within the mtime resolution of the underlying filesystem.

New in version 3.4.

11.5.1 The `dircmp` class

class filecmp.dircmp(*a*, *b*, *ignore=None*, *hide=None*)
Construct a new directory comparison object, to compare the directories *a* and *b*. *ignore* is a list of names to ignore, and defaults to *filecmp.DEFAULT_IGNORES*. *hide* is a list of names to hide, and defaults to `[os.curdir, os.pardir]`.

The *dircmp* class compares files by doing *shallow* comparisons as described for *filecmp.cmp()*.

The *dircmp* class provides the following methods:

report()
Print (to *sys.stdout*) a comparison between *a* and *b*.

report_partial_closure()
Print a comparison between *a* and *b* and common immediate subdirectories.

report_full_closure()
Print a comparison between *a* and *b* and common subdirectories (recursively).

The *dircmp* class offers a number of interesting attributes that may be used to get various bits of information about the directory trees being compared.

Note that via **__getattr__**() hooks, all attributes are computed lazily, so there is no speed penalty if only those attributes which are lightweight to compute are used.

left
The directory *a*.

right
The directory *b*.

left_list
Files and subdirectories in *a*, filtered by *hide* and *ignore*.

right_list
Files and subdirectories in *b*, filtered by *hide* and *ignore*.

common
Files and subdirectories in both *a* and *b*.

left_only
Files and subdirectories only in *a*.

right_only
Files and subdirectories only in *b*.

common_dirs
Subdirectories in both *a* and *b*.

common_files
Files in both *a* and *b*.

common_funny
Names in both *a* and *b*, such that the type differs between the directories, or names for which *os.stat()* reports an error.

same_files
Files which are identical in both *a* and *b*, using the class's file comparison operator.

diff_files
Files which are in both *a* and *b*, whose contents differ according to the class's file comparison operator.

funny_files
Files which are in both *a* and *b*, but could not be compared.

subdirs
A dictionary mapping names in *common_dirs* to *dircmp* objects.

filecmp.DEFAULT_IGNORES
New in version 3.4.

List of directories ignored by *dircmp* by default.

Here is a simplified example of using the `subdirs` attribute to search recursively through two directories to show common different files:

```
>>> from filecmp import dircmp
>>> def print_diff_files(dcmp):
...     for name in dcmp.diff_files:
...         print("diff_file %s found in %s and %s" % (name, dcmp.left,
...               dcmp.right))
...     for sub_dcmp in dcmp.subdirs.values():
...         print_diff_files(sub_dcmp)
...
>>> dcmp = dircmp('dir1', 'dir2')
>>> print_diff_files(dcmp)
```

11.6 tempfile — Generate temporary files and directories

Source code: Lib/tempfile.py

This module creates temporary files and directories. It works on all supported platforms. *TemporaryFile*, *NamedTemporaryFile*, *TemporaryDirectory*, and *SpooledTemporaryFile* are high-level interfaces which provide automatic cleanup and can be used as context managers. *mkstemp()* and *mkdtemp()* are lower-level functions which require manual cleanup.

All the user-callable functions and constructors take additional arguments which allow direct control over the location and name of temporary files and directories. Files names used by this module include a string of random characters which allows those files to be securely created in shared temporary directories. To maintain backward compatibility, the argument order is somewhat odd; it is recommended to use keyword arguments for clarity.

The module defines the following user-callable items:

tempfile.TemporaryFile(*mode='w+b'*, *buffering=None*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*)

Return a *file-like object* that can be used as a temporary storage area. The file is created securely, using the same rules as *mkstemp()*. It will be destroyed as soon as it is closed (including an implicit close when the object is garbage collected). Under Unix, the directory entry for the file is either not created at all or is removed immediately after the file is created. Other platforms do not support this; your code should not rely on a temporary file created using this function having or not having a visible name in the file system.

The resulting object can be used as a context manager (see *Examples*). On completion of the context or destruction of the file object the temporary file will be removed from the filesystem.

The *mode* parameter defaults to `'w+b'` so that the file created can be read and written without being closed. Binary mode is used so that it behaves consistently on all platforms without regard for the data that is stored. *buffering*, *encoding* and *newline* are interpreted as for *open()*.

The *dir*, *prefix* and *suffix* parameters have the same meaning and defaults as with *mkstemp()*.

The returned object is a true file object on POSIX platforms. On other platforms, it is a file-like object whose `file` attribute is the underlying true file object.

The *os.O_TMPFILE* flag is used if it is available and works (Linux-specific, requires Linux kernel 3.11 or later).

Changed in version 3.5: The *os.O_TMPFILE* flag is now used if available.

tempfile.NamedTemporaryFile(*mode='w+b'*, *buffering=None*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*, *delete=True*)

This function operates exactly as *TemporaryFile()* does, except that the file is guaranteed to have a visible name in the file system (on Unix, the directory entry is not unlinked). That name can be retrieved from the `name` attribute of the returned file-like object. Whether the name can be used to open the file a second time, while the named temporary file is still open, varies across platforms (it can be so used on Unix; it cannot on Windows NT or later). If *delete* is true (the default), the file is deleted as soon as it is closed. The returned object is always a file-like object whose `file` attribute is the underlying true file object. This file-like object can be used in a `with` statement, just like a normal file.

tempfile.SpooledTemporaryFile(*max_size=0*, *mode='w+b'*, *buffering=None*, *encoding=None*, *newline=None*, *suffix=None*, *prefix=None*, *dir=None*)

This function operates exactly as *TemporaryFile()* does, except that data is spooled in memory until the file size exceeds *max_size*, or until the file's `fileno()` method is called, at which point the contents are written to disk and operation proceeds as with *TemporaryFile()*.

The resulting file has one additional method, `rollover()`, which causes the file to roll over to an on-disk file regardless of its size.

The returned object is a file-like object whose `_file` attribute is either an *io.BytesIO* or *io.StringIO* object (depending on whether binary or text *mode* was specified) or a true file object, depending on whether `rollover()` has been called. This file-like object can be used in a `with` statement, just like a normal file.

Changed in version 3.3: the truncate method now accepts a `size` argument.

`tempfile.TemporaryDirectory`(*suffix=None*, *prefix=None*, *dir=None*)

This function securely creates a temporary directory using the same rules as *mkdtemp()*. The resulting object can be used as a context manager (see *Examples*). On completion of the context or destruction of the temporary directory object the newly created temporary directory and all its contents are removed from the filesystem.

The directory name can be retrieved from the `name` attribute of the returned object. When the returned object is used as a context manager, the `name` will be assigned to the target of the `as` clause in the `with` statement, if there is one.

The directory can be explicitly cleaned up by calling the `cleanup()` method.

New in version 3.2.

`tempfile.mkstemp`(*suffix=None*, *prefix=None*, *dir=None*, *text=False*)

Creates a temporary file in the most secure manner possible. There are no race conditions in the file's creation, assuming that the platform properly implements the *os.O_EXCL* flag for *os.open()*. The file is readable and writable only by the creating user ID. If the platform uses permission bits to indicate whether a file is executable, the file is executable by no one. The file descriptor is not inherited by child processes.

Unlike *TemporaryFile()*, the user of *mkstemp()* is responsible for deleting the temporary file when done with it.

If *suffix* is not `None`, the file name will end with that suffix, otherwise there will be no suffix. *mkstemp()* does not put a dot between the file name and the suffix; if you need one, put it at the beginning of *suffix*.

If *prefix* is not `None`, the file name will begin with that prefix; otherwise, a default prefix is used. The default is the return value of *gettemppprefix()* or *gettemppprefixb()*, as appropriate.

If *dir* is not `None`, the file will be created in that directory; otherwise, a default directory is used. The default directory is chosen from a platform-dependent list, but the user of the application can control the directory location by setting the *TMPDIR*, *TEMP* or *TMP* environment variables. There is thus no guarantee that the generated filename will have any nice properties, such as not requiring quoting when passed to external commands via `os.popen()`.

If any of *suffix*, *prefix*, and *dir* are not `None`, they must be the same type. If they are bytes, the returned name will be bytes instead of str. If you want to force a bytes return value with otherwise default behavior, pass `suffix=b''`.

If *text* is specified, it indicates whether to open the file in binary mode (the default) or text mode. On some platforms, this makes no difference.

mkstemp() returns a tuple containing an OS-level handle to an open file (as would be returned by *os.open()*) and the absolute pathname of that file, in that order.

Changed in version 3.5: *suffix*, *prefix*, and *dir* may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. *suffix* and *prefix* now accept and default to `None` to cause an appropriate default value to be used.

`tempfile.mkdtemp`(*suffix=None*, *prefix=None*, *dir=None*)

Creates a temporary directory in the most secure manner possible. There are no race conditions in the directory's creation. The directory is readable, writable, and searchable only by the creating user ID.

The user of *mkdtemp()* is responsible for deleting the temporary directory and its contents when done with it.

The *prefix*, *suffix*, and *dir* arguments are the same as for *mkstemp()*.

mkdtemp() returns the absolute pathname of the new directory.

Changed in version 3.5: *suffix*, *prefix*, and *dir* may now be supplied in bytes in order to obtain a bytes return value. Prior to this, only str was allowed. *suffix* and *prefix* now accept and default to `None` to cause an appropriate default value to be used.

tempfile.gettempdir()

Return the name of the directory used for temporary files. This defines the default value for the *dir* argument to all functions in this module.

Python searches a standard list of directories to find one which the calling user can create files in. The list is:

1. The directory named by the `TMPDIR` environment variable.
2. The directory named by the `TEMP` environment variable.
3. The directory named by the `TMP` environment variable.
4. A platform-specific location:
   - On Windows, the directories `C:\TEMP`, `C:\TMP`, `\TEMP`, and `\TMP`, in that order.
   - On all other platforms, the directories `/tmp`, `/var/tmp`, and `/usr/tmp`, in that order.
5. As a last resort, the current working directory.

The result of this search is cached, see the description of *tempdir* below.

tempfile.gettempdirb()

Same as *gettempdir()* but the return value is in bytes.

New in version 3.5.

tempfile.gettempprefix()

Return the filename prefix used to create temporary files. This does not contain the directory component.

tempfile.gettempprefixb()

Same as *gettempprefix()* but the return value is in bytes.

New in version 3.5.

The module uses a global variable to store the name of the directory used for temporary files returned by *gettempdir()*. It can be set directly to override the selection process, but this is discouraged. All functions in this module take a *dir* argument which can be used to specify the directory and this is the recommended approach.

tempfile.tempdir

When set to a value other than `None`, this variable defines the default value for the *dir* argument to the functions defined in this module.

If `tempdir` is `None` (the default) at any call to any of the above functions except *gettempprefix()* it is initialized following the algorithm described in *gettempdir()*.

11.6.1 Examples

Here are some examples of typical usage of the *tempfile* module:

```
>>> import tempfile

# create a temporary file and write some data to it
>>> fp = tempfile.TemporaryFile()
>>> fp.write(b'Hello world!')
# read data from file
>>> fp.seek(0)
>>> fp.read()
b'Hello world!'
# close the file, it will be removed
>>> fp.close()

# create a temporary file using a context manager
>>> with tempfile.TemporaryFile() as fp:
...     fp.write(b'Hello world!')
...     fp.seek(0)
...     fp.read()
b'Hello world!'
>>>
# file is now closed and removed

# create a temporary directory using the context manager
>>> with tempfile.TemporaryDirectory() as tmpdirname:
...     print('created temporary directory', tmpdirname)
>>>
# directory and contents have been removed
```

11.6.2 Deprecated functions and variables

A historical way to create temporary files was to first generate a file name with the *mktemp()* function and then create a file using this name. Unfortunately this is not secure, because a different process may create a file with this name in the time between the call to *mktemp()* and the subsequent attempt to create the file by the first process. The solution is to combine the two steps and create the file immediately. This approach is used by *mkstemp()* and the other functions described above.

tempfile.mktemp(*suffix=''*, *prefix='tmp'*, *dir=None*)

Deprecated since version 2.3: Use *mkstemp()* instead.

Return an absolute pathname of a file that did not exist at the time the call is made. The *prefix*, *suffix*, and *dir* arguments are similar to those of *mkstemp()*, except that bytes file names, `suffix=None` and `prefix=None` are not supported.

Warning: Use of this function may introduce a security hole in your program. By the time you get around to doing anything with the file name it returns, someone else may have beaten you to the punch. *mktemp()* usage can be replaced easily with *NamedTemporaryFile()*, passing it the `delete=False` parameter:

```
>>> f = NamedTemporaryFile(delete=False)
>>> f.name
'/tmp/tmptjujjt'
>>> f.write(b"Hello World!\n")
13
>>> f.close()
>>> os.unlink(f.name)
>>> os.path.exists(f.name)
False
```

11.7 glob — Unix style pathname pattern expansion

Source code: Lib/glob.py

The *glob* module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell, although results are returned in arbitrary order. No tilde expansion is done, but `*`, `?`, and character ranges expressed with `[]` will be correctly matched. This is done by using the *os.scandir()* and *fnmatch.fnmatch()* functions in concert, and not by actually invoking a subshell. Note that unlike *fnmatch.fnmatch()*, *glob* treats filenames beginning with a dot (`.`) as special cases. (For tilde and shell variable expansion, use *os.path.expanduser()* and *os.path.expandvars()*.)

For a literal match, wrap the meta-characters in brackets. For example, `'[?]'` matches the character `'?'`.

See also:

The *pathlib* module offers high-level path objects.

`glob.glob`(*pathname*, ***, *recursive=False*)
: Return a possibly-empty list of path names that match *pathname*, which must be a string containing a path specification. *pathname* can be either absolute (like `/usr/src/Python-1.5/Makefile`) or relative (like `../../Tools/*/*.gif`), and can contain shell-style wildcards. Broken symlinks are included in the results (as in the shell).

  If *recursive* is true, the pattern "`**`" will match any files and zero or more directories and subdirectories. If the pattern is followed by an `os.sep`, only directories and subdirectories match.

  Note: Using the "`**`" pattern in large directory trees may consume an inordinate amount of time.

  Changed in version 3.5: Support for recursive globs using "`**`".

`glob.iglob`(*pathname*, ***, *recursive=False*)
: Return an *iterator* which yields the same values as *glob()* without actually storing them all simultaneously.

`glob.escape`(*pathname*)
: Escape all special characters (`'?'`, `'*'` and `'['`). This is useful if you want to match an arbitrary literal string that may have special characters in it. Special characters in drive/UNC sharepoints are not escaped, e.g. on Windows `escape('//?/c:/Quo vadis?.txt')` returns `'//?/c:/Quo vadis[?].txt'`.

  New in version 3.4.

For example, consider a directory containing the following files: `1.gif`, `2.txt`, `card.gif` and a subdirectory `sub` which contains only the file `3.txt`. *glob()* will produce the following results. Notice how any leading components of the path are preserved.

```
>>> import glob
>>> glob.glob('./[0-9].*')
['./1.gif', './2.txt']
>>> glob.glob('*.gif')
['1.gif', 'card.gif']
>>> glob.glob('?.gif')
```

(continues on next page)

(continued from previous page)

```
['1.gif']
>>> glob.glob('**/*.txt', recursive=True)
['2.txt', 'sub/3.txt']
>>> glob.glob('./**/', recursive=True)
['./', './sub/']
```

If the directory contains files starting with `.` they won't be matched by default. For example, consider a directory containing `card.gif` and `.card.gif`:

```
>>> import glob
>>> glob.glob('*.gif')
['card.gif']
>>> glob.glob('.c*')
['.card.gif']
```

See also:

Module *fnmatch* Shell-style filename (not path) expansion

11.8 fnmatch — Unix filename pattern matching

Source code: Lib/fnmatch.py

This module provides support for Unix shell-style wildcards, which are *not* the same as regular expressions (which are documented in the *re* module). The special characters used in shell-style wildcards are:

Pattern	Meaning
`*`	matches everything
`?`	matches any single character
`[seq]`	matches any character in *seq*
`[!seq]`	matches any character not in *seq*

For a literal match, wrap the meta-characters in brackets. For example, `'[?]'` matches the character `'?'`.

Note that the filename separator (`'/'` on Unix) is *not* special to this module. See module *glob* for pathname expansion (*glob* uses *filter()* to match pathname segments). Similarly, filenames starting with a period are not special for this module, and are matched by the `*` and `?` patterns.

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

Test whether the *filename* string matches the *pattern* string, returning *True* or *False*. Both parameters are case-normalized using *os.path.normcase()*. *fnmatchcase()* can be used to perform a case-sensitive comparison, regardless of whether that's standard for the operating system.

This example will print all file names in the current directory with the extension `.txt`:

```
import fnmatch
import os

for file in os.listdir('.'):
    if fnmatch.fnmatch(file, '*.txt'):
        print(file)
```

fnmatch.**fnmatchcase**(*filename*, *pattern*)
Test whether *filename* matches *pattern*, returning *True* or *False*; the comparison is case-sensitive and does not apply *os.path.normcase()*.

fnmatch.**filter**(*names*, *pattern*)
Return the subset of the list of *names* that match *pattern*. It is the same as `[n for n in names if fnmatch(n, pattern)]`, but implemented more efficiently.

fnmatch.**translate**(*pattern*)
Return the shell-style *pattern* converted to a regular expression for using with *re.match()*.

Example:

```
>>> import fnmatch, re
>>>
>>> regex = fnmatch.translate('*.txt')
>>> regex
'(?s:.*\\.txt)\\Z'
>>> reobj = re.compile(regex)
>>> reobj.match('foobar.txt')
<re.Match object; span=(0, 10), match='foobar.txt'>
```

See also:

Module *glob* Unix shell-style path expansion.

11.9 linecache — Random access to text lines

Source code: Lib/linecache.py

The *linecache* module allows one to get any line from a Python source file, while attempting to optimize internally, using a cache, the common case where many lines are read from a single file. This is used by the *traceback* module to retrieve source lines for inclusion in the formatted traceback.

The *tokenize.open()* function is used to open files. This function uses *tokenize.detect_encoding()* to get the encoding of the file; in the absence of an encoding token, the file encoding defaults to UTF-8.

The *linecache* module defines the following functions:

linecache.**getline**(*filename*, *lineno*, *module_globals=None*)
Get line *lineno* from file named *filename*. This function will never raise an exception — it will return `''` on errors (the terminating newline character will be included for lines that are found).

If a file named *filename* is not found, the function will look for it in the module search path, `sys.path`, after first checking for a **PEP 302** `__loader__` in *module_globals*, in case the module was imported from a zipfile or other non-filesystem import source.

linecache.**clearcache**()
Clear the cache. Use this function if you no longer need lines from files previously read using *getline()*.

linecache.**checkcache**(*filename=None*)
Check the cache for validity. Use this function if files in the cache may have changed on disk, and you require the updated version. If *filename* is omitted, it will check all the entries in the cache.

linecache.**lazycache**(*filename*, *module_globals*)
Capture enough detail about a non-file-based module to permit getting its lines later via *getline()* even if *module_globals* is `None` in the later call. This avoids doing I/O until a line is actually needed, without having to carry the module globals around indefinitely.

New in version 3.5.

Example:

```
>>> import linecache
>>> linecache.getline(linecache.__file__, 8)
'import sys\n'
```

11.10 shutil — High-level file operations

Source code: Lib/shutil.py

The *shutil* module offers a number of high-level operations on files and collections of files. In particular, functions are provided which support file copying and removal. For operations on individual files, see also the *os* module.

Warning: Even the higher-level file copying functions (*shutil.copy()*, *shutil.copy2()*) cannot copy all file metadata.

On POSIX platforms, this means that file owner and group are lost as well as ACLs. On Mac OS, the resource fork and other metadata are not used. This means that resources will be lost and file type and creator codes will not be correct. On Windows, file owners, ACLs and alternate data streams are not copied.

11.10.1 Directory and files operations

shutil.copyfileobj(*fsrc*, *fdst*[, *length*])

Copy the contents of the file-like object *fsrc* to the file-like object *fdst*. The integer *length*, if given, is the buffer size. In particular, a negative *length* value means to copy the data without looping over the source data in chunks; by default the data is read in chunks to avoid uncontrolled memory consumption. Note that if the current file position of the *fsrc* object is not 0, only the contents from the current file position to the end of the file will be copied.

shutil.copyfile(*src*, *dst*, ***, *follow_symlinks=True*)

Copy the contents (no metadata) of the file named *src* to a file named *dst* and return *dst*. *src* and *dst* are path names given as strings. *dst* must be the complete target file name; look at *shutil.copy()* for a copy that accepts a target directory path. If *src* and *dst* specify the same file, *SameFileError* is raised.

The destination location must be writable; otherwise, an *OSError* exception will be raised. If *dst* already exists, it will be replaced. Special files such as character or block devices and pipes cannot be copied with this function.

If *follow_symlinks* is false and *src* is a symbolic link, a new symbolic link will be created instead of copying the file *src* points to.

Changed in version 3.3: *IOError* used to be raised instead of *OSError*. Added *follow_symlinks* argument. Now returns *dst*.

Changed in version 3.4: Raise *SameFileError* instead of *Error*. Since the former is a subclass of the latter, this change is backward compatible.

exception shutil.SameFileError
This exception is raised if source and destination in *copyfile()* are the same file.

New in version 3.4.

shutil.copymode(*src*, *dst*, ***, *follow_symlinks=True*)
Copy the permission bits from *src* to *dst*. The file contents, owner, and group are unaffected. *src* and *dst* are path names given as strings. If *follow_symlinks* is false, and both *src* and *dst* are symbolic links, *copymode()* will attempt to modify the mode of *dst* itself (rather than the file it points to). This functionality is not available on every platform; please see *copystat()* for more information. If *copymode()* cannot modify symbolic links on the local platform, and it is asked to do so, it will do nothing and return.

Changed in version 3.3: Added *follow_symlinks* argument.

shutil.copystat(*src*, *dst*, ***, *follow_symlinks=True*)
Copy the permission bits, last access time, last modification time, and flags from *src* to *dst*. On Linux, *copystat()* also copies the "extended attributes" where possible. The file contents, owner, and group are unaffected. *src* and *dst* are path names given as strings.

If *follow_symlinks* is false, and *src* and *dst* both refer to symbolic links, *copystat()* will operate on the symbolic links themselves rather than the files the symbolic links refer to—reading the information from the *src* symbolic link, and writing the information to the *dst* symbolic link.

Note: Not all platforms provide the ability to examine and modify symbolic links. Python itself can tell you what functionality is locally available.

- If `os.chmod in os.supports_follow_symlinks` is `True`, *copystat()* can modify the permission bits of a symbolic link.
- If `os.utime in os.supports_follow_symlinks` is `True`, *copystat()* can modify the last access and modification times of a symbolic link.
- If `os.chflags in os.supports_follow_symlinks` is `True`, *copystat()* can modify the flags of a symbolic link. (`os.chflags` is not available on all platforms.)

On platforms where some or all of this functionality is unavailable, when asked to modify a symbolic link, *copystat()* will copy everything it can. *copystat()* never returns failure.

Please see *os.supports_follow_symlinks* for more information.

Changed in version 3.3: Added *follow_symlinks* argument and support for Linux extended attributes.

shutil.copy(*src*, *dst*, ***, *follow_symlinks=True*)
Copies the file *src* to the file or directory *dst*. *src* and *dst* should be strings. If *dst* specifies a directory, the file will be copied into *dst* using the base filename from *src*. Returns the path to the newly created file.

If *follow_symlinks* is false, and *src* is a symbolic link, *dst* will be created as a symbolic link. If *follow_symlinks* is true and *src* is a symbolic link, *dst* will be a copy of the file *src* refers to.

copy() copies the file data and the file's permission mode (see *os.chmod()*). Other metadata, like the file's creation and modification times, is not preserved. To preserve all file metadata from the original, use *copy2()* instead.

Changed in version 3.3: Added *follow_symlinks* argument. Now returns path to the newly created file.

shutil.copy2(*src*, *dst*, ***, *follow_symlinks=True*)
Identical to *copy()* except that *copy2()* also attempts to preserve file metadata.

When *follow_symlinks* is false, and *src* is a symbolic link, *copy2()* attempts to copy all metadata from the *src* symbolic link to the newly-created *dst* symbolic link. However, this functionality is not

available on all platforms. On platforms where some or all of this functionality is unavailable, *copy2()* will preserve all the metadata it can; *copy2()* never returns failure.

copy2() uses *copystat()* to copy the file metadata. Please see *copystat()* for more information about platform support for modifying symbolic link metadata.

Changed in version 3.3: Added *follow_symlinks* argument, try to copy extended file system attributes too (currently Linux only). Now returns path to the newly created file.

shutil.ignore_patterns(**patterns*)

This factory function creates a function that can be used as a callable for *copytree()*'s *ignore* argument, ignoring files and directories that match one of the glob-style *patterns* provided. See the example below.

shutil.copytree(*src*, *dst*, *symlinks=False*, *ignore=None*, *copy_function=copy2*, *ignore_dangling_symlinks=False*)

Recursively copy an entire directory tree rooted at *src*, returning the destination directory. The destination directory, named by *dst*, must not already exist; it will be created as well as missing parent directories. Permissions and times of directories are copied with *copystat()*, individual files are copied using *shutil.copy2()*.

If *symlinks* is true, symbolic links in the source tree are represented as symbolic links in the new tree and the metadata of the original links will be copied as far as the platform allows; if false or omitted, the contents and metadata of the linked files are copied to the new tree.

When *symlinks* is false, if the file pointed by the symlink doesn't exist, an exception will be added in the list of errors raised in an *Error* exception at the end of the copy process. You can set the optional *ignore_dangling_symlinks* flag to true if you want to silence this exception. Notice that this option has no effect on platforms that don't support *os.symlink()*.

If *ignore* is given, it must be a callable that will receive as its arguments the directory being visited by *copytree()*, and a list of its contents, as returned by *os.listdir()*. Since *copytree()* is called recursively, the *ignore* callable will be called once for each directory that is copied. The callable must return a sequence of directory and file names relative to the current directory (i.e. a subset of the items in its second argument); these names will then be ignored in the copy process. *ignore_patterns()* can be used to create such a callable that ignores names based on glob-style patterns.

If exception(s) occur, an *Error* is raised with a list of reasons.

If *copy_function* is given, it must be a callable that will be used to copy each file. It will be called with the source path and the destination path as arguments. By default, *shutil.copy2()* is used, but any function that supports the same signature (like *shutil.copy()*) can be used.

Changed in version 3.3: Copy metadata when *symlinks* is false. Now returns *dst*.

Changed in version 3.2: Added the *copy_function* argument to be able to provide a custom copy function. Added the *ignore_dangling_symlinks* argument to silent dangling symlinks errors when *symlinks* is false.

shutil.rmtree(*path*, *ignore_errors=False*, *onerror=None*)

Delete an entire directory tree; *path* must point to a directory (but not a symbolic link to a directory). If *ignore_errors* is true, errors resulting from failed removals will be ignored; if false or omitted, such errors are handled by calling a handler specified by *onerror* or, if that is omitted, they raise an exception.

Note: On platforms that support the necessary fd-based functions a symlink attack resistant version of *rmtree()* is used by default. On other platforms, the *rmtree()* implementation is susceptible to a symlink attack: given proper timing and circumstances, attackers can manipulate symlinks on the filesystem to delete files they wouldn't be able to access otherwise. Applications can use the *rmtree.avoids_symlink_attacks* function attribute to determine which case applies.

If *onerror* is provided, it must be a callable that accepts three parameters: *function*, *path*, and *excinfo*.

The first parameter, *function*, is the function which raised the exception; it depends on the platform and implementation. The second parameter, *path*, will be the path name passed to *function*. The third parameter, *excinfo*, will be the exception information returned by *sys.exc_info()*. Exceptions raised by *onerror* will not be caught.

Changed in version 3.3: Added a symlink attack resistant version that is used automatically if platform supports fd-based functions.

rmtree.avoids_symlink_attacks
Indicates whether the current platform and implementation provides a symlink attack resistant version of *rmtree()*. Currently this is only true for platforms supporting fd-based directory access functions.

New in version 3.3.

shutil.move(*src*, *dst*, *copy_function=copy2*)
Recursively move a file or directory (*src*) to another location (*dst*) and return the destination.

If the destination is an existing directory, then *src* is moved inside that directory. If the destination already exists but is not a directory, it may be overwritten depending on *os.rename()* semantics.

If the destination is on the current filesystem, then *os.rename()* is used. Otherwise, *src* is copied to *dst* using *copy_function* and then removed. In case of symlinks, a new symlink pointing to the target of *src* will be created in or as *dst* and *src* will be removed.

If *copy_function* is given, it must be a callable that takes two arguments *src* and *dst*, and will be used to copy *src* to *dest* if *os.rename()* cannot be used. If the source is a directory, *copytree()* is called, passing it the `copy_function()`. The default *copy_function* is *copy2()*. Using *copy()* as the *copy_function* allows the move to succeed when it is not possible to also copy the metadata, at the expense of not copying any of the metadata.

Changed in version 3.3: Added explicit symlink handling for foreign filesystems, thus adapting it to the behavior of GNU's **mv**. Now returns *dst*.

Changed in version 3.5: Added the *copy_function* keyword argument.

shutil.disk_usage(*path*)
Return disk usage statistics about the given path as a *named tuple* with the attributes *total*, *used* and *free*, which are the amount of total, used and free space, in bytes. On Windows, *path* must be a directory; on Unix, it can be a file or directory.

New in version 3.3.

Availability: Unix, Windows.

shutil.chown(*path*, *user=None*, *group=None*)
Change owner *user* and/or *group* of the given *path*.

user can be a system user name or a uid; the same applies to *group*. At least one argument is required.

See also *os.chown()*, the underlying function.

Availability: Unix.

New in version 3.3.

shutil.which(*cmd*, *mode=os.F_OK | os.X_OK*, *path=None*)
Return the path to an executable which would be run if the given *cmd* was called. If no *cmd* would be called, return `None`.

mode is a permission mask passed to *os.access()*, by default determining if the file exists and executable.

When no *path* is specified, the results of *os.environ()* are used, returning either the "PATH" value or a fallback of *os.defpath*.

On Windows, the current directory is always prepended to the *path* whether or not you use the default or provide your own, which is the behavior the command shell uses when finding executables. Additionally, when finding the *cmd* in the *path*, the `PATHEXT` environment variable is checked. For example, if you call `shutil.which("python")`, *which()* will search `PATHEXT` to know that it should look for `python.exe` within the *path* directories. For example, on Windows:

```
>>> shutil.which("python")
'C:\\Python33\\python.EXE'
```

New in version 3.3.

exception shutil.Error

This exception collects exceptions that are raised during a multi-file operation. For *copytree()*, the exception argument is a list of 3-tuples (*srcname*, *dstname*, *exception*).

copytree example

This example is the implementation of the *copytree()* function, described above, with the docstring omitted. It demonstrates many of the other functions provided by this module.

```
def copytree(src, dst, symlinks=False):
    names = os.listdir(src)
    os.makedirs(dst)
    errors = []
    for name in names:
        srcname = os.path.join(src, name)
        dstname = os.path.join(dst, name)
        try:
            if symlinks and os.path.islink(srcname):
                linkto = os.readlink(srcname)
                os.symlink(linkto, dstname)
            elif os.path.isdir(srcname):
                copytree(srcname, dstname, symlinks)
            else:
                copy2(srcname, dstname)
            # XXX What about devices, sockets etc.?
        except OSError as why:
            errors.append((srcname, dstname, str(why)))
        # catch the Error from the recursive copytree so that we can
        # continue with other files
        except Error as err:
            errors.extend(err.args[0])
    try:
        copystat(src, dst)
    except OSError as why:
        # can't copy file access times on Windows
        if why.winerror is None:
            errors.extend((src, dst, str(why)))
    if errors:
        raise Error(errors)
```

Another example that uses the *ignore_patterns()* helper:

```
from shutil import copytree, ignore_patterns

copytree(source, destination, ignore=ignore_patterns('*.pyc', 'tmp*'))
```

This will copy everything except `.pyc` files and files or directories whose name starts with `tmp`.

Another example that uses the *ignore* argument to add a logging call:

```
from shutil import copytree
import logging

def _logpath(path, names):
    logging.info('Working in %s', path)
    return []   # nothing will be ignored

copytree(source, destination, ignore=_logpath)
```

rmtree example

This example shows how to remove a directory tree on Windows where some of the files have their read-only bit set. It uses the onerror callback to clear the readonly bit and reattempt the remove. Any subsequent failure will propagate.

```
import os, stat
import shutil

def remove_readonly(func, path, _):
    "Clear the readonly bit and reattempt the removal"
    os.chmod(path, stat.S_IWRITE)
    func(path)

shutil.rmtree(directory, onerror=remove_readonly)
```

11.10.2 Archiving operations

New in version 3.2.

Changed in version 3.5: Added support for the *xztar* format.

High-level utilities to create and read compressed and archived files are also provided. They rely on the *zipfile* and *tarfile* modules.

shutil.make_archive(*base_name*, *format*[, *root_dir*[, *base_dir*[, *verbose*[, *dry_run*[, *owner*[, *group*[, *logger*]]]]]]])

Create an archive file (such as zip or tar) and return its name.

base_name is the name of the file to create, including the path, minus any format-specific extension. *format* is the archive format: one of "zip" (if the *zlib* module is available), "tar", "gztar" (if the *zlib* module is available), "bztar" (if the *bz2* module is available), or "xztar" (if the *lzma* module is available).

root_dir is a directory that will be the root directory of the archive; for example, we typically chdir into *root_dir* before creating the archive.

base_dir is the directory where we start archiving from; i.e. *base_dir* will be the common prefix of all files and directories in the archive.

root_dir and *base_dir* both default to the current directory.

If *dry_run* is true, no archive is created, but the operations that would be executed are logged to *logger*.

owner and *group* are used when creating a tar archive. By default, uses the current owner and group.

logger must be an object compatible with **PEP 282**, usually an instance of *logging.Logger*.

The *verbose* argument is unused and deprecated.

`shutil.get_archive_formats()`
Return a list of supported formats for archiving. Each element of the returned sequence is a tuple `(name, description)`.

By default *shutil* provides these formats:

- *zip*: ZIP file (if the *zlib* module is available).
- *tar*: uncompressed tar file.
- *gztar*: gzip'ed tar-file (if the *zlib* module is available).
- *bztar*: bzip2'ed tar-file (if the *bz2* module is available).
- *xztar*: xz'ed tar-file (if the *lzma* module is available).

You can register new formats or provide your own archiver for any existing formats, by using *register_archive_format()*.

`shutil.register_archive_format(`*name*, *function*[, *extra_args*[, *description*]]`)`
Register an archiver for the format *name*.

function is the callable that will be used to unpack archives. The callable will receive the *base_name* of the file to create, followed by the *base_dir* (which defaults to *os.curdir*) to start archiving from. Further arguments are passed as keyword arguments: *owner*, *group*, *dry_run* and *logger* (as passed in *make_archive()*).

If given, *extra_args* is a sequence of `(name, value)` pairs that will be used as extra keywords arguments when the archiver callable is used.

description is used by *get_archive_formats()* which returns the list of archivers. Defaults to an empty string.

`shutil.unregister_archive_format(`*name*`)`
Remove the archive format *name* from the list of supported formats.

`shutil.unpack_archive(`*filename*[, *extract_dir*[, *format*]]`)`
Unpack an archive. *filename* is the full path of the archive.

extract_dir is the name of the target directory where the archive is unpacked. If not provided, the current working directory is used.

format is the archive format: one of "zip", "tar", "gztar", "bztar", or "xztar". Or any other format registered with *register_unpack_format()*. If not provided, *unpack_archive()* will use the archive file name extension and see if an unpacker was registered for that extension. In case none is found, a *ValueError* is raised.

Changed in version 3.7: Accepts a *path-like object* for *filename* and *extract_dir*.

`shutil.register_unpack_format(`*name*, *extensions*, *function*[, *extra_args*[, *description*]]`)`
Registers an unpack format. *name* is the name of the format and *extensions* is a list of extensions corresponding to the format, like `.zip` for Zip files.

function is the callable that will be used to unpack archives. The callable will receive the path of the archive, followed by the directory the archive must be extracted to.

When provided, *extra_args* is a sequence of `(name, value)` tuples that will be passed as keywords arguments to the callable.

description can be provided to describe the format, and will be returned by the *get_unpack_formats()* function.

`shutil.unregister_unpack_format(`*name*`)`
Unregister an unpack format. *name* is the name of the format.

`shutil.get_unpack_formats()`
Return a list of all registered formats for unpacking. Each element of the returned sequence is a tuple `(name, extensions, description)`.

By default *shutil* provides these formats:

- *zip*: ZIP file (unpacking compressed files works only if the corresponding module is available).
- *tar*: uncompressed tar file.
- *gztar*: gzip'ed tar-file (if the *zlib* module is available).
- *bztar*: bzip2'ed tar-file (if the *bz2* module is available).
- *xztar*: xz'ed tar-file (if the *lzma* module is available).

You can register new formats or provide your own unpacker for any existing formats, by using *register_unpack_format()*.

Archiving example

In this example, we create a gzip'ed tar-file archive containing all files found in the `.ssh` directory of the user:

```
>>> from shutil import make_archive
>>> import os
>>> archive_name = os.path.expanduser(os.path.join('~', 'myarchive'))
>>> root_dir = os.path.expanduser(os.path.join('~', '.ssh'))
>>> make_archive(archive_name, 'gztar', root_dir)
'/Users/tarek/myarchive.tar.gz'
```

The resulting archive contains:

```
$ tar -tzvf /Users/tarek/myarchive.tar.gz
drwx------ tarek/staff       0 2010-02-01 16:23:40 ./
-rw-r--r-- tarek/staff     609 2008-06-09 13:26:54 ./authorized_keys
-rwxr-xr-x tarek/staff      65 2008-06-09 13:26:54 ./config
-rwx------ tarek/staff     668 2008-06-09 13:26:54 ./id_dsa
-rwxr-xr-x tarek/staff     609 2008-06-09 13:26:54 ./id_dsa.pub
-rw------- tarek/staff    1675 2008-06-09 13:26:54 ./id_rsa
-rw-r--r-- tarek/staff     397 2008-06-09 13:26:54 ./id_rsa.pub
-rw-r--r-- tarek/staff   37192 2010-02-06 18:23:10 ./known_hosts
```

11.10.3 Querying the size of the output terminal

`shutil.get_terminal_size(`*fallback=(columns, lines)*`)`
Get the size of the terminal window.

For each of the two dimensions, the environment variable, `COLUMNS` and `LINES` respectively, is checked. If the variable is defined and the value is a positive integer, it is used.

When `COLUMNS` or `LINES` is not defined, which is the common case, the terminal connected to *sys.__stdout__* is queried by invoking *os.get_terminal_size()*.

If the terminal size cannot be successfully queried, either because the system doesn't support querying, or because we are not connected to a terminal, the value given in `fallback` parameter is used. `fallback` defaults to `(80, 24)` which is the default size used by many terminal emulators.

The value returned is a named tuple of type *os.terminal_size*.

See also: The Single UNIX Specification, Version 2, Other Environment Variables.

New in version 3.3.

11.11 `macpath` — Mac OS 9 path manipulation functions

Source code: Lib/macpath.py

Deprecated since version 3.7, will be removed in version 3.8.

This module is the Mac OS 9 (and earlier) implementation of the *os.path* module. It can be used to manipulate old-style Macintosh pathnames on Mac OS X (or any other platform).

The following functions are available in this module: `normcase()`, `normpath()`, `isabs()`, `join()`, `split()`, `isdir()`, `isfile()`, `walk()`, `exists()`. For other functions available in *os.path* dummy counterparts are available.

See also:

Module *os* Operating system interfaces, including functions to work with files at a lower level than Python *file objects*.

Module *io* Python's built-in I/O library, including both abstract classes and some concrete classes such as file I/O.

Built-in function *open()* The standard way to open files for reading and writing with Python.