operator.**__matmul__**(*a*, *b*)

Return `a @ b`.

New in version 3.5.

operator.**neg**(*obj*)

operator.**__neg__**(*obj*)

Return *obj* negated (`-obj`).

operator.**or_**(*a*, *b*)

operator.**__or__**(*a*, *b*)

Return the bitwise or of *a* and *b*.

operator.**pos**(*obj*)

operator.**__pos__**(*obj*)

Return *obj* positive (`+obj`).

operator.**pow**(*a*, *b*)

operator.**__pow__**(*a*, *b*)

Return `a ** b`, for *a* and *b* numbers.

operator.**rshift**(*a*, *b*)

operator.**__rshift__**(*a*, *b*)

Return *a* shifted right by *b*.

operator.**sub**(*a*, *b*)

operator.**__sub__**(*a*, *b*)

Return `a - b`.

operator.**truediv**(*a*, *b*)

operator.**__truediv__**(*a*, *b*)

Return `a / b` where 2/3 is .66 rather than 0. This is also known as "true" division.

operator.**xor**(*a*, *b*)

operator.**__xor__**(*a*, *b*)

Return the bitwise exclusive or of *a* and *b*.

Operations which work with sequences (some of them with mappings too) include:

operator.**concat**(*a*, *b*)

operator.**__concat__**(*a*, *b*)

Return `a + b` for *a* and *b* sequences.

operator.**contains**(*a*, *b*)

operator.**__contains__**(*a*, *b*)

Return the outcome of the test `b in a`. Note the reversed operands.

operator.**countOf**(*a*, *b*)

Return the number of occurrences of *b* in *a*.

operator.**delitem**(*a*, *b*)

operator.**__delitem__**(*a*, *b*)

Remove the value of *a* at index *b*.

operator.**getitem**(*a*, *b*)

operator.**__getitem__**(*a*, *b*)

Return the value of *a* at index *b*.

operator.**indexOf**(*a*, *b*)

Return the index of the first of occurrence of *b* in *a*.

operator.**setitem**(*a*, *b*, *c*)

operator.**__setitem__**(*a*, *b*, *c*)

Set the value of *a* at index *b* to *c*.

operator.**length_hint**(*obj*, *default=0*)

Return an estimated length for the object *obj*. First try to return its actual length, then an estimate using `object.__length_hint__()`, and finally return the default value.

New in version 3.4.

The following operation works with callables:

operator.**call**(*obj*, */*, **args*, ***kwargs*)

operator.**__call__**(*obj*, */*, **args*, ***kwargs*)

Return `obj(*args, **kwargs)`.

New in version 3.11.

The *operator* module also defines tools for generalized attribute and item lookups. These are useful for making fast field extractors as arguments for *map()*, *sorted()*, *itertools.groupby()*, or other functions that expect a function argument.

operator.**attrgetter**(*attr*)

operator.**attrgetter**(**attrs*)

Return a callable object that fetches *attr* from its operand. If more than one attribute is requested, returns a tuple of attributes. The attribute names can also contain dots. For example:

- After `f = attrgetter('name')`, the call `f(b)` returns `b.name`.
- After `f = attrgetter('name', 'date')`, the call `f(b)` returns `(b.name, b.date)`.
- After `f = attrgetter('name.first', 'name.last')`, the call `f(b)` returns `(b.name.first, b.name.last)`.

Equivalent to:

```
def attrgetter(*items):
    if any(not isinstance(item, str) for item in items):
        raise TypeError('attribute name must be a string')
    if len(items) == 1:
        attr = items[0]
        def g(obj):
            return resolve_attr(obj, attr)
    else:
        def g(obj):
            return tuple(resolve_attr(obj, attr) for attr in items)
    return g

def resolve_attr(obj, attr):
    for name in attr.split("."):
        obj = getattr(obj, name)
    return obj
```

operator.**itemgetter**(*item*)

operator.**itemgetter**(**items*)

Return a callable object that fetches *item* from its operand using the operand's *__getitem__()* method. If multiple items are specified, returns a tuple of lookup values. For example:

- After `f = itemgetter(2)`, the call `f(r)` returns `r[2]`.
- After `g = itemgetter(2, 5, 3)`, the call `g(r)` returns `(r[2], r[5], r[3])`.

Equivalent to:

```
def itemgetter(*items):
    if len(items) == 1:
        item = items[0]
        def g(obj):
            return obj[item]
    else:
        def g(obj):
            return tuple(obj[item] for item in items)
    return g
```

The items can be any type accepted by the operand's *__getitem__()* method. Dictionaries accept any *hashable* value. Lists, tuples, and strings accept an index or a slice:

```
>>> itemgetter(1)('ABCDEFG')
'B'
>>> itemgetter(1, 3, 5)('ABCDEFG')
('B', 'D', 'F')
>>> itemgetter(slice(2, None))('ABCDEFG')
'CDEFG'
>>> soldier = dict(rank='captain', name='dotterbart')
>>> itemgetter('rank')(soldier)
'captain'
```

Example of using *itemgetter()* to retrieve specific fields from a tuple record:

```
>>> inventory = [('apple', 3), ('banana', 2), ('pear', 5), ('orange', 1)]
>>> getcount = itemgetter(1)
>>> list(map(getcount, inventory))
[3, 2, 5, 1]
>>> sorted(inventory, key=getcount)
[('orange', 1), ('banana', 2), ('apple', 3), ('pear', 5)]
```

operator.**methodcaller**(*name*, */*, **args*, ***kwargs*)

Return a callable object that calls the method *name* on its operand. If additional arguments and/or keyword arguments are given, they will be given to the method as well. For example:

- After `f = methodcaller('name')`, the call `f(b)` returns `b.name()`.
- After `f = methodcaller('name', 'foo', bar=1)`, the call `f(b)` returns `b.name('foo', bar=1)`.

Equivalent to:

```
def methodcaller(name, /, *args, **kwargs):
    def caller(obj):
        return getattr(obj, name)(*args, **kwargs)
    return caller
```

### 10.3.1 Mapping Operators to Functions

This table shows how abstract operations correspond to operator symbols in the Python syntax and the functions in the *operator* module.

| Operation | Syntax | Function |
|---|---|---|
| Addition | `a + b` | `add(a, b)` |
| Concatenation | `seq1 + seq2` | `concat(seq1, seq2)` |
| Containment Test | `obj in seq` | `contains(seq, obj)` |
| Division | `a / b` | `truediv(a, b)` |

continues on next page

Table 1 – continued from previous page

| Operation | Syntax | Function |
|---|---|---|
| Division | `a // b` | `floordiv(a, b)` |
| Bitwise And | `a & b` | `and_(a, b)` |
| Bitwise Exclusive Or | `a ^ b` | `xor(a, b)` |
| Bitwise Inversion | `~ a` | `invert(a)` |
| Bitwise Or | `a \| b` | `or_(a, b)` |
| Exponentiation | `a ** b` | `pow(a, b)` |
| Identity | `a is b` | `is_(a, b)` |
| Identity | `a is not b` | `is_not(a, b)` |
| Indexed Assignment | `obj[k] = v` | `setitem(obj, k, v)` |
| Indexed Deletion | `del obj[k]` | `delitem(obj, k)` |
| Indexing | `obj[k]` | `getitem(obj, k)` |
| Left Shift | `a << b` | `lshift(a, b)` |
| Modulo | `a % b` | `mod(a, b)` |
| Multiplication | `a * b` | `mul(a, b)` |
| Matrix Multiplication | `a @ b` | `matmul(a, b)` |
| Negation (Arithmetic) | `- a` | `neg(a)` |
| Negation (Logical) | `not a` | `not_(a)` |
| Positive | `+ a` | `pos(a)` |
| Right Shift | `a >> b` | `rshift(a, b)` |
| Slice Assignment | `seq[i:j] = values` | `setitem(seq, slice(i, j), values)` |
| Slice Deletion | `del seq[i:j]` | `delitem(seq, slice(i, j))` |
| Slicing | `seq[i:j]` | `getitem(seq, slice(i, j))` |
| String Formatting | `s % obj` | `mod(s, obj)` |
| Subtraction | `a - b` | `sub(a, b)` |
| Truth Test | `obj` | `truth(obj)` |
| Ordering | `a < b` | `lt(a, b)` |
| Ordering | `a <= b` | `le(a, b)` |
| Equality | `a == b` | `eq(a, b)` |
| Difference | `a != b` | `ne(a, b)` |
| Ordering | `a >= b` | `ge(a, b)` |
| Ordering | `a > b` | `gt(a, b)` |

### 10.3.2 In-place Operators

Many operations have an "in-place" version. Listed below are functions providing a more primitive access to in-place operators than the usual syntax does; for example, the *statement* `x += y` is equivalent to `x = operator.iadd(x, y)`. Another way to put it is to say that `z = operator.iadd(x, y)` is equivalent to the compound statement `z = x; z += y`.

In those examples, note that when an in-place method is called, the computation and assignment are performed in two separate steps. The in-place functions listed below only do the first step, calling the in-place method. The second step, assignment, is not handled.

For immutable targets such as strings, numbers, and tuples, the updated value is computed, but not assigned back to the input variable:

```
>>> a = 'hello'
>>> iadd(a, ' world')
'hello world'
>>> a
'hello'
```

For mutable targets such as lists and dictionaries, the in-place method will perform the update, so no subsequent assignment is necessary:

```
>>> s = ['h', 'e', 'l', 'l', 'o']
>>> iadd(s, [' ', 'w', 'o', 'r', 'l', 'd'])
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> s
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

operator.**iadd**(*a*, *b*)

operator.**__iadd__**(*a*, *b*)

`a = iadd(a, b)` is equivalent to `a += b`.

operator.**iand**(*a*, *b*)

operator.**__iand__**(*a*, *b*)

`a = iand(a, b)` is equivalent to `a &= b`.

operator.**iconcat**(*a*, *b*)

operator.**__iconcat__**(*a*, *b*)

`a = iconcat(a, b)` is equivalent to `a += b` for *a* and *b* sequences.

operator.**ifloordiv**(*a*, *b*)

operator.**__ifloordiv__**(*a*, *b*)

`a = ifloordiv(a, b)` is equivalent to `a //= b`.

operator.**ilshift**(*a*, *b*)

operator.**__ilshift__**(*a*, *b*)

`a = ilshift(a, b)` is equivalent to `a <<= b`.

operator.**imod**(*a*, *b*)

operator.**__imod__**(*a*, *b*)

`a = imod(a, b)` is equivalent to `a %= b`.

operator.**imul**(*a*, *b*)

operator.**__imul__**(*a*, *b*)

`a = imul(a, b)` is equivalent to `a *= b`.

operator.**imatmul**(*a*, *b*)

operator.**__imatmul__**(*a*, *b*)

`a = imatmul(a, b)` is equivalent to `a @= b`.

New in version 3.5.

operator.**ior**(*a*, *b*)

operator.**__ior__**(*a*, *b*)

`a = ior(a, b)` is equivalent to `a |= b`.

operator.**ipow**(*a*, *b*)

operator.**__ipow__**(*a*, *b*)

`a = ipow(a, b)` is equivalent to `a **= b`.

operator.**irshift**(*a*, *b*)

operator.**__irshift__**(*a*, *b*)

`a = irshift(a, b)` is equivalent to `a >>= b`.

operator.**isub**(*a*, *b*)

operator.**__isub__**(*a*, *b*)

`a = isub(a, b)` is equivalent to `a -= b`.

operator.**itruediv**(*a*, *b*)

operator.**__itruediv__**(*a*, *b*)

a = itruediv(a, b) is equivalent to a /= b.

operator.**ixor**(*a*, *b*)

operator.**__ixor__**(*a*, *b*)

a = ixor(a, b) is equivalent to a ^= b.

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:

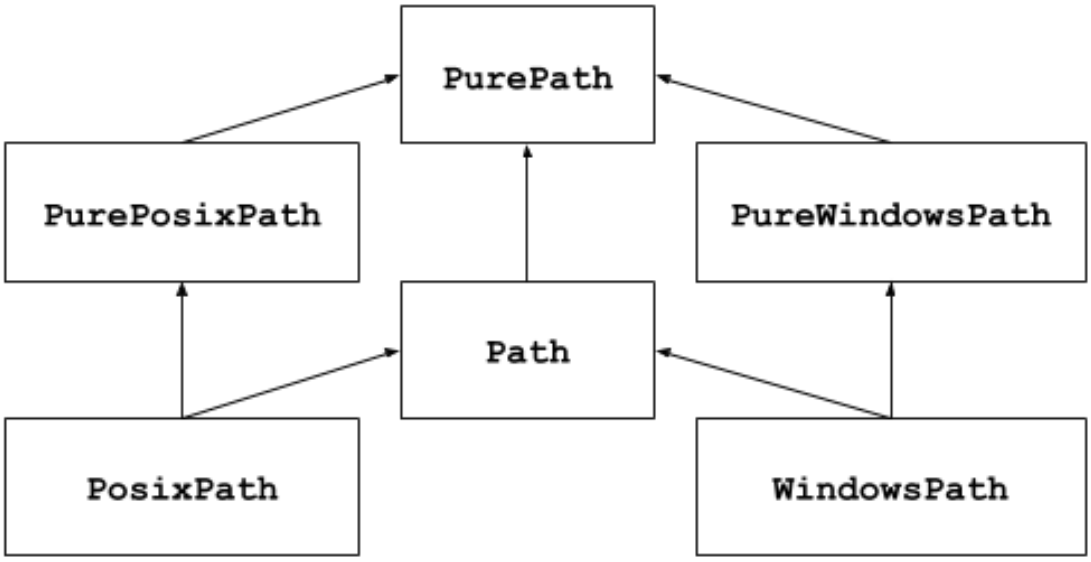
## 11.1 `pathlib` — Object-oriented filesystem paths

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('.')
```

If a segment is an absolute path, all previous segments are ignored (like *os.path.join()*):

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

On Windows, the drive is not reset when a rooted relative path segment (e.g., `r'\foo'`) is encountered:

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

Spurious slashes and single dots are collapsed, but double dots (`'..'`) and leading double slashes (`'//'`) are not, since this would change the meaning of a path for various reasons (e.g. symbolic links, UNC paths):

```
>>> PurePath('foo//bar')
PurePosixPath('foo/bar')
>>> 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, including UNC paths:

```
>>> PureWindowsPath('c:/Program Files/')
PureWindowsPath('c:/Program Files')
>>> PureWindowsPath('//server/share/file')
PureWindowsPath('//server/share/file')
```

*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, like *os.path.join()*. If the argument is an absolute path, the previous path is ignored. On Windows, the drive is not reset when the argument is a rooted relative path (e.g., `r'\foo'`):

```
>>> p = PurePath('/etc')
>>> p
PurePosixPath('/etc')
>>> p / 'init.d' / 'apache2'
PurePosixPath('/etc/init.d/apache2')
>>> q = PurePath('bin')
>>> '/usr' / q
PurePosixPath('/usr/bin')
>>> p / '/an_absolute_path'
PurePosixPath('/an_absolute_path')
>>> PureWindowsPath('c:/Windows', '/Program Files')
PureWindowsPath('c:/Program Files')
```

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')
>>> 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
'\\'
```

If the path starts with more than two successive slashes, *PurePosixPath* collapses them:

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

**Note:** This behavior conforms to *The Open Group Base Specifications Issue 6*, paragraph 4.11 Pathname Resolution:

*"A pathname that begins with two successive slashes may be interpreted in an implementation-defined manner, although more than two leading slashes shall be treated as a single slash."*

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:/')
```

Changed in version 3.10: The parents sequence now supports *slices* and negative index values.

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_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*, 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')
>>> 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* 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* 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* 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* 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).

Changed in version 3.8: *exists()*, *is_dir()*, *is_file()*, *is_mount()*, *is_symlink()*, *is_block_device()*, *is_char_device()*, *is_fifo()*, *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()*):

```
>>> 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). If the home directory can't be resolved, *RuntimeError* is raised.

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

New in version 3.5.

Path.**stat**(*, *follow_symlinks=True*)

Return a *os.stat_result* object containing information about this path, like *os.stat()*. The result is looked up at each call to this method.

This method normally follows symlinks; to stat a symlink add the argument `follow_symlinks=False`, or use *lstat()*.

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

Changed in version 3.10: The *follow_symlinks* parameter was added.

Path.**chmod**(*mode*, *, *follow_symlinks=True*)

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

This method normally follows symlinks. Some Unix flavours support changing permissions on the symlink itself; on these platforms you may add the argument `follow_symlinks=False`, or use *lchmod()*.

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

Changed in version 3.10: The *follow_symlinks* parameter was added.

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()*. If a home directory can't be resolved, *RuntimeError* is raised.

```
>>> 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')]
```

Patterns are the same as for *fnmatch*, with the addition of "**" which 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* `pathlib.Path.glob` with arguments `self`, `pattern`.

Changed in version 3.11: Return only directories if *pattern* ends with a pathname components separator (*sep* or *altsep*).

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')
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 a path object for that file be included is unspecified.

`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
>>> 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.**readlink**()

Return the path to which the symbolic link points (as returned by *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. On Windows, if *target* exists, *FileExistsError* will be raised. *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.

It is implemented in terms of *os.rename()* and gives the same guarantees.

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 empty 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.**absolute**()

Make the path absolute, without normalization or resolving symlinks. Returns a new path object:

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

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 (pre-3.6 behavior is strict).

Path.**rglob**(*pattern*)

This is like calling *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* `pathlib.Path.rglob` with arguments `self`, `pattern`.

Changed in version 3.11: Return only directories if *pattern* ends with a pathname components separator (*sep* or *altsep*).

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.**hardlink_to**(*target*)

Make this path a hard link to the same file as *target*.

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

New in version 3.10.

Path.**link_to**(*target*)

Make *target* a hard link to this path.

**Warning:** This function does not make this path a hard link to *target*, despite the implication of the function and argument names. The argument order (target, link) is the reverse of *Path.symlink_to()* and *Path.hardlink_to()*, but matches that of *os.link()*.

New in version 3.8.

Deprecated since version 3.10: This method is deprecated in favor of *Path.hardlink_to()*, as the argument order of *Path.link_to()* does not match that of *Path.symlink_to()*.

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**(*missing_ok=False*)

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

If *missing_ok* is false (the default), *FileNotFoundError* is raised if the path does not exist.

If *missing_ok* is true, *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.**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*, *newline=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()*.

New in version 3.5.

Changed in version 3.10: The *newline* parameter was added.

### 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:** Not all pairs of functions/methods below are equivalent. Some of them, despite having some overlapping use-cases, have different semantics. They include *os.path.abspath()* and *Path.absolute()*, *os.path.relpath()* and *PurePath.relative_to()*.

| os and os.path | pathlib |
|---|---|
| os.path.abspath() | Path.absolute()[1] |
| os.path.realpath() | Path.resolve() |
| os.chmod() | Path.chmod() |
| os.mkdir() | Path.mkdir() |
| os.makedirs() | 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.listdir() | Path.iterdir() |
| os.path.isdir() | Path.is_dir() |
| os.path.isfile() | Path.is_file() |
| os.path.islink() | Path.is_symlink() |
| os.link() | Path.hardlink_to() |
| os.symlink() | Path.symlink_to() |
| os.readlink() | Path.readlink() |
| os.path.relpath() | PurePath.relative_to()[2] |
| 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.stem and PurePath.suffix |

## 11.2 os.path — Common pathname manipulations

**Source code:** Lib/posixpath.py (for POSIX) and Lib/ntpath.py (for Windows).

---

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 strings, or bytes, or any object implementing the os.PathLike protocol.

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

[1] os.path.abspath() normalizes the resulting path, which may change its meaning in the presence of symlinks, while Path.absolute() does not.

[2] PurePath.relative_to() requires self to be the subpath of the argument, but os.path.relpath() does not.

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

---

Changed in version 3.8: *exists()*, *lexists()*, *isdir()*, *isfile()*, *islink()*, and *ismount()* now return `False` instead of raising an exception for paths that contain characters or bytes unrepresentable at the OS level.

`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* contain both absolute and relative pathnames, the *paths* are on the different drives 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'

>>> 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, `USERPROFILE` will be used if set, otherwise a combination of `HOMEPATH` and `HOMEDRIVE` will be used. An initial `~user` is handled by checking that the last directory component of the current user's home directory matches `USERNAME`, and replacing it if so.

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*.

Changed in version 3.8: No longer uses `HOME` on Windows.

`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. It is not able to reliably detect bind mounts on the same filesystem. 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 segments intelligently. The return value is the concatenation of *path* and all members of **paths*, with exactly one directory separator following each non-empty part, except the last. That is, the result will only end in a separator if the last part is either empty or ends in a separator. If a segment is an absolute path (which on Windows requires both a drive and a root), then all previous segments are ignored and joining continues from the absolute path segment.

On Windows, the drive is not reset when a rooted path segment (e.g., `r'\foo'`) is encountered. If a segment is on a different drive or is an absolute path, all previous segments are ignored and the drive 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 Windows, convert all characters in the pathname to lowercase, and also convert forward slashes to backward slashes. On other operating systems, return the path unchanged.

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()*.

---

**Note:**

On POSIX systems, in accordance with IEEE Std 1003.1 2013 Edition; 4.13 Pathname Resolution, if a pathname begins with exactly two slashes, the first component following the leading characters may be interpreted in an implementation-defined manner, although more than two leading characters shall be treated as a single character.

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

---

os.path.**realpath**(*path*, *, *strict=False*)

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

If a path doesn't exist or a symlink loop is encountered, and *strict* is `True`, `OSError` is raised. If *strict* is `False`, the path is resolved as far as possible and any remainder is appended without checking whether it exists.

---

**Note:** This function emulates the operating system's procedure for making a path canonical, which differs slightly between Windows and UNIX with respect to how links and subsequent path components interact.

Operating system APIs make paths canonical as needed, so it's not normally necessary to call this function.

---

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

Changed in version 3.8: Symbolic links and junctions are now resolved on Windows.

Changed in version 3.10: The *strict* parameter was added.

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*. On Windows, `ValueError` is raised when *path* and *start* are on different drives.

*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:

```
>>> splitdrive("c:/dir")
("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:

```
>>> splitdrive("//host/computer/dir")
("//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 the extension, *ext*, is empty or begins with a period and contains at most one period.

If the path contains no extension, *ext* will be `''`:

```
>>> splitext('bar')
('bar', '')
```

If the path contains an extension, then *ext* will be set to this extension, including the leading period. Note that previous periods will be ignored:

```
>>> splitext('foo.bar.exe')
('foo.bar', '.exe')
>>> splitext('/foo/bar.exe')
('/foo/bar', '.exe')
```

Leading periods of the last component of the path are considered to be part of the root:

```
>>> splitext('.cshrc')
('.cshrc', '')
>>> splitext('/foo/....jpg')
('/foo/....jpg', '')
```

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).