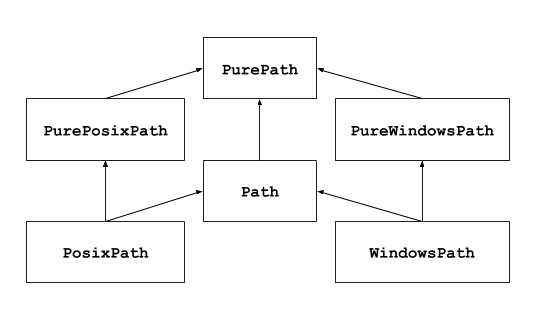
# [pathlib](https://docs.python.org/3/library/pathlib.html" \l "module-pathlib" \o "pathlib: Object-oriented filesystem paths) — Object-oriented filesystem paths

*New in version 3.4.*

**Source code:** [Lib/pathlib.py](https://github.com/python/cpython/tree/3.9/Lib/pathlib.py)

This module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between [pure paths](https://docs.python.org/3/library/pathlib.html#pure-paths), which provide purely computational operations without I/O, and [concrete paths](https://docs.python.org/3/library/pathlib.html#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](https://docs.python.org/3/library/pathlib.html#pathlib.Path) is most likely what you need. It instantiates a [concrete path](https://docs.python.org/3/library/pathlib.html#concrete-paths) 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](https://docs.python.org/3/library/pathlib.html" \l "pathlib.WindowsPath" \o "pathlib.WindowsPath) when running on Unix, but you can instantiate [PureWindowsPath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PureWindowsPath" \o "pathlib.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**](https://www.python.org/dev/peps/pep-0428): The pathlib module – object-oriented filesystem paths.

**See also**

For low-level path manipulation on strings, you can also use the [os.path](https://docs.python.org/3/library/os.path.html" \l "module-os.path" \o "os.path: Operations on pathnames.) module.

## **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'

## **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](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePosixPath" \o "pathlib.PurePosixPath) or a [PureWindowsPath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PureWindowsPath" \o "pathlib.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](https://docs.python.org/3/library/os.html" \l "os.PathLike" \o "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()](https://docs.python.org/3/library/os.path.html" \l "os.path.join" \o "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](https://docs.python.org/3/library/os.html" \l "os.PathLike" \o "os.PathLike) interface, allowing them to be used anywhere the interface is accepted.

*Changed in version 3.6:*Added support for the [os.PathLike](https://docs.python.org/3/library/os.html" \l "os.PathLike" \o "os.PathLike) interface.

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

A subclass of [PurePath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePath" \o "pathlib.PurePath), this path flavour represents non-Windows filesystem paths:

>>>

**>>>** PurePosixPath('/etc')

PurePosixPath('/etc')

pathsegments is specified similarly to [PurePath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePath" \o "pathlib.PurePath).

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

A subclass of [PurePath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePath" \o "pathlib.PurePath), this path flavour represents Windows filesystem paths:

>>>

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

PureWindowsPath('c:/Program Files')

pathsegments is specified similarly to [PurePath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePath" \o "pathlib.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()](https://docs.python.org/3/library/os.path.html" \l "os.path.join" \o "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](https://docs.python.org/3/library/os.html" \l "os.PathLike" \o "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')

**>>>** str(p)

'c:\\Program Files'

Similarly, calling [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) on a path gives the raw filesystem path as a bytes object, as encoded by [os.fsencode()](https://docs.python.org/3/library/os.html" \l "os.fsencode" \o "os.fsencode):

>>>

**>>>** bytes(p)

b'/etc'

**Note**

Calling [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) is only recommended under Unix. Under Windows, the unicode form is the canonical representation of filesystem paths.

### **Accessing individual parts**[**¶**](https://docs.python.org/3/library/pathlib.html#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()](https://docs.python.org/3/library/pathlib.html" \l "pathlib.Path.resolve" \o "pathlib.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](https://docs.python.org/3/library/exceptions.html" \l "ValueError" \o "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\_relative\_to**(\*other)

Return whether or not this path is relative to the other path.

>>>

**>>>** p = PurePath('/etc/passwd')

**>>>** p.is\_relative\_to('/etc')

True

**>>>** p.is\_relative\_to('/usr')

False

*New in version 3.9.*

PurePath.**is\_reserved**()

With [PureWindowsPath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PureWindowsPath" \o "pathlib.PureWindowsPath), return True if the path is considered reserved under Windows, False otherwise. With [PurePosixPath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePosixPath" \o "pathlib.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')

**>>>** 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 follows platform defaults:

>>>

**>>>** PurePosixPath('b.py').match('\*.PY')

False

**>>>** 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' is not in the subpath of '/usr' OR one path is relative and the other absolute.

NOTE: This function is part of [PurePath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePath" \o "pathlib.PurePath) and works with strings. It does not check or access the underlying file structure.

PurePath.**with\_name**(name)

Return a new path with the [name](https://docs.python.org/3/library/pathlib.html#pathlib.PurePath.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):

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\_stem**(stem)

Return a new path with the [stem](https://docs.python.org/3/library/pathlib.html#pathlib.PurePath.stem) changed. If the original path doesn’t have a name, ValueError is raised:

>>>

**>>>** p = PureWindowsPath('c:/Downloads/draft.txt')

**>>>** p.with\_stem('final')

PureWindowsPath('c:/Downloads/final.txt')

**>>>** p = PureWindowsPath('c:/Downloads/pathlib.tar.gz')

**>>>** p.with\_stem('lib')

PureWindowsPath('c:/Downloads/lib.gz')

**>>>** p = PureWindowsPath('c:/')

**>>>** p.with\_stem('')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "/home/antoine/cpython/default/Lib/pathlib.py", line 861, in with\_stem

**return** self.with\_name(stem + self.suffix)

File "/home/antoine/cpython/default/Lib/pathlib.py", line 851, in with\_name

**raise** ValueError("*%r* has an empty name" % (self,))

ValueError: PureWindowsPath('c:/') has an empty name

*New in version 3.9.*

PurePath.**with\_suffix**(suffix)

Return a new path with the [suffix](https://docs.python.org/3/library/pathlib.html#pathlib.PurePath.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')

## **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](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePath" \o "pathlib.PurePath), this class represents concrete paths of the system’s path flavour (instantiating it creates either a [PosixPath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PosixPath" \o "pathlib.PosixPath) or a [WindowsPath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.WindowsPath" \o "pathlib.WindowsPath)):

>>>

**>>>** Path('setup.py')

PosixPath('setup.py')

pathsegments is specified similarly to [PurePath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePath" \o "pathlib.PurePath).

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

A subclass of [Path](https://docs.python.org/3/library/pathlib.html#pathlib.Path) and [PurePosixPath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePosixPath" \o "pathlib.PurePosixPath), this class represents concrete non-Windows filesystem paths:

>>>

**>>>** PosixPath('/etc')

PosixPath('/etc')

pathsegments is specified similarly to [PurePath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePath" \o "pathlib.PurePath).

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

A subclass of [Path](https://docs.python.org/3/library/pathlib.html#pathlib.Path) and [PureWindowsPath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PureWindowsPath" \o "pathlib.PureWindowsPath), this class represents concrete Windows filesystem paths:

>>>

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

WindowsPath('c:/Program Files')

pathsegments is specified similarly to [PurePath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePath" \o "pathlib.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](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "OSError) if a system call fails (for example because the path doesn’t exist).

*Changed in version 3.8:*[exists()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.exists), [is\_dir()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.is_dir), [is\_file()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.is_file), [is\_mount()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.is_mount), [is\_symlink()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.is_symlink), [is\_block\_device()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.is_block_device), [is\_char\_device()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.is_char_device), [is\_fifo()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.is_fifo), [is\_socket()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.is_socket) now return False instead of raising an exception for paths that contain characters unrepresentable at the OS level.

classmethod Path.**cwd**()

Return a new path object representing the current directory (as returned by [os.getcwd()](https://docs.python.org/3/library/os.html" \l "os.getcwd" \o "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()](https://docs.python.org/3/library/os.path.html" \l "os.path.expanduser" \o "os.path.expanduser) with ~ construct):

>>>

**>>>** Path.home()

PosixPath('/home/antoine')

*New in version 3.5.*

Path.**stat**()

Return a [os.stat\_result](https://docs.python.org/3/library/os.html" \l "os.stat_result" \o "os.stat_result) object containing information about this path, like [os.stat()](https://docs.python.org/3/library/os.html" \l "os.stat" \o "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()](https://docs.python.org/3/library/os.html" \l "os.chmod" \o "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()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.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()](https://docs.python.org/3/library/os.path.html" \l "os.path.expanduser" \o "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 relative 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.

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) pathlib.Path.glob with arguments self, pattern.

Path.**group**()

Return the name of the group owning the file. [KeyError](https://docs.python.org/3/library/exceptions.html" \l "KeyError" \o "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')

PosixPath('docs/\_static')

PosixPath('docs/Makefile')

The children are yielded in arbitrary order, and the special entries '.' and '..' are not included. If a file is removed from or added to the directory after creating the iterator, whether an path object for that file be included is unspecified.

Path.**lchmod**(mode)

Like [Path.chmod()](https://docs.python.org/3/library/pathlib.html" \l "pathlib.Path.chmod" \o "pathlib.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()](https://docs.python.org/3/library/pathlib.html" \l "pathlib.Path.stat" \o "pathlib.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](https://docs.python.org/3/library/exceptions.html" \l "FileExistsError" \o "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](https://docs.python.org/3/library/exceptions.html" \l "FileNotFoundError" \o "FileNotFoundError).

If exist\_ok is false (the default), [FileExistsError](https://docs.python.org/3/library/exceptions.html" \l "FileExistsError" \o "FileExistsError) is raised if the target directory already exists.

If exist\_ok is true, [FileExistsError](https://docs.python.org/3/library/exceptions.html" \l "FileExistsError" \o "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()](https://docs.python.org/3/library/functions.html#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](https://docs.python.org/3/library/exceptions.html" \l "KeyError" \o "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

**>>>** p.read\_text()

'Text file contents'

The file is opened and then closed. The optional parameters have the same meaning as in [open()](https://docs.python.org/3/library/functions.html#open).

*New in version 3.5.*

Path.**readlink**()

Return the path to which the symbolic link points (as returned by [os.readlink()](https://docs.python.org/3/library/os.html" \l "os.readlink" \o "os.readlink)):

>>>

**>>>** p = Path('mylink')

**>>>** p.symlink\_to('setup.py')

**>>>** p.readlink()

PosixPath('setup.py')

*New in version 3.9.*

Path.**rename**(target)

Rename this file or directory to the given target, and return a new Path instance pointing to 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)

PosixPath('bar')

**>>>** target.open().read()

'some text'

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, not the directory of the Path object.

*Changed in version 3.8:*Added return value, return the new Path instance.

Path.**replace**(target)

Rename this file or directory to the given target, and return a new Path instance pointing to target. If target points to an existing file or directory, it will be unconditionally replaced.

The target path may be absolute or relative. Relative paths are interpreted relative to the current working directory, not the directory of the Path object.

*Changed in version 3.8:*Added return value, return the new Path instance.

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](https://docs.python.org/3/library/exceptions.html" \l "FileNotFoundError" \o "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](https://docs.python.org/3/library/exceptions.html" \l "RuntimeError" \o "RuntimeError) is raised.

*New in version 3.6:*The strict argument (pre-3.6 behavior is strict).

Path.**rglob**(pattern)

This is like calling [Path.glob()](https://docs.python.org/3/library/pathlib.html" \l "pathlib.Path.glob" \o "pathlib.Path.glob) with “\*\*/” added in front of the given relative pattern:

>>>

**>>>** sorted(Path().rglob("\*.py"))

[PosixPath('build/lib/pathlib.py'),

PosixPath('docs/conf.py'),

PosixPath('pathlib.py'),

PosixPath('setup.py'),

PosixPath('test\_pathlib.py')]

Raises an [auditing event](https://docs.python.org/3/library/sys.html#auditing) pathlib.Path.rglob with arguments self, pattern.

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()](https://docs.python.org/3/library/os.path.html" \l "os.path.samefile" \o "os.path.samefile) and [os.path.samestat()](https://docs.python.org/3/library/os.path.html" \l "os.path.samestat" \o "os.path.samestat).

An [OSError](https://docs.python.org/3/library/exceptions.html" \l "OSError" \o "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()](https://docs.python.org/3/library/os.html" \l "os.symlink" \o "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](https://docs.python.org/3/library/exceptions.html" \l "FileExistsError" \o "FileExistsError) is raised.

Path.**unlink**(missing\_ok=False)

Remove this file or symbolic link. If the path points to a directory, use [Path.rmdir()](https://docs.python.org/3/library/pathlib.html" \l "pathlib.Path.rmdir" \o "pathlib.Path.rmdir) instead.

If missing\_ok is false (the default), [FileNotFoundError](https://docs.python.org/3/library/exceptions.html" \l "FileNotFoundError" \o "FileNotFoundError) is raised if the path does not exist.

If missing\_ok is true, [FileNotFoundError](https://docs.python.org/3/library/exceptions.html" \l "FileNotFoundError" \o "FileNotFoundError) exceptions will be ignored (same behavior as the POSIX rm -f command).

*Changed in version 3.8:*The missing\_ok parameter was added.

Path.**link\_to**(target)

Create a hard link pointing to a path named target.

*New in version 3.8.*

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')

18

**>>>** p.read\_text()

'Text file contents'

An existing file of the same name is overwritten. The optional parameters have the same meaning as in [open()](https://docs.python.org/3/library/functions.html#open).

*New in version 3.5.*

## **Correspondence to tools in the [os](https://docs.python.org/3/library/os.html" \l "module-os" \o "os: Miscellaneous operating system interfaces.) module**

Below is a table mapping various [os](https://docs.python.org/3/library/os.html" \l "module-os" \o "os: Miscellaneous operating system interfaces.) functions to their corresponding [PurePath](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePath" \o "pathlib.PurePath)/[Path](https://docs.python.org/3/library/pathlib.html#pathlib.Path) equivalent.

**Note**

Although [os.path.relpath()](https://docs.python.org/3/library/os.path.html" \l "os.path.relpath" \o "os.path.relpath) and [PurePath.relative\_to()](https://docs.python.org/3/library/pathlib.html" \l "pathlib.PurePath.relative_to" \o "pathlib.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()](https://docs.python.org/3/library/os.path.html#os.path.abspath) | [Path.resolve()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.resolve) |
| [os.chmod()](https://docs.python.org/3/library/os.html#os.chmod) | [Path.chmod()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.chmod) |
| [os.mkdir()](https://docs.python.org/3/library/os.html#os.mkdir) | [Path.mkdir()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.mkdir) |
| [os.makedirs()](https://docs.python.org/3/library/os.html#os.makedirs) | [Path.mkdir()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.mkdir) |
| [os.rename()](https://docs.python.org/3/library/os.html#os.rename) | [Path.rename()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.rename) |
| [os.replace()](https://docs.python.org/3/library/os.html#os.replace) | [Path.replace()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.replace) |
| [os.rmdir()](https://docs.python.org/3/library/os.html#os.rmdir) | [Path.rmdir()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.rmdir) |
| [os.remove()](https://docs.python.org/3/library/os.html#os.remove), [os.unlink()](https://docs.python.org/3/library/os.html" \l "os.unlink" \o "os.unlink) | [Path.unlink()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.unlink) |
| [os.getcwd()](https://docs.python.org/3/library/os.html#os.getcwd) | [Path.cwd()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.cwd) |
| [os.path.exists()](https://docs.python.org/3/library/os.path.html#os.path.exists) | [Path.exists()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.exists) |
| [os.path.expanduser()](https://docs.python.org/3/library/os.path.html#os.path.expanduser) | [Path.expanduser()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.expanduser) and [Path.home()](https://docs.python.org/3/library/pathlib.html" \l "pathlib.Path.home" \o "pathlib.Path.home) |
| [os.listdir()](https://docs.python.org/3/library/os.html#os.listdir) | [Path.iterdir()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.iterdir) |
| [os.path.isdir()](https://docs.python.org/3/library/os.path.html#os.path.isdir) | [Path.is\_dir()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.is_dir) |
| [os.path.isfile()](https://docs.python.org/3/library/os.path.html#os.path.isfile) | [Path.is\_file()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.is_file) |
| [os.path.islink()](https://docs.python.org/3/library/os.path.html#os.path.islink) | [Path.is\_symlink()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.is_symlink) |
| [os.link()](https://docs.python.org/3/library/os.html#os.link) | [Path.link\_to()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.link_to) |
| [os.symlink()](https://docs.python.org/3/library/os.html#os.symlink) | [Path.symlink\_to()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.symlink_to) |
| [os.readlink()](https://docs.python.org/3/library/os.html#os.readlink) | [Path.readlink()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.readlink) |
| [os.stat()](https://docs.python.org/3/library/os.html#os.stat) | [Path.stat()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.stat), [Path.owner()](https://docs.python.org/3/library/pathlib.html" \l "pathlib.Path.owner" \o "pathlib.Path.owner), [Path.group()](https://docs.python.org/3/library/pathlib.html" \l "pathlib.Path.group" \o "pathlib.Path.group) |
| [os.path.isabs()](https://docs.python.org/3/library/os.path.html#os.path.isabs) | [PurePath.is\_absolute()](https://docs.python.org/3/library/pathlib.html#pathlib.PurePath.is_absolute) |
| [os.path.join()](https://docs.python.org/3/library/os.path.html#os.path.join) | [PurePath.joinpath()](https://docs.python.org/3/library/pathlib.html#pathlib.PurePath.joinpath) |
| [os.path.basename()](https://docs.python.org/3/library/os.path.html#os.path.basename) | [PurePath.name](https://docs.python.org/3/library/pathlib.html#pathlib.PurePath.name) |
| [os.path.dirname()](https://docs.python.org/3/library/os.path.html#os.path.dirname) | [PurePath.parent](https://docs.python.org/3/library/pathlib.html#pathlib.PurePath.parent) |
| [os.path.samefile()](https://docs.python.org/3/library/os.path.html#os.path.samefile) | [Path.samefile()](https://docs.python.org/3/library/pathlib.html#pathlib.Path.samefile) |
| [os.path.splitext()](https://docs.python.org/3/library/os.path.html#os.path.splitext) | [PurePath.suffix](https://docs.python.org/3/library/pathlib.html#pathlib.PurePath.suffix) |