`!pathlib` --- Object-oriented filesystem paths

- Basic use
- Exceptions
- Pure paths
  - General properties
  - Operators
  - Accessing individual parts
  - Methods and properties
- Concrete paths
  - Parsing and generating URIs
  - Expanding and resolving paths
  - Querying file type and status
  - Reading and writing files
  - Reading directories
  - Creating files and directories
  - Copying, moving and deleting
  - Permissions and ownership
- Pattern language
- Comparison to the glob module
- Comparison to the os and os.path modules
  - Corresponding tools
- Protocols

pathlib

3.4

Source code: `Lib/pathlib/`

single: path; operations

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

Inheritance diagram showing the classes available in pathlib. The most basic class is PurePath, whic

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 <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` 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.

`428`: The pathlib module -- object-oriented filesystem paths.

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

Exceptions

UnsupportedOperation

An exception inheriting `NotImplementedError` that is raised when an unsupported operation is called on a path object.

3.13

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*:

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, or an object implementing the `os.PathLike` interface where the `~os.PathLike.__fspath__` method returns a string, such as 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 or consists only of empty strings, the current directory is assumed:

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

3.6 Added support for the `os.PathLike` interface.

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

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

pathsegments is specified similarly to `PurePath`.

A subclass of `PurePath`, this path flavour represents Windows filesystem paths, including UNC paths:

```
>>> PureWindowsPath('c:/', 'Users', 'Ximénez')
PureWindowsPath('c:/Users/Ximénez')
>>> 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

from pathlib import PurePath, PurePosixPath, PureWindowsPath

The `~PurePath.parent`, `~PurePath.name`, `~PurePath.stem`, and `~PurePath.suffix` properties describe different parts of a path. For example, given:

```
>>> p = PurePosixPath('/srv/holy/grail.tar.gz')
```

For p = PurePosixPath('/srv/holy/grail.tar.gz'), p.parent is PurePosixPath('/srv/holy'), p.name is 'grail.

The stem excludes only the final suffix: in this example, it includes `.tar`.

Pure paths provide the following methods and properties:

PurePath.parser

The implementation of the `os.path` module used for low-level path parsing and joining: either `!posixpath` or `!ntpath`.

3.13

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, `~pathlib.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:/')
```

3.10 The parents sequence now supports `slices <slice>` 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 last dot-separated portion of the final component, if any:

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

This is commonly called the file extension.

3.14

A single dot (`.`) is considered a valid suffix.

PurePath.suffixes

A list of the path's suffixes, often called file extensions:

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

3.14

A single dot ("`.`") is considered a valid suffix.

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'
```

3.14

A single dot ("`.`") is considered a valid suffix.

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.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

This method is string-based; it neither accesses the filesystem nor treats "`..`" segments specially. The following code is equivalent:

> >>> u = PurePath('/usr') >>> u == p or u in p.parents False

3.9

3.12 3.14

Passing additional arguments is deprecated; if supplied, they are joined with *other*.

PurePath.joinpath(*pathsegments)

Calling this method is equivalent to combining the path with each of the given *pathsegments* 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.full_match(pattern, *, case_sensitive=None)

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

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

Patterns ending in "`/**`", as in the example above, may surprise when passed to `full_match`. Even though "`**`" can match 0 segments, "`.../**`" requires a separator to match. Hence "`/a/**`" does not match path "`/a`". Since `PurePath` drops trailing slashes, this pattern also does not match "`/a/`".

```
>>> PurePath('/a').full_match('/a/**')
False
>>> PurePath('/a/').full_match('/a/**')
False
```

`pathlib-pattern-language` documentation.

As with other methods, case-sensitivity follows platform defaults:

```
>>> PurePosixPath('b.py').full_match('*.PY')
False
>>> PureWindowsPath('b.py').full_match('*.PY')
True
```

Set *case_sensitive* to `True` or `False` to override this behaviour.

3.13

PurePath.match(pattern, *, case_sensitive=None)

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

This method is similar to `~PurePath.full_match`, but empty patterns aren't allowed (`ValueError` is raised), the recursive wildcard "`**`" isn't supported (it acts like non-recursive "`*`"), and if a relative pattern is provided, then 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
```

3.12 The *pattern* parameter accepts a `path-like object`.

3.12 The *case_sensitive* parameter was added.

PurePath.relative_to(other, walk_up=False)

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 941, in relative_to
    raise ValueError(error_message.format(str(self), str(formatted)))
ValueError: '/etc/passwd' is not in the subpath of '/usr' OR one path is relative and the
other is absolute.
```

When *walk_up* is false (the default), the path must start with *other*. When the argument is true, `..` entries may be added to form the relative path. In all other cases, such as the paths referencing different drives, `ValueError` is raised.:

```
>>> p.relative_to('/usr', walk_up=True)
PurePosixPath('../etc/passwd')
>>> p.relative_to('foo', walk_up=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pathlib.py", line 941, in relative_to
    raise ValueError(error_message.format(str(self), str(formatted)))
ValueError: '/etc/passwd' is not on the same drive as 'foo' OR one path is relative and the
other is absolute.
```

> [!WARNING] This function is part of `PurePath` and works with strings. It does not check or access the underlying file structure. This can impact the *walk_up* option as it assumes that no symlinks are present in the path; call `~Path.resolve` first if necessary to resolve symlinks.

3.12 The *walk_up* parameter was added (old behavior is the same as `walk_up=False`).

3.12 3.14

Passing additional positional arguments is deprecated; if supplied, they are joined with *other*.

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
```

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')
```

3.14

A single dot ("`.`") is considered a valid suffix. In previous versions, `ValueError` is raised if a single dot is supplied.

PurePath.with_segments(*pathsegments)

Create a new path object of the same type by combining the given *pathsegments*. This method is called whenever a derivative path is created, such as from `parent` and `relative_to`. Subclasses may override this method to pass information to derivative paths, for example:

```
from pathlib import PurePosixPath

class MyPath(PurePosixPath):
    def __init__(self, *pathsegments, session_id):
        super().__init__(*pathsegments)
        self.session_id = session_id

    def with_segments(self, *pathsegments):
        return type(self)(*pathsegments, session_id=self.session_id)

etc = MyPath('/etc', session_id=42)
hosts = etc / 'hosts'
print(hosts.session_id)  # 42
```

3.12

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:

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`.

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

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

pathsegments is specified similarly to `PurePath`.

3.13 Raises `UnsupportedOperation` on Windows. In previous versions, `NotImplementedError` was raised instead.

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

```
>>> WindowsPath('c:/', 'Users', 'Ximénez')
WindowsPath('c:/Users/Ximénez')
```

pathsegments is specified similarly to `PurePath`.

3.13 Raises `UnsupportedOperation` on non-Windows platforms. In previous versions, `NotImplementedError` was raised instead.

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__,))
UnsupportedOperation: cannot instantiate 'WindowsPath' on your system
```

Some concrete path methods can raise an `OSError` if a system call fails (for example because the path doesn't exist).

Parsing and generating URIs

Concrete path objects can be created from, and represented as, 'file' URIs conforming to `8089`.

> [!NOTE] File URIs are not portable across machines with different `filesystem encodings <filesystem-encoding>`.

Path.from_uri(uri)

Return a new path object from parsing a 'file' URI. For example:

```
>>> p = Path.from_uri('file:///etc/hosts')
PosixPath('/etc/hosts')
```

On Windows, DOS device and UNC paths may be parsed from URIs:

```
>>> p = Path.from_uri('file:///c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file://server/share')
WindowsPath('//server/share')
```

Several variant forms are supported:

```
>>> p = Path.from_uri('file:////server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file://///server/share')
WindowsPath('//server/share')
>>> p = Path.from_uri('file:c:/windows')
WindowsPath('c:/windows')
>>> p = Path.from_uri('file:/c|/windows')
WindowsPath('c:/windows')
```

`ValueError` is raised if the URI does not start with `file:`, or the parsed path isn't absolute.

3.13

3.14 The URL authority is discarded if it matches the local hostname. Otherwise, if the authority isn't empty or `localhost`, then on Windows a UNC path is returned (as before), and on other platforms a `ValueError` is raised.

Path.as_uri()

Represent the path as a 'file' URI. `ValueError` is raised if the path isn't absolute.

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

3.14 3.19

Calling this method from `PurePath` rather than `Path` is possible but deprecated. The method's use of `os.fsencode` makes it strictly impure.

Expanding and resolving paths

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')
```

3.5

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')
```

3.5

Path.cwd()

Return a new path object representing the current directory (as returned by `os.getcwd`):

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

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 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.

3.6 The *strict* parameter was added (pre-3.6 behavior is strict).

3.13 Symlink loops are treated like other errors: `OSError` is raised in strict mode, and no exception is raised in non-strict mode. In previous versions, `RuntimeError` is raised no matter the value of *strict*.

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')
```

3.9

3.13 Raises `UnsupportedOperation` if `os.readlink` is not available. In previous versions, `NotImplementedError` was raised.

Querying file type and status

3.8

`~Path.exists`, `~Path.is_dir`, `~Path.is_file`, `~Path.is_mount`, `~Path.is_symlink`, `~Path.is_block_device`, `~Path.is_char_device`, `~Path.is_fifo`, `~Path.is_socket` now return `False` instead of raising an exception for paths that contain characters unrepresentable at the OS level.

3.14

The methods given above now return `False` instead of raising any `OSError` exception from the operating system. In previous versions, some kinds of `OSError` exception are raised, and others suppressed. The new behaviour is consistent with `os.path.exists`, `os.path.isdir`, etc. Use `~Path.stat` to retrieve the file status without suppressing exceptions.

Path.stat(*, follow_symlinks=True)

Return an `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 `~Path.lstat`.

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

3.10 The *follow_symlinks* parameter was added.

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.exists(*, follow_symlinks=True)

Return `True` if the path points to an existing file or directory and `False` if the path is invalid, inaccessible or missing. Use `Path.stat` to distinguish between these cases.

This method normally follows symlinks; to check if a symlink exists, add the argument `follow_symlinks=False`.

```
>>> Path('').exists()  # The current directory.
True
>>> Path('.').exists()
True
>>> Path('setup.py').exists()
True
>>> Path('/etc').exists()
True
>>> Path('nonexistentfile').exists()
False
```

3.12 The *follow_symlinks* parameter was added.

Path.is_file(*, follow_symlinks=True)

Return `True` if the path points to a regular file. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a regular file. Use `Path.stat` to distinguish between these cases.

This method normally follows symlinks; to exclude symlinks, add the argument `follow_symlinks=False`.

3.13 The *follow_symlinks* parameter was added.

Path.is_dir(*, follow_symlinks=True)

Return `True` if the path points to a directory. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a directory. Use `Path.stat` to distinguish between these cases.

This method normally follows symlinks; to exclude symlinks to directories, add the argument `follow_symlinks=False`.

3.13 The *follow_symlinks* parameter was added.

Path.is_symlink()

Return `True` if the path points to a symbolic link, even if that symlink is broken. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a symbolic link. Use `Path.stat` to distinguish between these cases.

Path.is_junction()

Return `True` if the path points to a junction, and `False` for any other type of file. Currently only Windows supports junctions.

3.12

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. On Windows, a mount point is considered to be a drive letter root (e.g. `c:\`), a UNC share (e.g. `\\server\share`), or a mounted filesystem directory.

3.7

3.12 Windows support was added.

Path.is_socket()

Return `True` if the path points to a Unix socket. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a Unix socket. Use `Path.stat` to distinguish between these cases.

Path.is_fifo()

Return `True` if the path points to a FIFO. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a FIFO. Use `Path.stat` to distinguish between these cases.

Path.is_block_device()

Return `True` if the path points to a block device. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a block device. Use `Path.stat` to distinguish between these cases.

Path.is_char_device()

Return `True` if the path points to a character device. `False` will be returned if the path is invalid, inaccessible or missing, or if it points to something other than a character device. Use `Path.stat` to distinguish between these cases.

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
```

3.5

Path.info

A `~pathlib.types.PathInfo` object that supports querying file type information. The object exposes methods that cache their results, which can help reduce the number of system calls needed when switching on file type. For example:

```
>>> p = Path('src')
>>> if p.info.is_symlink():
...     print('symlink')
... elif p.info.is_dir():
...     print('directory')
... elif p.info.exists():
...     print('something else')
... else:
...     print('not found')
...
```

```
directory
```

If the path was generated from `Path.iterdir` then this attribute is initialized with some information about the file type gleaned from scanning the parent directory. Merely accessing `Path.info` does not perform any filesystem queries.

To fetch up-to-date information, it's best to call `Path.is_dir`, `~Path.is_file` and `~Path.is_symlink` rather than methods of this attribute. There is no way to reset the cache; instead you can create a new path object with an empty info cache via `p = Path(p)`.

3.14

Reading and writing files

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.read_text(encoding=None, errors=None, newline=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`.

3.5

3.13 The *newline* parameter was added.

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'
```

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'
```

Return the number of characters written.

An existing file of the same name is overwritten. The optional parameters have the same meaning as in `open`.

3.5

3.10 The *newline* 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'
```

Return the number of bytes written.

An existing file of the same name is overwritten.

Reading directories

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, it is unspecified whether a path object for that file is included.

If the path is not a directory or otherwise inaccessible, `OSError` is raised.

Path.glob(pattern, *, case_sensitive=None, recurse_symlinks=False)

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

> [!NOTE] The paths are returned in no particular order. If you need a specific order, sort the results.

`pathlib-pattern-language` documentation.

By default, or when the *case_sensitive* keyword-only argument is set to `None`, this method matches paths using platform-specific casing rules: typically, case-sensitive on POSIX, and case-insensitive on Windows. Set *case_sensitive* to `True` or `False` to override this behaviour.

By default, or when the *recurse_symlinks* keyword-only argument is set to `False`, this method follows symlinks except when expanding "`**`" wildcards. Set *recurse_symlinks* to `True` to always follow symlinks.

> [!NOTE] Any `OSError` exceptions raised from scanning the filesystem are suppressed. This includes `PermissionError` when accessing directories without read permission.

pathlib.Path.glob self,pattern pathlib.Path.glob

3.12 The *case_sensitive* parameter was added.

3.13 The *recurse_symlinks* parameter was added.

3.13 The *pattern* parameter accepts a `path-like object`.

3.13 Any `OSError` exceptions raised from scanning the filesystem are suppressed. In previous versions, such exceptions are suppressed in many cases, but not all.

Path.rglob(pattern, *, case_sensitive=None, recurse_symlinks=False)

Glob the given relative *pattern* recursively. This is like calling `Path.glob` with "`**/`" added in front of the *pattern*.

> [!NOTE] The paths are returned in no particular order. If you need a specific order, sort the results.

> [!NOTE] Any `OSError` exceptions raised from scanning the filesystem are suppressed. This includes `PermissionError` when accessing directories without

> read permission.

`pathlib-pattern-language` and `Path.glob` documentation.

pathlib.Path.rglob self,pattern pathlib.Path.rglob

3.12 The *case_sensitive* parameter was added.

3.13 The *recurse_symlinks* parameter was added.

3.13 The *pattern* parameter accepts a `path-like object`.

Path.walk(top_down=True, on_error=None, follow_symlinks=False)

Generate the file names in a directory tree by walking the tree either top-down or bottom-up.

For each directory in the directory tree rooted at *self* (including *self* but excluding '.' and '..'), the method yields a 3-tuple of `(dirpath, dirnames, filenames)`.

dirpath is a `Path` to the directory currently being walked, *dirnames* is a list of strings for the names of subdirectories in *dirpath* (excluding `'.'` and `'..'`), and *filenames* is a list of strings for the names of the non-directory files in *dirpath*. To get a full path (which begins with *self*) to a file or directory in *dirpath*, do `dirpath / name`. Whether or not the lists are sorted is file system-dependent.

If the optional argument *top_down* is true (which is the default), the triple for a directory is generated before the triples for any of its subdirectories (directories are walked top-down). If *top_down* is false, the triple for a directory is generated after the triples for all of its subdirectories (directories are walked bottom-up). No matter the value of *top_down*, the list of subdirectories is retrieved before the triples for the directory and its subdirectories are walked.

When *top_down* is true, the caller can modify the *dirnames* list in-place (for example, using `del` or slice assignment), and `Path.walk` will only recurse into the subdirectories whose names remain in *dirnames*. This can be used to prune the search, or to impose a specific order of visiting, or even to inform `Path.walk` about directories the caller creates or renames before it resumes `Path.walk` again. Modifying *dirnames* when *top_down* is false has no effect on the behavior of `Path.walk` since the directories in *dirnames* have already been generated by the time *dirnames* is yielded to the caller.

By default, errors from `os.scandir` are ignored. If the optional argument *on_error* is specified, it should be a callable; it will be called with one argument, an `OSError` instance. The callable can handle the error to continue the walk or re-raise it to stop the walk. Note that the filename is available as the `filename` attribute of the exception object.

By default, `Path.walk` does not follow symbolic links, and instead adds them to the *filenames* list. Set *follow_symlinks* to true to resolve symlinks and place them in *dirnames* and *filenames* as appropriate for their targets, and consequently visit directories pointed to by symlinks (where supported).

> [!NOTE] Be aware that setting *follow_symlinks* to true can lead to infinite recursion if a link points to a parent directory of itself. `Path.walk` does not keep track of the directories it has already visited.

> [!NOTE] `Path.walk` assumes the directories it walks are not modified during execution. For example, if a directory from *dirnames* has been replaced with a symlink and *follow_symlinks* is false, `Path.walk` will still try to descend into it. To prevent such behavior, remove directories from *dirnames* as appropriate.

> [!NOTE] Unlike `os.walk`, `Path.walk` lists symlinks to directories in *filenames* if *follow_symlinks* is false.

This example displays the number of bytes used by all files in each directory, while ignoring `__pycache__` directories:

```
from pathlib import Path
for root, dirs, files in Path("cpython/Lib/concurrent").walk(on_error=print):
  print(
      root,
      "consumes",
      sum((root / file).stat().st_size for file in files),
      "bytes in",
      len(files),
      "non-directory files"
  )
  if '__pycache__' in dirs:
        dirs.remove('__pycache__')
```

This next example is a simple implementation of `shutil.rmtree`. Walking the tree bottom-up is essential as `rmdir` doesn't allow deleting a directory before it is empty:

```
# Delete everything reachable from the directory "top".
# CAUTION:  This is dangerous! For example, if top == Path('/'),
# it could delete all of your files.
for root, dirs, files in top.walk(top_down=False):
    for name in files:
        (root / name).unlink()
    for name in dirs:
        (root / name).rmdir()
```

3.12

Creating files and directories

Path.touch(mode=0o666, exist_ok=True)

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

The `~Path.open`, `~Path.write_text` and `~Path.write_bytes` methods are often used to create files.

Path.mkdir(mode=0o777, parents=False, exist_ok=False, *, parent_mode=None)

Create a new directory at this given path. If *mode* is given, it is combined with the process's `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 *parent_mode* is not `None`, it is used as the mode for any newly-created, intermediate-level directories when *parents* is true. Like *mode*, it is combined with the process's `umask` value. Otherwise, intermediate directories are created with the default permissions (also subject to the umask).

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` will not be raised unless the given path already exists in the file system and is not a directory (same behavior as the POSIX `mkdir -p` command).

3.5 The *exist_ok* parameter was added.

3.15 The *parent_mode* parameter.

Path.symlink_to(target, target_is_directory=False)

Make this path a symbolic link pointing to *target*.

On Windows, a symlink represents either a file or a directory, and does not morph to the target dynamically. If the target is present, the type of the symlink will be created to match. Otherwise, the symlink will be created as a directory if *target_is_directory* is true or a file symlink (the default) otherwise. On non-Windows platforms, *target_is_directory* 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.

3.13 Raises `UnsupportedOperation` if `os.symlink` is not available. In previous versions, `NotImplementedError` was raised.

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.

3.10

3.13 Raises `UnsupportedOperation` if `os.link` is not available. In previous versions, `NotImplementedError` was raised.

Copying, moving and deleting

Path.copy(target, *, follow_symlinks=True, preserve_metadata=False)

Copy this file or directory tree to the given *target*, and return a new `!Path` instance pointing to *target*.

If the source is a file, the target will be replaced if it is an existing file. If the source is a symlink and *follow_symlinks* is true (the default), the symlink's target is copied. Otherwise, the symlink is recreated at the destination.

If *preserve_metadata* is false (the default), only directory structures and file data are guaranteed to be copied. Set *preserve_metadata* to true to ensure that file and directory permissions, flags, last access and modification times, and extended attributes are copied where supported. This argument has no effect when copying files on Windows (where metadata is always preserved).

> [!NOTE] Where supported by the operating system and file system, this method performs a lightweight copy, where data blocks are only copied when modified. This is known as copy-on-write.

3.14

Path.copy_into(target_dir, *, follow_symlinks=True, preserve_metadata=False)

Copy this file or directory tree into the given *target_dir*, which should be an existing directory. Other arguments are handled identically to `Path.copy`. Returns a new `!Path` instance pointing to the copy.

3.14

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.

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.

3.8 Added return value, return the new `!Path` instance.

Path.move(target)

Move this file or directory tree to the given *target*, and return a new `!Path` instance pointing to *target*.

If the *target* doesn't exist it will be created. If both this path and the *target* are existing files, then the target is overwritten. If both paths point to the same file or directory, or the *target* is a non-empty directory, then `OSError` is raised.

If both paths are on the same filesystem, the move is performed with `os.replace`. Otherwise, this path is copied (preserving metadata and symlinks) and then deleted.

3.14

Path.move_into(target_dir)

Move this file or directory tree into the given *target_dir*, which should be an existing directory. Returns a new `!Path` instance pointing to the moved path.

3.14

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).

3.8 The *missing_ok* parameter was added.

Path.rmdir()

Remove this directory. The directory must be empty.

Permissions and ownership

Path.owner(*, follow_symlinks=True)

Return the name of the user owning the file. `KeyError` is raised if the file's user identifier (UID) isn't found in the system database.

This method normally follows symlinks; to get the owner of the symlink, add the argument `follow_symlinks=False`.

3.13 Raises `UnsupportedOperation` if the `pwd` module is not available. In earlier versions, `NotImplementedError` was raised.

3.13 The *follow_symlinks* parameter was added.

Path.group(*, follow_symlinks=True)

Return the name of the group owning the file. `KeyError` is raised if the file's group identifier (GID) isn't found in the system database.

This method normally follows symlinks; to get the group of the symlink, add the argument `follow_symlinks=False`.

3.13 Raises `UnsupportedOperation` if the `grp` module is not available. In earlier versions, `NotImplementedError` was raised.

3.13 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 `~Path.lchmod`.

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

3.10 The *follow_symlinks* parameter was added.

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.

Pattern language

The following wildcards are supported in patterns for `~PurePath.full_match`, `~Path.glob` and `~Path.rglob`:

`**` (entire segment)
Matches any number of file or directory segments, including zero. Note that in `~PurePath.full_match`, "`a/**`" does not match "`a`", whereas `~Path.glob` yields "`a`" for that pattern.

`*` (entire segment)
Matches one file or directory segment.

`*` (part of a segment)
Matches any number of non-separator characters, including zero.

`?`
Matches one non-separator character.

`[seq]`
Matches one character in *seq*, where *seq* is a sequence of characters. Range expressions are supported; for example, `[a-z]` matches any lowercase ASCII letter. Multiple ranges can be combined: `[a-zA-Z0-9_]` matches any ASCII letter, digit, or underscore.

`[!seq]`
Matches one character not in *seq*, where *seq* follows the same rules as above.

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

The "`**`" wildcard enables recursive globbing. A few examples:

Pattern	Meaning
"`**/*`"	Any path with at least one segment.
"`**/*.py`"	Any path with a final segment ending "`.py`".
"`assets/**`"	Any path starting with "`assets/`".
"`assets/**/*`"	Any path starting with "`assets/`", excluding "`assets/`" itself.

> [!NOTE] Globbing with the "`**`" wildcard visits every directory in the tree. Large directory trees may take a long time to search.

3.13 Globbing with a pattern that ends with "`**`" returns both files and directories. In previous versions, only directories were returned.

In `Path.glob` and `~Path.rglob`, a trailing slash may be added to the pattern to match only directories.

3.11 Globbing with a pattern that ends with a pathname components separator (`~os.sep` or `~os.altsep`) returns only directories.

Comparison to the `glob` module

The patterns accepted and results generated by `Path.glob` and `Path.rglob` differ slightly from those by the `glob` module:

1. Files beginning with a dot are not special in pathlib. This is like passing `include_hidden=True` to `glob.glob`.
2. "`**`" pattern components are always recursive in pathlib. This is like passing `recursive=True` to `glob.glob`.
3. "`**`" pattern components do not follow symlinks by default in pathlib. This behaviour has no equivalent in `glob.glob`, but you can pass `recurse_symlinks=True` to `Path.glob` for compatible behaviour.
4. Like all `PurePath` and `Path` objects, the values returned from `Path.glob` and `Path.rglob` don't include trailing slashes.

5. The values returned from pathlib's `path.glob()` and `path.rglob()` include the *path* as a prefix, unlike the results of `glob.glob(root_dir=path)`.
6. The values returned from pathlib's `path.glob()` and `path.rglob()` may include *path* itself, for example when globbing "`**`", whereas the results of `glob.glob(root_dir=path)` never include an empty string that would correspond to *path*.

Comparison to the `os` and `os.path` modules

pathlib implements path operations using `PurePath` and `Path` objects, and so it's said to be *object-oriented*. On the other hand, the `os` and `os.path` modules supply functions that work with low-level `str` and `bytes` objects, which is a more *procedural* approach. Some users consider the object-oriented style to be more readable.

Many functions in `os` and `os.path` support `bytes` paths and `paths relative to directory descriptors <dir_fd>`. These features aren't available in pathlib.

Python's `str` and `bytes` types, and portions of the `os` and `os.path` modules, are written in C and are very speedy. pathlib is written in pure Python and is often slower, but rarely slow enough to matter.

pathlib's path normalization is slightly more opinionated and consistent than `os.path`. For example, whereas `os.path.abspath` eliminates "`..`" segments from a path, which may change its meaning if symlinks are involved, `Path.absolute` preserves these segments for greater safety.

pathlib's path normalization may render it unsuitable for some applications:

1. pathlib normalizes `Path("my_folder/")` to `Path("my_folder")`, which changes a path's meaning when supplied to various operating system APIs and command-line utilities. Specifically, the absence of a trailing separator may allow the path to be resolved as either a file or directory, rather than a directory only.
2. pathlib normalizes `Path("./my_program")` to `Path("my_program")`, which changes a path's meaning when used as an executable search path, such as in a shell or when spawning a child process. Specifically, the absence of a separator in the path may force it to be looked up in `PATH` rather than the current directory.

As a consequence of these differences, pathlib is not a drop-in replacement for `os.path`.

Corresponding tools

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

`os` and `os.path`	`!pathlib`
`os.path.dirname`	`PurePath.parent`
`os.path.basename`	`PurePath.name`
`os.path.splitext`	`PurePath.stem`, `PurePath.suffix`
`os.path.join`	`PurePath.joinpath`
`os.path.isabs`	`PurePath.is_absolute`
`os.path.relpath`	`PurePath.relative_to`[^1]
`os.path.expanduser`	`Path.expanduser`[^2]
`os.path.realpath`	`Path.resolve`
`os.path.abspath`	`Path.absolute`[^3]
`os.path.exists`	`Path.exists`
`os.path.isfile`	`Path.is_file`
`os.path.isdir`	`Path.is_dir`
`os.path.islink`	`Path.is_symlink`
`os.path.isjunction`	`Path.is_junction`
`os.path.ismount`	`Path.is_mount`
`os.path.samefile`	`Path.samefile`
`os.getcwd`	`Path.cwd`
`os.stat`	`Path.stat`
`os.lstat`	`Path.lstat`
`os.listdir`	`Path.iterdir`
`os.walk`	`Path.walk`[^4]
`os.mkdir`, `os.makedirs`	`Path.mkdir`
`os.link`	`Path.hardlink_to`
`os.symlink`	`Path.symlink_to`
`os.readlink`	`Path.readlink`
`os.rename`	`Path.rename`
`os.replace`	`Path.replace`
`os.remove`, `os.unlink`	`Path.unlink`
`os.rmdir`	`Path.rmdir`
`os.chmod`	`Path.chmod`
`os.lchmod`	`Path.lchmod`

Footnotes

Protocols

pathlib.types

The `!pathlib.types` module provides types for static type checking.

3.14

A `typing.Protocol` describing the `Path.info <pathlib.Path.info>` attribute. Implementations may return cached results from their methods.

exists(*, follow_symlinks=True)

Return `True` if the path is an existing file or directory, or any other kind of file; return `False` if the path doesn't exist.

If *follow_symlinks* is `False`, return `True` for symlinks without checking if their targets exist.

is_dir(*, follow_symlinks=True)

Return `True` if the path is a directory, or a symbolic link pointing to a directory; return `False` if the path is (or points to) any other kind of file, or if it doesn't exist.

If *follow_symlinks* is `False`, return `True` only if the path is a directory (without following symlinks); return `False` if the path is any other kind of file, or if it doesn't exist.

is_file(*, follow_symlinks=True)

Return `True` if the path is a file, or a symbolic link pointing to a file; return `False` if the path is (or points to) a directory or other non-file, or if it doesn't exist.

If *follow_symlinks* is `False`, return `True` only if the path is a file (without following symlinks); return `False` if the path is a directory or other non-file, or if it doesn't exist.

is_symlink()

Return `True` if the path is a symbolic link (even if broken); return `False` if the path is a directory or any kind of file, or if it doesn't exist.

[^1]: `os.path.relpath` calls `~os.path.abspath` to make paths absolute and remove "`..`" parts, whereas `PurePath.relative_to` is a lexical operation that raises `ValueError` when its inputs' anchors differ (e.g. if one path is absolute and the other relative.)

[^2]: `os.path.expanduser` returns the path unchanged if the home directory can't be resolved, whereas `Path.expanduser` raises `RuntimeError`.

[^3]: `os.path.abspath` removes "`..`" components without resolving symlinks, which may change the meaning of the path, whereas `Path.absolute` leaves any "`..`" components in the path.

[^4]: `os.walk` always follows symlinks when categorizing paths into *dirnames* and *filenames*, whereas `Path.walk` categorizes all symlinks into *filenames* when *follow_symlinks* is false (the default.)

□□□ #2
□ □□□ □□□ 2026-10-31 17:16:32 UTC
□ □□□ □□□ 2026-10-31 17:16:36 UTC