mutapath

Release 0.17.1

'matfax'

Jul 13, 2023

CONTENTS

This library is for you if you are also annoyed that there is no mutable pathlib wrapper for use cases in which paths are often changed. mutapath solves this by wrapping both, the Python 3 pathlib library, and the alternate path library, and providing a mutable context manager for them.

CHAPTER

ONE

MUTAPATH CLASS

The MutaPath Class allows direct mutation of its attributes at any time, just as any mutable object. Once a file operation is called that is intended to modify its path, the underlying path is also mutated.

```
>>> from mutapath import MutaPath
```

```
>>> folder = MutaPath("/home/joe/doe/folder/sub")
>>> folder
Path('/home/joe/doe/folder/sub')
```

```
>>> folder.name = "top"
>>> folder
Path('/home/joe/doe/folder/top')
```

```
>>> next = MutaPath("/home/joe/doe/folder/next")
>>> next
Path('/home/joe/doe/folder/next')
```

```
>>> next.rename(folder)
>>> next
Path('/home/joe/doe/folder/top')
>>> next.exists()
True
>>> Path('/home/joe/doe/folder/sub').exists()
False
```

CHAPTER

TWO

PATH CLASS

This class is immutable by default, just as the `pathlib.Path`. However, it allows to open a editing context via `mutate()`. Moreover, there are additional contexts for file operations. They update the file and its path while closing the context. If the file operations don't succeed, they throw an exception and fall back to the original path value.

```
>>> from mutapath import Path
```

```
>>> folder = Path("/home/joe/doe/folder/sub")
>>> folder
Path('/home/joe/doe/folder/sub')
```

```
>>> folder.name = "top"
AttributeError: mutapath.Path is an immutable class, unless mutate() context is used.
>>> folder
Path('/home/joe/doe/folder/sub')
```

```
>>> with folder.mutate() as m:
...     m.name = "top"
>>> folder
Path('/home/joe/doe/folder/top')
```

```
>>> next = Path("/home/joe/doe/folder/next")
>>> next.copy(folder)
>>> next
Path('/home/joe/doe/folder/next')
>>> folder.exists()
True
>>> folder.remove()
```

```
>>> with next.renaming() as m:
...     m.stem = folder.stem
...     m.suffix = ".txt"
>>> next
Path("/home/joe/doe/folder/sub.txt")
>>> next.exists()
True
>>> next.with_name("next").exists()
False
```

For more in-depth examples, check the tests folder.

CHAPTER

THREE

LOCKS

Soft Locks can easily be accessed via the lazy lock property. Moreover, the mutable context managers in `Path` (i.e., `renaming`, `moving`, `copying`) allow implicit locking. The lock object is cached as long as the file is not mutated. Once the lock is mutated, it is released and regenerated, respecting the new file name.

```
>>> my_path = Path('/home/doe/folder/sub')
>>> with my_path.lock:
...     my_path.write_text("I can write")
```

CHAPTER

FOUR

HASHING

mutapath paths are hashable by caching the generated hash the first time it is accessed. However, it also adds a warning so that unintended hash usage is avoided. Once mutated after that, the generated hashes don't provide collision detection in binary trees anymore. Don't use them in sets or as keys in dicts. Use the explicit string representation instead, to make the hashing input transparent.

```
>>> p = Path("/home")
>>> hash(p)
1083235232
>>> hash(Path("/home")) == hash(p)
True
>>> with p.mutate() as m:
...     m.name = "home4"
>>> hash(p) # same hash
1083235232
>>> hash(Path("/home")) == hash(p) # they are not equal anymore
True
```

4.1 Documentation

Path([contained, posix, string_repr])	Immutable Path
MutaPath([contained, posix, string_repr])	Mutable Path
PathException	Exception about inconsistencies between the virtual path and the real file system.
DummyFileLock(lock_file[, timeout, mode, ...])	Create a new lock object.

4.1.1 mutapath.Path

class mutapath.**Path**(*contained:* Path | *Path* | *PurePath* | *str* = '', *, *posix: bool* | *None* = *None*, *string_repr: bool* | *None* = *None*)

Bases: object

Immutable Path

__init__(*contained:* Path | *Path* | *PurePath* | *str* = '', *, *posix: bool* | *None* = *None*, *string_repr: bool* | *None* = *None*)

Methods

absolute()	Return an absolute version of this path.
abspath()	Return an absolute path.
access(*args, **kwargs)	Return does the real user have access to this path.
as_posix()	Return the string representation of the path with forward (/) slashes.
as_uri()	Return the path as a 'file' URI.
basename()	**See also:** *name*, os.path.basename()
capitalize()	Return a capitalized version of the string.
casefold()	Return a version of the string suitable for caseless comparisons.
cd()	Change the current working directory to the specified path.
center(width[, fillchar])	Return a centered string of length width.
chdir()	Change the current working directory to the specified path.
chmod(mode)	Set the mode.
chown([uid, gid])	Change the owner and group by names or numbers.
chroot()	Change root directory to path.
chunks(size, *args, **kwargs)	Returns a generator yielding chunks of the file, so it can
clone(contained)	Clone this path with a new given wrapped path representation, having the same remaining attributes.
copy(dst, *[, follow_symlinks])	Copy data and mode bits ("cp src dst").
copy2(dst, *[, follow_symlinks])	Copy data and metadata.
copyfile(dst, *[, follow_symlinks])	Copy data from src to dst in the most efficient way possible.
copying([lock, timeout, method])	Create a copying context for this immutable path.
copymode(dst, *[, follow_symlinks])	Copy mode bits from src to dst.
copystat(dst, *[, follow_symlinks])	Copy file metadata
copytree(dst[, symlinks, ignore, ...])	Recursively copy a directory tree and return the destination directory.
count(sub[, start[, end]])	Return the number of non-overlapping occurrences of substring sub in string S[start:end].
dirs(*args, **kwargs)	List of this directory's subdirectories.
encode([encoding, errors])	Encode the string using the codec registered for encoding.
endswith(suffix[, start[, end]])	Return True if S ends with the specified suffix, False otherwise.
exists()	Test whether a path exists.
expand()	Clean up a filename by calling *expandvars()*, *expanduser()*, and *normpath()* on it.
expandtabs([tabsize])	Return a copy where all tab characters are expanded using spaces.
expanduser()	Expand ~ and ~user constructions.
expandvars()	Expand shell variables of form $var and ${var}.
files(*args, **kwargs)	List of the files in self.

continues on next page

Table 1 – continued from previous page

find(sub[, start[, end]])	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
fnmatch(pattern[, normcase])	Return `True` if *self.name* matches the given *pattern*.
format(*args, **kwargs)	Return a formatted version of S, using substitutions from args and kwargs.
format_map(mapping)	Return a formatted version of S, using substitutions from mapping.
get_owner()	Return the name of the owner of this file or directory.
getatime()	**See also:** *atime*, `os.path.getatime()`
getctime()	**See also:** *ctime*, `os.path.getctime()`
getcwd()	**See also:** `pathlib.Path.cwd()`
getmtime()	**See also:** *mtime*, `os.path.getmtime()`
getsize()	**See also:** *size*, `os.path.getsize()`
glob(pattern)	**See also:** `pathlib.Path.glob()`
group()	Return the group name of the file gid.
iglob(pattern)	Return an iterator of Path objects that match the pattern.
in_place([mode, buffering, encoding, ...])	A context in which a file may be re-written in-place with new content.
index(sub[, start[, end]])	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
is_absolute()	True if the path is absolute (has both a root and, if applicable, a drive).
is_block_device()	Whether this path is a block device.
is_char_device()	Whether this path is a character device.
is_dir()	Whether this path is a directory.
is_fifo()	Whether this path is a FIFO.
is_file()	Whether this path is a regular file (also True for symlinks pointing to regular files).
is_mount()	Check if this path is a POSIX mount point
is_reserved()	Return True if the path contains one of the special names reserved by the system, if any.
is_socket()	Whether this path is a socket.
is_symlink()	Whether this path is a symbolic link.

continues on next page

Table 1 – continued from previous page

isabs()	`>>> Path('.').isabs()`
isalnum()	Return True if the string is an alpha-numeric string, False otherwise.
isalpha()	Return True if the string is an alphabetic string, False otherwise.
isascii()	Return True if all characters in the string are ASCII, False otherwise.
isdecimal()	Return True if the string is a decimal string, False otherwise.
isdigit()	Return True if the string is a digit string, False otherwise.
isdir()	Return true if the pathname refers to an existing directory.
isfile()	Test whether a path is a regular file
isidentifier()	Return True if the string is a valid Python identifier, False otherwise.
islink()	Test whether a path is a symbolic link
islower()	Return True if the string is a lowercase string, False otherwise.
ismount()	`>>> Path('.').ismount()`
isnumeric()	Return True if the string is a numeric string, False otherwise.
isprintable()	Return True if the string is printable, False otherwise.
isspace()	Return True if the string is a whitespace string, False otherwise.
istitle()	Return True if the string is a title-cased string, False otherwise.
isupper()	Return True if the string is an uppercase string, False otherwise.
iterdir()	Iterate over the files in this directory.
join(iterable, /)	Concatenate any number of strings.
joinpath(*others)	**See also:** `pathlib.PurePath.joinpath()`
lchmod(mode)	Like chmod(), except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.
lines([encoding, errors, retain])	Open this file, read all lines, return them in a list.
link(newpath)	Create a hard link at *newpath*, pointing to this file.
link_to(target)	Make the target path a hard link pointing to this path.
listdir([match])	List of items in this directory.
ljust(width[, fillchar])	Return a left-justified string of length width.
lower()	Return a copy of the string converted to lowercase.
lstat()	Like *stat()*, but do not follow symbolic links.
lstrip([chars])	Return a copy of the string with leading whitespace removed.

continues on next page

Table 1 – continued from previous page

makedirs(name [[, mode, exist_ok])	Super-mkdir; create a leaf directory and all intermediate ones.
makedirs_p([mode])	Like *makedirs()*, but does not raise an exception if the directory already exists.
match(path_pattern)	Return True if this path matches the given pattern.
merge_tree(dst[, symlinks, copy_function, ...])	Copy entire contents of self to dst, overwriting existing contents in dst with those in self.
mkdir([mode])	Create a directory.
mkdir_p([mode])	Like *mkdir()*, but does not raise an exception if the directory already exists.
move(dst[, copy_function])	Recursively move a file or directory to another location.
moving([lock, timeout, method])	Create a moving context for this immutable path.
mutate()	Create a mutable context for this immutable path.
normcase()	Normalize case of pathname.
normpath()	Normalize path, eliminating double slashes, etc.
open(*args, **kwargs)	Open file and return a stream.
partition(sep, /)	Partition the string into three parts using the given separator.
parts()	`>>> Path('/foo/bar/baz').parts()`
pathconf(name)	Return the configuration limit name for the file or directory path.
posix_string()	Get this path as string with posix-like separators (i.e., '/').
read_bytes()	Return the contents of this file as bytes.
read_hash(hash_name)	Calculate given hash for this file.
read_hexhash(hash_name)	Calculate given hash for this file, returning hexdigest.
read_md5()	Calculate the md5 hash for this file.
read_text([encoding, errors])	Open this file, read it in, return the content as a string.
readlink()	Return the path to which this symbolic link points.
readlinkabs()	Return the path to which this symbolic link points.
realpath()	Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path.
relative_to(*other)	Return the relative path to another path identified by the passed arguments.
relpath([start])	Return this path as a relative path, based from *start*, which defaults to the current working directory.
relpathto(dest)	Return a relative path from *self* to *dest*.
remove()	Remove a file (same as unlink()).
remove_p()	Like *remove()*, but does not raise an exception if the file does not exist.
removedirs(name)	Super-rmdir; remove a leaf directory and all empty intermediate ones.
removedirs_p()	Like *removedirs()*, but does not raise an exception if the directory is not empty or does not exist.
rename(new)	Rename a file or directory.
renames(old, new)	Super-rename; create directories as necessary and delete any left empty.
renaming([lock, timeout, method])	Create a renaming context for this immutable path.

continues on next page

Table 1 – continued from previous page

replace(old, new[, count])	Return a copy with all occurrences of substring old replaced by new.
resolve([strict])	Make the path absolute, resolving all symlinks on the way and also normalizing it (for example turning slashes into backslashes under Windows).
rfind(sub[, start[, end]])	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
rglob(pattern)	Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.
rindex(sub[, start[, end]])	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
rjust(width[, fillchar])	Return a right-justified string of length width.
rmdir()	Remove a directory.
rmdir_p()	Like *rmdir()*, but does not raise an exception if the directory is not empty or does not exist.
rmtree([ignore_errors, onerror])	Recursively delete a directory tree.
rmtree_p()	Like *rmtree()*, but does not raise an exception if the directory does not exist.
rpartition(sep, /)	Partition the string into three parts using the given separator.
rsplit([sep, maxsplit])	Return a list of the words in the string, using sep as the delimiter string.
rstrip([chars])	Return a copy of the string with trailing whitespace removed.
samefile(other)	Test whether two pathnames reference the same actual file or directory
set_atime(value)	
set_mtime(value)	
split([sep, maxsplit])	Return a list of the words in the string, using sep as the delimiter string.
splitall()	Return a list of the path components in this path.
splitdrive()	Return two-tuple of `.drive` and rest without drive.
splitext()	Return two-tuple of `.stripext()` and `.ext`.
splitlines([keepends])	Return a list of the lines in the string, breaking at line boundaries.
splitpath()	Return two-tuple of `.parent`, `.name`.
startfile()	Open this path in a platform-dependant manner.
startswith(prefix[, start[, end]])	Return True if S starts with the specified prefix, False otherwise.
stat()	Perform a `stat()` system call on this path.
statvfs()	Perform a `statvfs()` system call on this path.
strip([chars])	Return a copy of the string with leading and trailing whitespace removed.
stripext()	Remove one file extension from the path.
swapcase()	Convert uppercase characters to lowercase and lowercase characters to uppercase.
symlink([newlink])	Create a symbolic link at *newlink*, pointing here.
symlink_to(target[, target_is_directory])	Make this path a symlink pointing to the target path.

continues on next page

Table 1 – continued from previous page

title()	Return a version of the string where each word is titlecased.
touch()	Set the access/modified times of this file to the current time.
translate(table, /)	Replace each character in the string using the given translation table.
unlink()	Remove a file (same as unlink()).
unlink_p()	Like *remove()*, but does not raise an exception if the file does not exist.
upper()	Return a copy of the string converted to uppercase.
using_module(module)	
utime(*args, **kwargs)	Set the access and modified times of this file.
walk([match, errors])	Iterator over files and subdirs, recursively.
walkdirs(*args, **kwargs)	Iterator over subdirs, recursively.
walkfiles(*args, **kwargs)	Iterator over files, recursively.
with_base(base[, strip_length])	Clone this path with a new base.
with_name(new_name)	**See also:** `pathlib.PurePath.with_name()`
with_parent(new_parent)	Clone this path with a new parent.
with_poxis_enabled([enable])	Clone this path in posix format with posix-like separators (i.e., '/').
with_stem(new_stem)	Clone this path with a new stem.
with_string_repr_enabled([enable])	Clone this path in with string representation enabled.
with_suffix(suffix)	Return a new path with the file suffix changed (or added, if none)
write_bytes(bytes[, append])	Open this file and write the given bytes to it.
write_lines(lines[, encoding, errors, ...])	Write the given lines of text to this file.
write_text(text[, encoding, errors, ...])	Write the given text to this file.
zfill(width, /)	Pad a numeric string with zeros on the left, to fill a field of the given width.

mutapath.Path.absolute

`Path.`**`absolute`**`()`

Return an absolute version of this path. This function works even if the path doesn't point to anything.

No normalization is done, i.e. all '.' and '..' will be kept along. Use resolve() to get the canonical path to a file.

mutapath.Path.abspath

Path.**abspath**()

Return an absolute path.

mutapath.Path.access

Path.**access**(**args*, ***kwargs*)

Return does the real user have access to this path.

```
>>> Path('.').access(os.F_OK)
True
```

See also:

`os.access()`

mutapath.Path.as_posix

Path.**as_posix**()

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

mutapath.Path.as_uri

Path.**as_uri**()

Return the path as a 'file' URI.

mutapath.Path.basename

Path.**basename**()

See also:

name, `os.path.basename()`

mutapath.Path.capitalize

Path.**capitalize**()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

mutapath.Path.casefold

Path.**casefold**()

Return a version of the string suitable for caseless comparisons.

mutapath.Path.cd

Path.**cd**()

Change the current working directory to the specified path.

path may always be specified as a string. On some platforms, path may also be specified as an open file descriptor.

If this functionality is unavailable, using it raises an exception.

mutapath.Path.center

Path.**center**(*width*, *fillchar=' '*, */*)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

mutapath.Path.chdir

Path.**chdir**()

Change the current working directory to the specified path.

path may always be specified as a string. On some platforms, path may also be specified as an open file descriptor.

If this functionality is unavailable, using it raises an exception.

mutapath.Path.chmod

Path.**chmod**(*mode*)

Set the mode. May be the new mode (os.chmod behavior) or a symbolic mode.

```
>>> a_file = Path(getfixture('tmp_path')).joinpath('afile.txt').touch()
>>> a_file.chmod(0o700)
Path(...
>>> a_file.chmod('u+x')
Path(...
```

See also:

os.chmod()

mutapath.Path.chown

Path.**chown**(*uid=-1*, *gid=-1*)

Change the owner and group by names or numbers.

See also:

os.chown()

mutapath.Path.chroot

Path.**chroot**()

Change root directory to path.

mutapath.Path.chunks

Path.**chunks**(*size*, **args*, ***kwargs*)

Returns a generator yielding chunks of the file, so it can
be read piece by piece with a simple for loop.

Any argument you pass after *size* will be passed to *open()*.

Example

```
>>> hash = hashlib.md5()
>>> for chunk in Path("NEWS.rst").chunks(8192, mode='rb'):
...     hash.update(chunk)
```

This will read the file by chunks of 8192 bytes.

mutapath.Path.clone

Path.**clone**(*contained*) → *Path*

Clone this path with a new given wrapped path representation, having the same remaining attributes. :param contained: the new contained path element :return: the cloned path

mutapath.Path.copy

Path.**copy**(*dst*, ***, *follow_symlinks=True*)

Copy data and mode bits ("cp src dst"). Return the file's destination.

The destination may be a directory.

If follow_symlinks is false, symlinks won't be followed. This resembles GNU's "cp -P src dst".

If source and destination are the same file, a SameFileError will be raised.

mutapath.Path.copy2

Path.**copy2**(*dst*, *, *follow_symlinks=True*)

Copy data and metadata. Return the file's destination.

Metadata is copied with copystat(). Please see the copystat function for more information.

The destination may be a directory.

If follow_symlinks is false, symlinks won't be followed. This resembles GNU's "cp -P src dst".

mutapath.Path.copyfile

Path.**copyfile**(*dst*, *, *follow_symlinks=True*)

Copy data from src to dst in the most efficient way possible.

If follow_symlinks is not set and src is a symbolic link, a new symlink will be created instead of copying the file it points to.

mutapath.Path.copying

Path.**copying**(*lock=True, timeout=1, method: ~typing.Callable[[~mutapath.immutapath.Path, ~mutapath.immutapath.Path], ~mutapath.immutapath.Path] = <function copy>*)

Create a copying context for this immutable path. The external value is only changed if the copying succeeds.

Parameters

- **timeout** – the timeout in seconds how long the lock file should be acquired
- **lock** – if the source file should be locked as long as this context is open
- **method** – an alternative method that copies the path and returns the new path (e.g., shutil.copy2)

Example

```
>>> with Path('/home/doe/folder/a.txt').copying() as mut:
...     mut.stem = "b"
Path('/home/doe/folder/b.txt')
```

mutapath.Path.copymode

Path.**copymode**(*dst*, *, *follow_symlinks=True*)

Copy mode bits from src to dst.

If follow_symlinks is not set, symlinks aren't followed if and only if both *src* and *dst* are symlinks. If *lchmod* isn't available (e.g. Linux) this method does nothing.

mutapath.Path.copystat

Path.**copystat**(*dst*, *, *follow_symlinks=True*)

Copy file metadata

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

If the optional flag *follow_symlinks* is not set, symlinks aren't followed if and only if both *src* and *dst* are symlinks.

mutapath.Path.copytree

Path.**copytree**(*dst*, *symlinks=False*, *ignore=None*, *copy_function=<function copy2>*, *ignore_dangling_symlinks=False*, *dirs_exist_ok=False*)

Recursively copy a directory tree and return the destination directory.

dirs_exist_ok dictates whether to raise an exception in case dst or any missing parent directory already exists.

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

If the optional symlinks flag is true, symbolic links in the source tree result in symbolic links in the destination tree; if it is false, the contents of the files pointed to by symbolic links are copied. If the file pointed by the symlink doesn't exist, an exception will be added in the list of errors raised in an Error exception at the end of the copy process.

You can set the optional ignore_dangling_symlinks flag to true if you want to silence this exception. Notice that this has no effect on platforms that don't support os.symlink.

The optional ignore argument is a callable. If given, it is called with the *src* parameter, which is the directory being visited by copytree(), and *names* which is the list of *src* contents, as returned by os.listdir():

callable(src, names) -> ignored_names

Since copytree() is called recursively, the callable will be called once for each directory that is copied. It returns a list of names relative to the *src* directory that should not be copied.

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

mutapath.Path.count

Path.**count**(*sub*[, *start*[, *end*]]) → int

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

mutapath.Path.dirs

Path.**dirs**(**args*, ***kwargs*)

List of this directory's subdirectories.

The elements of the list are Path objects. This does not walk recursively into subdirectories (but see *walkdirs()*).

Accepts parameters to *listdir()*.

mutapath.Path.encode

Path.**encode**(*encoding='utf-8'*, *errors='strict'*)

Encode the string using the codec registered for encoding.

encoding
The encoding in which to encode the string.

errors
The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

mutapath.Path.endswith

Path.**endswith**(*suffix*[, *start*[, *end*]]) → bool

Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

mutapath.Path.exists

Path.**exists**()

Test whether a path exists. Returns False for broken symbolic links

mutapath.Path.expand

Path.**expand**()

Clean up a filename by calling *expandvars()*, *expanduser()*, and *normpath()* on it.

This is commonly everything needed to clean up a filename read from a configuration file, for example.

mutapath.Path.expandtabs

Path.**expandtabs**(*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

mutapath.Path.expanduser

Path.**expanduser**()

Expand ~ and ~user constructions. If user or $HOME is unknown, do nothing.

mutapath.Path.expandvars

Path.**expandvars**()

Expand shell variables of form $var and ${var}. Unknown variables are left unchanged.

mutapath.Path.files

Path.**files**(**args*, ***kwargs*)

List of the files in self.

The elements of the list are Path objects. This does not walk into subdirectories (see *walkfiles()*).

Accepts parameters to *listdir()*.

mutapath.Path.find

Path.**find**(*sub*[, *start*[, *end*]]) → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

mutapath.Path.fnmatch

Path.**fnmatch**(*pattern*, *normcase=None*)

Return `True` if *self.name* matches the given *pattern*.

***pattern* - A filename pattern with wildcards,**
for example `'*.py'`. If the pattern contains a *normcase* attribute, it is applied to the name and path prior to comparison.

***normcase* - (optional) A function used to normalize the pattern and**
filename before matching. Defaults to normcase from `self.module`, `os.path.normcase()`.

See also:

`fnmatch.fnmatch()`

mutapath.Path.format

Path.**format**(**args*, ***kwargs*) → str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

mutapath.Path.format_map

Path.**format_map**(*mapping*) → str

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

mutapath.Path.get_owner

Path.**get_owner**()

Return the name of the owner of this file or directory. Follow symbolic links.

See also:

owner

mutapath.Path.getatime

Path.**getatime**()

See also:

atime, os.path.getatime()

mutapath.Path.getctime

Path.**getctime**()

See also:

ctime, os.path.getctime()

mutapath.Path.getcwd

classmethod Path.**getcwd**() → *Path*

See also:

pathlib.Path.cwd()

mutapath.Path.getmtime

Path.**getmtime**()

See also:

mtime, os.path.getmtime()

mutapath.Path.getsize

Path.**getsize**()

See also:

size, os.path.getsize()

mutapath.Path.glob

Path.**glob**(*pattern*) → Iterable[*Path*]

See also:

pathlib.Path.glob()

mutapath.Path.group

Path.**group**()

Return the group name of the file gid.

mutapath.Path.iglob

Path.**iglob**(*pattern*)

Return an iterator of Path objects that match the pattern.

pattern - a path relative to this directory, with wildcards.

For example, `Path('/users').iglob('*/bin/*')` returns an iterator of all the files users have in their `bin` directories.

See also:

glob.iglob()

Note: Glob is **not** recursive, even when using **. To do recursive globbing see *walk()*, *walkdirs()* or *walkfiles()*.

---

mutapath.Path.in_place

Path.**in_place**(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *backup_extension=None*)

A context in which a file may be re-written in-place with new content.

Yields a tuple of `(readable, writable)` file objects, where *writable* replaces *readable*.

If an exception occurs, the old file is restored, removing the written data.

Mode *must not* use `'w'`, `'a'`, or `'+'`; only read-only-modes are allowed. A `ValueError` is raised on invalid modes.

For example, to add line numbers to a file:

```
p = Path(filename)
assert p.isfile()
with p.in_place() as (reader, writer):
    for number, line in enumerate(reader, 1):
        writer.write('{0:3}: '.format(number)))
        writer.write(line)
```

Thereafter, the file at *filename* will have line numbers in it.

mutapath.Path.index

Path.**index**(*sub*[, *start*[, *end*]]) → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

mutapath.Path.is_absolute

Path.**is_absolute**()

True if the path is absolute (has both a root and, if applicable, a drive).

mutapath.Path.is_block_device

Path.**is_block_device**()

Whether this path is a block device.

mutapath.Path.is_char_device

Path.**is_char_device**()

Whether this path is a character device.

mutapath.Path.is_dir

Path.**is_dir**()

Whether this path is a directory.

mutapath.Path.is_fifo

Path.**is_fifo**()

Whether this path is a FIFO.

mutapath.Path.is_file

Path.**is_file**()

Whether this path is a regular file (also True for symlinks pointing to regular files).

mutapath.Path.is_mount

Path.**is_mount**()

Check if this path is a POSIX mount point

mutapath.Path.is_reserved

Path.**is_reserved**()

Return True if the path contains one of the special names reserved by the system, if any.

mutapath.Path.is_socket

Path.**is_socket**()

Whether this path is a socket.

mutapath.Path.is_symlink

Path.**is_symlink**()

Whether this path is a symbolic link.

mutapath.Path.isabs

Path.**isabs**()

```
>>> Path('.').isabs()
False
```

See also:

os.path.isabs()

mutapath.Path.isalnum

Path.**isalnum**()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

mutapath.Path.isalpha

Path.**isalpha**()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

mutapath.Path.isascii

Path.**isascii**()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

mutapath.Path.isdecimal

Path.**isdecimal**()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

mutapath.Path.isdigit

Path.**isdigit**()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

mutapath.Path.isdir

Path.**isdir**()

Return true if the pathname refers to an existing directory.

mutapath.Path.isfile

Path.**isfile**()

Test whether a path is a regular file

mutapath.Path.isidentifier

Path.**isidentifier**()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

mutapath.Path.islink

Path.**islink**()

Test whether a path is a symbolic link

mutapath.Path.islower

Path.**islower**()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

mutapath.Path.ismount

Path.**ismount**()

```
>>> Path('.').ismount()
False
```

See also:

os.path.ismount()

mutapath.Path.isnumeric

Path.**isnumeric**()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

mutapath.Path.isprintable

Path.**isprintable**()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

mutapath.Path.isspace

Path.**isspace**()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

mutapath.Path.istitle

Path.**istitle**()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

mutapath.Path.isupper

Path.**isupper**()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

mutapath.Path.iterdir

Path.**iterdir**()

Iterate over the files in this directory. Does not yield any result for the special paths '.' and '..'.

mutapath.Path.join

Path.**join**(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

mutapath.Path.joinpath

Path.**joinpath**(**others*) → *Path*

See also:

`pathlib.PurePath.joinpath()`

mutapath.Path.lchmod

Path.**lchmod**(*mode*)

Like chmod(), except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.

mutapath.Path.lines

Path.**lines**(*encoding=None*, *errors=None*, *retain=True*)

Open this file, read all lines, return them in a list.

Optional arguments:

***encoding* - The Unicode encoding (or character set) of**
the file. The default is `None`, meaning use `locale.getpreferredencoding()`.

***errors* - How to handle Unicode errors; see**
open for the options. Default is `None` meaning "strict".

***retain* - If `True` (default), retain newline characters,**
but translate all newline characters to `\n`. If `False`, newline characters are omitted.

See also:

text()

mutapath.Path.link

Path.**link**(*newpath*)

Create a hard link at *newpath*, pointing to this file.

See also:

`os.link()`

mutapath.Path.link_to

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

Make the target path a hard link pointing to this path.

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

mutapath.Path.listdir

Path.**listdir**(*match=None*)

List of items in this directory.

Use *files()* or *dirs()* instead if you want a listing of just files or just subdirectories.

The elements of the list are Path objects.

With the optional *match* argument, a callable, only return items whose names match the given pattern.

See also:

files(), *dirs()*

mutapath.Path.ljust

Path.**ljust**(*width*, *fillchar=' '*, */*)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

mutapath.Path.lower

Path.**lower**()

Return a copy of the string converted to lowercase.

mutapath.Path.lstat

Path.**lstat**()

Like *stat()*, but do not follow symbolic links.

```
>>> Path('.').lstat() == Path('.').stat()
True
```

See also:

stat(), os.lstat()

mutapath.Path.lstrip

Path.**lstrip**(*chars=None*, */*)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

mutapath.Path.makedirs

Path.**makedirs**(*name [, mode=0o777][, exist_ok=False]*)

Super-mkdir; create a leaf directory and all intermediate ones. Works like mkdir, except that any intermediate path segment (not just the rightmost) will be created if it does not exist. If the target directory already exists, raise an OSError if exist_ok is False. Otherwise no exception is raised. This is recursive.

mutapath.Path.makedirs_p

Path.**makedirs_p**(*mode=511*)

Like *makedirs()*, but does not raise an exception if the directory already exists.

mutapath.Path.match

Path.**match**(*path_pattern*)

Return True if this path matches the given pattern.

mutapath.Path.merge_tree

Path.**merge_tree**(*dst*, *symlinks=False*, ***, *copy_function=<function copy2>*, *ignore=<function Path.<lambda>>*)

Copy entire contents of self to dst, overwriting existing contents in dst with those in self.

Pass `symlinks=True` to copy symbolic links as links.

Accepts a `copy_function`, similar to copytree.

To avoid overwriting newer files, supply a copy function wrapped in `only_newer`. For example:

```
src.merge_tree(dst, copy_function=only_newer(shutil.copy2))
```

mutapath.Path.mkdir

Path.**mkdir**(*mode=511*)

Create a directory.

If dir_fd is not None, it should be a file descriptor open to a directory,
and path should be relative; path will then be relative to that directory.

dir_fd may not be implemented on your platform.
If it is unavailable, using it will raise a NotImplementedError.

The mode argument is ignored on Windows.

mutapath.Path.mkdir_p

Path.**mkdir_p**(*mode=511*)

Like *mkdir()*, but does not raise an exception if the directory already exists.

mutapath.Path.move

Path.**move**(*dst*, *copy_function=<function copy2>*)

Recursively move a file or directory to another location. This is similar to the Unix "mv" command. Return the file or directory's destination.

If the destination is a directory or a symlink to a directory, the source is moved inside the directory. The destination path must not already exist.

If the destination already exists but is not a directory, it may be overwritten depending on os.rename() semantics.

If the destination is on our current filesystem, then rename() is used. Otherwise, src is copied to the destination and then removed. Symlinks are recreated under the new name if os.rename() fails because of cross filesystem renames.

The optional *copy_function* argument is a callable that will be used to copy the source or it will be delegated to *copytree*. By default, copy2() is used, but any function that supports the same signature (like copy()) can be used.

A lot more could be done here… A look at a mv.c shows a lot of the issues this implementation glosses over.

mutapath.Path.moving

Path.**moving**(*lock=True, timeout=1, method: ~typing.Callable[[~os.PathLike, ~os.PathLike], str] = <function move>*)

Create a moving context for this immutable path. The external value is only changed if the moving succeeds.

Parameters

- **timeout** – the timeout in seconds how long the lock file should be acquired
- **lock** – if the source file should be locked as long as this context is open
- **method** – an alternative method that moves the path and returns the new path

Example

```
>>> with Path('/home/doe/folder/a.txt').moving() as mut:
...     mut.stem = "b"
Path('/home/doe/folder/b.txt')
```

mutapath.Path.mutate

Path.**mutate**()

Create a mutable context for this immutable path.

Example

```
>>> with Path('/home/doe/folder/sub').mutate() as mut:
...     mut.name = "top"
Path('/home/doe/folder/top')
```

mutapath.Path.normcase

Path.**normcase**()

Normalize case of pathname. Has no effect under Posix

mutapath.Path.normpath

Path.**normpath**()

Normalize path, eliminating double slashes, etc.

mutapath.Path.open

Path.**open**(**args*, ***kwargs*)

Open file and return a stream. Raise OSError upon failure.

file is either a text or byte string giving the name (and the path if the file isn't in the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless closefd is set to False.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to 'r' which means open for reading in text mode. Other common values are 'w' for writing (truncating the file if it already exists), 'x' for creating and writing to a new file, and 'a' for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent: locale.getpreferredencoding(False) is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	create a new file and open it for writing
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)
'U'	universal newline mode (deprecated)

The default mode is 'rt' (open for reading text). For binary random access, the mode 'w+b' opens and truncates the file to 0 bytes, while 'r+b' opens the file without truncation. The 'x' mode implies 'w' and raises an *FileExistsError* if the file already exists.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn't. Files opened in binary mode (appending 'b' to the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when 't' is appended to the mode argument), the contents of the file are returned as strings, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

'U' mode is deprecated and will raise an exception in future versions of Python. It has no effect in Python 3. Use newline to control universal newlines mode.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on *io.DEFAULT_BUFFER_SIZE*. On many systems, the buffer will typically be 4096 or 8192 bytes long.
- "Interactive" text files (files for which isatty() returns True) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent, but any encoding supported by Python can be passed. See the codecs module for the list of supported encodings.

errors is an optional string that specifies how encoding errors are to be handled—this argument should not be used in binary mode. Pass 'strict' to raise a ValueError exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) See the documentation for codecs.register or run 'help(codecs.Codec)' for a list of the permitted encoding error strings.

newline controls how universal newlines works (it only applies to text mode). It can be None, '', 'n', 'r', and 'rn'. It works as follows:

- On input, if newline is None, universal newlines mode is enabled. Lines in the input can end in 'n', 'r', or 'rn', and these are translated into 'n' before being returned to the caller. If it is '', universal newline mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- On output, if newline is None, any 'n' characters written are translated to the system default line separator, os.linesep. If newline is '' or 'n', no translation takes place. If newline is any of the other legal values, any 'n' characters written are translated to the given string.

If closefd is False, the underlying file descriptor will be kept open when the file is closed. This does not work when a file name is given and must be True in that case.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*file*, *flags*). *opener* must return an open file descriptor (passing os.open as *opener* results in functionality similar to passing None).

open() returns a file object whose type depends on the mode, and through which the standard file operations such as reading and writing are performed. When open() is used to open a file in a text mode ('w', 'r', 'wt', 'rt', etc.), it returns a TextIOWrapper. When used to open a file in a binary mode, the returned class varies: in read binary mode, it returns a BufferedReader; in write binary and append binary modes, it returns a BufferedWriter, and in read/write mode, it returns a BufferedRandom.

It is also possible to use a string or bytearray as a file for both reading and writing. For strings StringIO can be used like a file opened in a text mode, and for bytes a BytesIO can be used like a file opened in a binary mode.

mutapath.Path.partition

Path.**partition**(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

mutapath.Path.parts

Path.**parts**()

```
>>> Path('/foo/bar/baz').parts()
(Path('/'), 'foo', 'bar', 'baz')
```

mutapath.Path.pathconf

Path.**pathconf**(*name*)

Return the configuration limit name for the file or directory path.

If there is no limit, return -1. On some platforms, path may also be specified as an open file descriptor.

If this functionality is unavailable, using it raises an exception.

mutapath.Path.posix_string

Path.**posix_string**() → str

Get this path as string with posix-like separators (i.e., '/').

Example

```
>>> Path("\home\\doe/folder\sub").with_poxis_enabled()
'/home/joe/doe/folder/sub'
```

mutapath.Path.read_bytes

Path.**read_bytes**()

Return the contents of this file as bytes.

mutapath.Path.read_hash

Path.**read_hash**(*hash_name*)

Calculate given hash for this file.

List of supported hashes can be obtained from `hashlib` package. This reads the entire file.

See also:

`hashlib.hash.digest()`

mutapath.Path.read_hexhash

Path.**read_hexhash**(*hash_name*)

Calculate given hash for this file, returning hexdigest.

List of supported hashes can be obtained from `hashlib` package. This reads the entire file.

See also:

`hashlib.hash.hexdigest()`

mutapath.Path.read_md5

Path.**read_md5**()

Calculate the md5 hash for this file.

This reads through the entire file.

See also:

read_hash()

mutapath.Path.read_text

Path.**read_text**(*encoding=None*, *errors=None*)

Open this file, read it in, return the content as a string.

Optional parameters are passed to *open()*.

See also:

lines()

mutapath.Path.readlink

Path.**readlink**()

Return the path to which this symbolic link points.

The result may be an absolute or a relative path.

See also:

readlinkabs(), os.readlink()

mutapath.Path.readlinkabs

Path.**readlinkabs**()

Return the path to which this symbolic link points.

The result is always an absolute path.

See also:

readlink(), os.readlink()

mutapath.Path.realpath

Path.**realpath**()

Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path.

mutapath.Path.relative_to

Path.**relative_to**(**other*)

Return the relative path to another path identified by the passed arguments. If the operation is not possible (because this is not a subpath of the other path), raise ValueError.

mutapath.Path.relpath

Path.**relpath**(*start='.'*)

Return this path as a relative path, based from *start*, which defaults to the current working directory.

mutapath.Path.relpathto

Path.**relpathto**(*dest*)

Return a relative path from *self* to *dest*.

If there is no relative path from *self* to *dest*, for example if they reside on different drives in Windows, then this returns `dest.abspath()`.

mutapath.Path.remove

Path.**remove**()

Remove a file (same as unlink()).

If dir_fd is not None, it should be a file descriptor open to a directory,
and path should be relative; path will then be relative to that directory.

dir_fd may not be implemented on your platform.
If it is unavailable, using it will raise a NotImplementedError.

mutapath.Path.remove_p

Path.**remove_p**()

Like *remove()*, but does not raise an exception if the file does not exist.

mutapath.Path.removedirs

Path.**removedirs**(*name*)

Super-rmdir; remove a leaf directory and all empty intermediate ones. Works like rmdir except that, if the leaf directory is successfully removed, directories corresponding to rightmost path segments will be pruned away until either the whole path is consumed or an error occurs. Errors during this latter phase are ignored – they generally mean that a directory was not empty.

mutapath.Path.removedirs_p

Path.**removedirs_p**()

Like *removedirs()*, but does not raise an exception if the directory is not empty or does not exist.

mutapath.Path.rename

Path.**rename**(*new*)

Rename a file or directory.

If either src_dir_fd or dst_dir_fd is not None, it should be a file
descriptor open to a directory, and the respective path string (src or dst) should be relative; the path will then be relative to that directory.

src_dir_fd and dst_dir_fd, may not be implemented on your platform.
If they are unavailable, using them will raise a NotImplementedError.

mutapath.Path.renames

Path.**renames**(*old*, *new*)

Super-rename; create directories as necessary and delete any left empty. Works like rename, except creation of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned until either the whole path is consumed or a nonempty directory is found.

Note: this function can fail with the new directory structure made if you lack permissions needed to unlink the leaf directory or file.

mutapath.Path.renaming

Path.**renaming**(*lock=True, timeout=1, method: ~typing.Callable[[str, str], None] = <built-in function rename>*)

Create a renaming context for this immutable path. The external value is only changed if the renaming succeeds.

Parameters

- **timeout** – the timeout in seconds how long the lock file should be acquired
- **lock** – if the source file should be locked as long as this context is open
- **method** – an alternative method that renames the path (e.g., os.renames)

Example

```
>>> with Path('/home/doe/folder/a.txt').renaming() as mut:
...     mut.stem = "b"
Path('/home/doe/folder/b.txt')
```

mutapath.Path.replace

Path.**replace**(*old*, *new*, *count=-1*, */*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

mutapath.Path.resolve

Path.**resolve**(*strict=False*)

Make the path absolute, resolving all symlinks on the way and also normalizing it (for example turning slashes into backslashes under Windows).

mutapath.Path.rfind

Path.**rfind**(*sub*[, *start*[, *end*]]) → int

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

mutapath.Path.rglob

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

Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.

mutapath.Path.rindex

Path.**rindex**(*sub*[, *start*[, *end*]]) → int

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

mutapath.Path.rjust

Path.**rjust**(*width*, *fillchar=' '*, */*)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

mutapath.Path.rmdir

Path.**rmdir**()

Remove a directory.

If dir_fd is not None, it should be a file descriptor open to a directory,
and path should be relative; path will then be relative to that directory.

dir_fd may not be implemented on your platform.
If it is unavailable, using it will raise a NotImplementedError.

mutapath.Path.rmdir_p

Path.**rmdir_p**()

Like *rmdir()*, but does not raise an exception if the directory is not empty or does not exist.

mutapath.Path.rmtree

Path.**rmtree**(*ignore_errors=False*, *onerror=None*)

Recursively delete a directory tree.

If ignore_errors is set, errors are ignored; otherwise, if onerror is set, it is called to handle the error with arguments (func, path, exc_info) where func is platform and implementation dependent; path is the argument to that function that caused it to fail; and exc_info is a tuple returned by sys.exc_info(). If ignore_errors is false and onerror is None, an exception is raised.

mutapath.Path.rmtree_p

Path.**rmtree_p**()

Like *rmtree()*, but does not raise an exception if the directory does not exist.

mutapath.Path.rpartition

Path.**rpartition**(*sep*, */*)

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

mutapath.Path.rsplit

Path.**rsplit**(*sep=None*, *maxsplit=-1*)

Return a list of the words in the string, using sep as the delimiter string.

sep
The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit
Maximum number of splits to do. -1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

mutapath.Path.rstrip

Path.**rstrip**(*chars=None*, */*)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

mutapath.Path.samefile

Path.**samefile**(*other*)

Test whether two pathnames reference the same actual 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.

mutapath.Path.set_atime

Path.**set_atime**(*value*)

mutapath.Path.set_mtime

Path.**set_mtime**(*value*)

mutapath.Path.split

Path.**split**(*sep=None*, *maxsplit=-1*)

Return a list of the words in the string, using sep as the delimiter string.

sep
: The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit
: Maximum number of splits to do. -1 (the default value) means no limit.

mutapath.Path.splitall

Path.**splitall**()

Return a list of the path components in this path.

The first item in the list will be a Path. Its value will be either `os.curdir`, `os.pardir`, empty, or the root directory of this path (for example, `'/'` or `'C:\\'`). The other items in the list will be strings.

`Path.joinpath(*result)` will yield the original path.

```
>>> Path('/foo/bar/baz').splitall()
[Path('/'), 'foo', 'bar', 'baz']
```

mutapath.Path.splitdrive

Path.**splitdrive**()

Return two-tuple of `.drive` and rest without drive.

Split the drive specifier from this path. If there is no drive specifier, *`p.drive`* is empty, so the return value is simply `(Path(''), p)`. This is always the case on Unix.

See also:

`os.path.splitdrive()`

mutapath.Path.splitext

Path.**splitext**()

Return two-tuple of `.stripext()` and `.ext`.

Split the filename extension from this path and return the two parts. Either part may be empty.

The extension is everything from `'.'` to the end of the last path segment. This has the property that if `(a, b) == p.splitext()`, then `a + b == p`.

See also:

`os.path.splitext()`

mutapath.Path.splitlines

Path.**splitlines**(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

mutapath.Path.splitpath

Path.**splitpath**()

Return two-tuple of `.parent`, `.name`.

See also:

`parent`, *`name`*, `os.path.split()`

mutapath.Path.startfile

Path.**startfile**()

Open this path in a platform-dependant manner. This method follows the best practice from Openstack.

See also:

`os.startfile()`

mutapath.Path.startswith

Path.**startswith**(*prefix*[, *start*[, *end*]]) → bool

Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

mutapath.Path.stat

Path.**stat**()

Perform a `stat()` system call on this path.

```
>>> Path('.').stat()
os.stat_result(...)
```

See also:

lstat(), `os.stat()`

mutapath.Path.statvfs

Path.**statvfs**()

Perform a `statvfs()` system call on this path.

See also:

`os.statvfs()`

mutapath.Path.strip

Path.**strip**(*chars=None*, */*)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

mutapath.Path.stripext

Path.**stripext**()

Remove one file extension from the path.

For example, `Path('/home/guido/python.tar.gz').stripext()` returns `Path('/home/guido/python.tar')`.

mutapath.Path.swapcase

Path.**swapcase**()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

mutapath.Path.symlink

Path.**symlink**(*newlink=None*)

Create a symbolic link at *newlink*, pointing here.

If newlink is not supplied, the symbolic link will assume the name self.basename(), creating the link in the cwd.

See also:

`os.symlink()`

mutapath.Path.symlink_to

Path.**symlink_to**(*target*, *target_is_directory=False*)

Make this path a symlink pointing to the target path. Note the order of arguments (link, target) is the reverse of os.symlink.

mutapath.Path.title

Path.**title**()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

mutapath.Path.touch

Path.**touch**()

Set the access/modified times of this file to the current time. Create the file if it does not exist.

mutapath.Path.translate

Path.**translate**(*table*, */*)

Replace each character in the string using the given translation table.

table
Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

mutapath.Path.unlink

Path.**unlink**()

Remove a file (same as unlink()).

If dir_fd is not None, it should be a file descriptor open to a directory,
and path should be relative; path will then be relative to that directory.

dir_fd may not be implemented on your platform.
If it is unavailable, using it will raise a NotImplementedError.

mutapath.Path.unlink_p

Path.**unlink_p**()

Like *remove()*, but does not raise an exception if the file does not exist.

mutapath.Path.upper

Path.**upper**()

Return a copy of the string converted to uppercase.

mutapath.Path.using_module

Path.**using_module**(*module*)

mutapath.Path.utime

Path.**utime**(**args*, ***kwargs*)

Set the access and modified times of this file.

See also:

os.utime()

mutapath.Path.walk

Path.**walk**(*match=None*, *errors='strict'*)

Iterator over files and subdirs, recursively.

The iterator yields Path objects naming each child item of this directory and its descendants. This requires that D.isdir().

This performs a depth-first traversal of the directory tree. Each directory is returned just before all its children.

The *errors=* keyword argument controls behavior when an error occurs. The default is 'strict', which causes an exception. Other allowed values are 'warn' (which reports the error via warnings.warn()), and 'ignore'. *errors* may also be an arbitrary callable taking a msg parameter.

mutapath.Path.walkdirs

Path.**walkdirs**(**args*, ***kwargs*)

Iterator over subdirs, recursively.

mutapath.Path.walkfiles

Path.**walkfiles**(**args*, ***kwargs*)

Iterator over files, recursively.

mutapath.Path.with_base

Path.**with_base**(*base*, *strip_length: int = 0*)

Clone this path with a new base.

The given path is used in its full length as base of this path, if strip_length is not specified.

Example

```
>>> Path('/home/doe/folder/sub').with_base("/home/joe")
Path('/home/joe/folder/sub')
```

If strip_length is specified, the given number of path elements are stripped from the left side, and the given base is prepended.

Example

```
>>> Path('/home/doe/folder/sub').with_base("/home/joe", strip_length=1)
Path('/home/joe/doe/folder/sub')
```

mutapath.Path.with_name

Path.**with_name**(*new_name*) → *Path*

See also:

`pathlib.PurePath.with_name()`

mutapath.Path.with_parent

Path.**with_parent**(*new_parent*) → *Path*

Clone this path with a new parent.

mutapath.Path.with_poxis_enabled

Path.**with_poxis_enabled**(*enable: bool = True*) → *Path*

Clone this path in posix format with posix-like separators (i.e., '/').

Example

```
>>> Path("\home\\doe/folder\sub").with_poxis_enabled()
Path('/home/joe/doe/folder/sub')
```

mutapath.Path.with_stem

Path.**with_stem**(*new_stem*) → *Path*

Clone this path with a new stem.

mutapath.Path.with_string_repr_enabled

Path.**with_string_repr_enabled**(*enable: bool = True*) → *Path*

Clone this path in with string representation enabled.

Example

```
>>> Path("/home/doe/folder/sub").with_string_repr_enabled()
'/home/joe/doe/folder/sub'
```

mutapath.Path.with_suffix

Path.**with_suffix**(*suffix*)

Return a new path with the file suffix changed (or added, if none)

```
>>> Path('/home/guido/python.tar.gz').with_suffix(".foo")
Path('/home/guido/python.tar.foo')
```

```
>>> Path('python').with_suffix('.zip')
Path('python.zip')
```

```
>>> Path('filename.ext').with_suffix('zip')
Traceback (most recent call last):
...
ValueError: Invalid suffix 'zip'
```

mutapath.Path.write_bytes

Path.**write_bytes**(*bytes*, *append=False*)

Open this file and write the given bytes to it.

Default behavior is to overwrite any existing file. Call `p.write_bytes(bytes, append=True)` to append instead.

mutapath.Path.write_lines

Path.**write_lines**(*lines*, *encoding=None*, *errors='strict'*, *linesep=<object object>*, *append=False*)

Write the given lines of text to this file.

By default this overwrites any existing file at this path.

This puts a platform-specific newline sequence on every line. See *linesep* below.

lines - A list of strings.

***encoding* - A Unicode encoding to use. This applies only if** *lines* contains any Unicode strings.

***errors* - How to handle errors in Unicode encoding. This** also applies only to Unicode strings.

linesep - (deprecated) The desired line-ending. This line-ending is applied to every line. If a line already has any standard line ending (`'\r'`, `'\n'`, `'\r\n'`, `u'\x85'`, `u'\r\x85'`, `u'\u2028'`), that will be stripped off and this will be used instead. The default is os.linesep, which is platform-dependent (`'\r\n'` on Windows, `'\n'` on Unix, etc.). Specify `None` to write the lines as-is, like `.writelines` on a file object.

Use the keyword argument `append=True` to append lines to the file. The default is to overwrite the file.

mutapath.Path.write_text

Path.**write_text**(*text*, *encoding=None*, *errors='strict'*, *linesep='\n'*, *append=False*)

Write the given text to this file.

The default behavior is to overwrite any existing file; to append instead, use the *append=True* keyword argument.

There are two differences between *write_text()* and *write_bytes()*: newline handling and Unicode handling. See below.

Parameters

- **written.** (*text - str/bytes - The text to be*) –
- **used.** (*encoding - str - The text encoding*) –
- **errors.** (*errors - str - How to handle Unicode encoding*) – Default is `'strict'`. See `help(unicode.encode)` for the options. Ignored if *text* isn't a Unicode string.
- **of** (*linesep - keyword argument - str/unicode - The sequence*) – characters to be used to mark end-of-line. The default is `os.linesep`. Specify `None` to use newlines unmodified.
- **if** (*append - keyword argument - bool - Specifies what to do*) – the file already exists (`True`: append to the end of it; `False`: overwrite it). The default is `False`.

— Newline handling.

`write_text()` converts all standard end-of-line sequences (`'\n'`, `'\r'`, and `'\r\n'`) to your platform's default end-of-line sequence (see `os.linesep`; on Windows, for example, the end-of-line marker is `'\r\n'`).

To override the platform's default, pass the *linesep=* keyword argument. To preserve the newlines as-is, pass `linesep=None`.

This handling applies to Unicode text and bytes, except with Unicode, additional non-ASCII newlines are recognized: `\x85`, `\r\x85`, and `\u2028`.

— Unicode

If *text* isn't Unicode, then apart from newline handling, the bytes are written verbatim to the file. The *encoding* and *errors* arguments are not used and must be omitted.

If *text* is Unicode, it is first converted to *bytes()* using the specified *encoding* (or the default encoding if *encoding* isn't specified). The *errors* argument applies only to this conversion.

mutapath.Path.zfill

`Path.`**`zfill`**`(`*width*`, /)`

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

Attributes

anchor	The concatenation of the drive and root, or ''.
atime	Last access time of the file.
base	Get the path base (i.e., the parent of the file).
bytes	Read the file as bytes stream and return its content.
ctime	Creation time of the file.
cwd	Return a new path pointing to the current working directory (as returned by os.getcwd()).
dirname	Returns the directory component of a pathname
drive	The drive specifier, for example `'C:'`.
ext	The file extension, for example `'.py'`.
home	Get the home path of the current path representation.
lock	Generate a cached file locker for this file with the additional suffix '.lock'.
mtime	Last modified time of the file.
name	The final path component, if any.
parent	The logical parent of the path.
parents	A sequence of this path's logical parents.
permissions	Permissions.
posix_enabled	If set to True, the the representation of this path will always follow the posix format, even on NT filesystems.
root	The root of the path, if any.
size	Size of the file, in bytes.
stem	The final path component, minus its last suffix.
string_repr_enabled	If set to True, the the representation of this path will always be returned unwrapped as the path's string.
suffix	The final component's last suffix, if any.
suffixes	A list of the final component's suffixes, if any.
text	Read the file as text stream and return its content.
to_pathlib	Return the contained path as pathlib.Path representation.

mutapath.Path.anchor

property Path.**anchor**

The concatenation of the drive and root, or ‘’.

mutapath.Path.atime

property Path.**atime**

Last access time of the file.

```
>>> Path('.').atime > 0
True
```

Allows setting:

```
>>> some_file = Path(getfixture('tmp_path')).joinpath('file.txt').touch()
>>> MST = datetime.timezone(datetime.timedelta(hours=-7))
>>> some_file.atime = datetime.datetime(1976, 5, 7, 10, tzinfo=MST)
>>> some_file.atime
200336400.0
```

See also:

getatime(), os.path.getatime()

mutapath.Path.base

property Path.**base:** *Path*

Get the path base (i.e., the parent of the file).

See also:

parent

mutapath.Path.bytes

Path.**bytes**

Read the file as bytes stream and return its content. This property caches the returned value. Clone this object to have a new path with a cleared cache or simply use read_bytes().

See also:

pathlib.Path.read_bytes()

mutapath.Path.ctime

property Path.**ctime**

Creation time of the file.

See also:

getctime(), os.path.getctime()

mutapath.Path.cwd

property Path.**cwd**

Return a new path pointing to the current working directory (as returned by os.getcwd()).

mutapath.Path.dirname

property Path.**dirname**: *Path*

Returns the directory component of a pathname

mutapath.Path.drive

property Path.**drive**

The drive specifier, for example `'C:'`.

This is always empty on systems that don't use drive specifiers.

mutapath.Path.ext

property Path.**ext**

The file extension, for example `'.py'`.

mutapath.Path.home

property Path.**home**: *Path*

Get the home path of the current path representation.

Returns
the home path

Example

```
>>> Path("/home/doe/folder/sub").home
Path("home")
```

mutapath.Path.lock

Path.**lock**

Generate a cached file locker for this file with the additional suffix '.lock'. If this path refers not to an existing file or to an existing folder, a dummy lock is returned that does not do anything.

Once this path is modified (cloning != modifying), the lock is released and regenerated for the new path.

Example

```
>>> my_path = Path('/home/doe/folder/sub')
>>> with my_path.lock:
...     my_path.write_text("I can write")
```

See also:

SoftFileLock, *DummyFileLock*

mutapath.Path.mtime

property Path.**mtime**

Last modified time of the file.

Allows setting:

```
>>> some_file = Path(getfixture('tmp_path')).joinpath('file.txt').touch()
>>> MST = datetime.timezone(datetime.timedelta(hours=-7))
>>> some_file.mtime = datetime.datetime(1976, 5, 7, 10, tzinfo=MST)
>>> some_file.mtime
200336400.0
```

See also:

getmtime(), os.path.getmtime()

mutapath.Path.name

property Path.**name**: *Path*

The final path component, if any.

mutapath.Path.parent

property Path.**parent**: *Path*

The logical parent of the path.

mutapath.Path.parents

property Path.**parents**

A sequence of this path's logical parents.

mutapath.Path.permissions

property Path.**permissions**

Permissions.

```
>>> perms = Path('.').permissions
>>> isinstance(perms, int)
True
>>> set(perms.symbolic) <= set('rwx-')
True
>>> perms.symbolic
'r...'
```

mutapath.Path.posix_enabled

property Path.**posix_enabled**: bool

If set to True, the the representation of this path will always follow the posix format, even on NT filesystems.

mutapath.Path.root

property Path.**root**

The root of the path, if any.

mutapath.Path.size

property Path.**size**

Size of the file, in bytes.

See also:

getsize(), os.path.getsize()

mutapath.Path.stem

property Path.**stem**: str

The final path component, minus its last suffix.

mutapath.Path.string_repr_enabled

property Path.**string_repr_enabled**: bool

If set to True, the the representation of this path will always be returned unwrapped as the path's string.

mutapath.Path.suffix

property Path.**suffix**: str

The final component's last suffix, if any.

This includes the leading period. For example: '.txt'

mutapath.Path.suffixes

property Path.**suffixes**

A list of the final component's suffixes, if any.

These include the leading periods. For example: ['.tar', '.gz']

mutapath.Path.text

Path.**text**

Read the file as text stream and return its content. This property caches the returned value. Clone this object to have a new path with a cleared cache or simply use `read_text()`.

See also:

`pathlib.Path.read_text()`

mutapath.Path.to_pathlib

property Path.**to_pathlib:** `Path`

Return the contained path as pathlib.Path representation. :return: the converted path

4.1.2 mutapath.MutaPath

class mutapath.**MutaPath**(*contained:* MutaPath | Path | *Path* | *PurePath* | *str* = '', *, *posix:* *bool* | *None* = *None*, *string_repr:* *bool* | *None* = *None*)

Bases: *Path*

Mutable Path

__init__(*contained:* MutaPath | Path | *Path* | *PurePath* | *str* = '', *, *posix:* *bool* | *None* = *None*, *string_repr:* *bool* | *None* = *None*)

Methods

absolute()	Return an absolute version of this path.
abspath()	Return an absolute path.
access(*args, **kwargs)	Return does the real user have access to this path.
as_posix()	Return the string representation of the path with forward (/) slashes.
as_uri()	Return the path as a 'file' URI.
basename()	**See also:** *name*, `os.path.basename()`
capitalize()	Return a capitalized version of the string.
casefold()	Return a version of the string suitable for caseless comparisons.
cd()	Change the current working directory to the specified path.
center(width[, fillchar])	Return a centered string of length width.
chdir()	Change the current working directory to the specified path.
chmod(mode)	Set the mode.
chown([uid, gid])	Change the owner and group by names or numbers.
chroot()	Change root directory to path.

continues on next page

Table 2 – continued from previous page

chunks(size, *args, **kwargs)	Returns a generator yielding chunks of the file, so it can
clone(contained)	Clone this path with a new given wrapped path representation, having the same remaining attributes.
copy(dst, *[, follow_symlinks])	Copy data and mode bits ("cp src dst").
copy2(dst, *[, follow_symlinks])	Copy data and metadata.
copyfile(dst, *[, follow_symlinks])	Copy data from src to dst in the most efficient way possible.
copying([lock, timeout, method])	Create a copying context for this immutable path.
copymode(dst, *[, follow_symlinks])	Copy mode bits from src to dst.
copystat(dst, *[, follow_symlinks])	Copy file metadata
copytree(dst[, symlinks, ignore, ...])	Recursively copy a directory tree and return the destination directory.
count(sub[, start[, end]])	Return the number of non-overlapping occurrences of substring sub in string S[start:end].
dirs(*args, **kwargs)	List of this directory's subdirectories.
encode([encoding, errors])	Encode the string using the codec registered for encoding.
endswith(suffix[, start[, end]])	Return True if S ends with the specified suffix, False otherwise.
exists()	Test whether a path exists.
expand()	Clean up a filename by calling *expandvars()*, *expanduser()*, and *normpath()* on it.
expandtabs([tabsize])	Return a copy where all tab characters are expanded using spaces.
expanduser()	Expand ~ and ~user constructions.
expandvars()	Expand shell variables of form $var and ${var}.
files(*args, **kwargs)	List of the files in self.
find(sub[, start[, end]])	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
fnmatch(pattern[, normcase])	Return `True` if *self.name* matches the given *pattern*.
format(*args, **kwargs)	Return a formatted version of S, using substitutions from args and kwargs.
format_map(mapping)	Return a formatted version of S, using substitutions from mapping.
get_owner()	Return the name of the owner of this file or directory.
getatime()	**See also:** *atime*, `os.path.getatime()`
getctime()	**See also:** *ctime*, `os.path.getctime()`
getcwd()	**See also:** `pathlib.Path.cwd()`
getmtime()	**See also:** *mtime*, `os.path.getmtime()`

continues on next page

Table 2 – continued from previous page

getsize()	**See also:** *size*, os.path.getsize()
glob(pattern)	**See also:** pathlib.Path.glob()
group()	Return the group name of the file gid.
iglob(pattern)	Return an iterator of Path objects that match the pattern.
in_place([mode, buffering, encoding, ...])	A context in which a file may be re-written in-place with new content.
index(sub[, start[, end]])	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
is_absolute()	True if the path is absolute (has both a root and, if applicable, a drive).
is_block_device()	Whether this path is a block device.
is_char_device()	Whether this path is a character device.
is_dir()	Whether this path is a directory.
is_fifo()	Whether this path is a FIFO.
is_file()	Whether this path is a regular file (also True for symlinks pointing to regular files).
is_mount()	Check if this path is a POSIX mount point
is_reserved()	Return True if the path contains one of the special names reserved by the system, if any.
is_socket()	Whether this path is a socket.
is_symlink()	Whether this path is a symbolic link.
isabs()	`>>> Path('.').isabs()`
isalnum()	Return True if the string is an alpha-numeric string, False otherwise.
isalpha()	Return True if the string is an alphabetic string, False otherwise.
isascii()	Return True if all characters in the string are ASCII, False otherwise.
isdecimal()	Return True if the string is a decimal string, False otherwise.
isdigit()	Return True if the string is a digit string, False otherwise.
isdir()	Return true if the pathname refers to an existing directory.
isfile()	Test whether a path is a regular file
isidentifier()	Return True if the string is a valid Python identifier, False otherwise.
islink()	Test whether a path is a symbolic link
islower()	Return True if the string is a lowercase string, False otherwise.

continues on next page

Table 2 – continued from previous page

ismount()	`>>> Path('.').ismount()`
isnumeric()	Return True if the string is a numeric string, False otherwise.
isprintable()	Return True if the string is printable, False otherwise.
isspace()	Return True if the string is a whitespace string, False otherwise.
istitle()	Return True if the string is a title-cased string, False otherwise.
isupper()	Return True if the string is an uppercase string, False otherwise.
iterdir()	Iterate over the files in this directory.
join(iterable, /)	Concatenate any number of strings.
joinpath(*others)	partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.
lchmod(mode)	Like chmod(), except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.
lines([encoding, errors, retain])	Open this file, read all lines, return them in a list.
link(newpath)	Create a hard link at *newpath*, pointing to this file.
link_to(target)	Make the target path a hard link pointing to this path.
listdir([match])	List of items in this directory.
ljust(width[, fillchar])	Return a left-justified string of length width.
lower()	Return a copy of the string converted to lowercase.
lstat()	Like *stat()*, but do not follow symbolic links.
lstrip([chars])	Return a copy of the string with leading whitespace removed.
makedirs(name [[, mode, exist_ok])	Super-mkdir; create a leaf directory and all intermediate ones.
makedirs_p([mode])	Like *makedirs()*, but does not raise an exception if the directory already exists.
match(path_pattern)	Return True if this path matches the given pattern.
merge_tree(other, *args, **kwargs)	Move, merge and mutate this path to the given other path.
mkdir([mode])	Create a directory.
mkdir_p([mode])	Like *mkdir()*, but does not raise an exception if the directory already exists.
move(dst[, copy_function])	Recursively move a file or directory to another location.
moving([lock, timeout, method])	Create a moving context for this immutable path.
mutate()	Create a mutable context for this immutable path.
normcase()	Normalize case of pathname.
normpath()	Normalize path, eliminating double slashes, etc.
open(*args, **kwargs)	Open file and return a stream.
partition(sep, /)	Partition the string into three parts using the given separator.
parts()	`>>> Path('/foo/bar/baz').parts()`

continues on next page

Table 2 – continued from previous page

pathconf(name)	Return the configuration limit name for the file or directory path.
posix_string()	Get this path as string with posix-like separators (i.e., '/').
read_bytes()	Return the contents of this file as bytes.
read_hash(hash_name)	Calculate given hash for this file.
read_hexhash(hash_name)	Calculate given hash for this file, returning hexdigest.
read_md5()	Calculate the md5 hash for this file.
read_text([encoding, errors])	Open this file, read it in, return the content as a string.
readlink()	Return the path to which this symbolic link points.
readlinkabs()	Return the path to which this symbolic link points.
realpath()	Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path.
relative_to(*other)	Return the relative path to another path identified by the passed arguments.
relpath([start])	Return this path as a relative path, based from *start*, which defaults to the current working directory.
relpathto(dest)	Return a relative path from *self* to *dest*.
remove()	Remove a file (same as unlink()).
remove_p()	Like *remove()*, but does not raise an exception if the file does not exist.
removedirs(name)	Super-rmdir; remove a leaf directory and all empty intermediate ones.
removedirs_p()	Like *removedirs()*, but does not raise an exception if the directory is not empty or does not exist.
rename(new)	Rename a file or directory.
renames(old, new)	Super-rename; create directories as necessary and delete any left empty.
renaming([lock, timeout, method])	Create a renaming context for this immutable path.
replace(old, new[, count])	Return a copy with all occurrences of substring old replaced by new.
resolve([strict])	Make the path absolute, resolving all symlinks on the way and also normalizing it (for example turning slashes into backslashes under Windows).
rfind(sub[, start[, end]])	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
rglob(pattern)	Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.
rindex(sub[, start[, end]])	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end].
rjust(width[, fillchar])	Return a right-justified string of length width.
rmdir()	Remove a directory.
rmdir_p()	Like *rmdir()*, but does not raise an exception if the directory is not empty or does not exist.
rmtree([ignore_errors, onerror])	Recursively delete a directory tree.
rmtree_p()	Like *rmtree()*, but does not raise an exception if the directory does not exist.
rpartition(sep, /)	Partition the string into three parts using the given separator.

continues on next page

Table 2 – continued from previous page

rsplit([sep, maxsplit])	Return a list of the words in the string, using sep as the delimiter string.
rstrip([chars])	Return a copy of the string with trailing whitespace removed.
samefile(other)	Test whether two pathnames reference the same actual file or directory
set_atime(value)	
set_mtime(value)	
split([sep, maxsplit])	Return a list of the words in the string, using sep as the delimiter string.
splitall()	Return a list of the path components in this path.
splitdrive()	Return two-tuple of `.drive` and rest without drive.
splitext()	Return two-tuple of `.stripext()` and `.ext`.
splitlines([keepends])	Return a list of the lines in the string, breaking at line boundaries.
splitpath()	Return two-tuple of `.parent`, `.name`.
startfile()	Open this path in a platform-dependant manner.
startswith(prefix[, start[, end]])	Return True if S starts with the specified prefix, False otherwise.
stat()	Perform a `stat()` system call on this path.
statvfs()	Perform a `statvfs()` system call on this path.
strip([chars])	Return a copy of the string with leading and trailing whitespace removed.
stripext()	Remove one file extension from the path.
swapcase()	Convert uppercase characters to lowercase and lowercase characters to uppercase.
symlink([newlink])	Create a symbolic link at *newlink*, pointing here.
symlink_to(target[, target_is_directory])	Make this path a symlink pointing to the target path.
title()	Return a version of the string where each word is titlecased.
touch()	Set the access/modified times of this file to the current time.
translate(table, /)	Replace each character in the string using the given translation table.
unlink()	Remove a file (same as unlink()).
unlink_p()	Like *remove()*, but does not raise an exception if the file does not exist.
upper()	Return a copy of the string converted to uppercase.
using_module(module)	
utime(*args, **kwargs)	Set the access and modified times of this file.
walk([match, errors])	Iterator over files and subdirs, recursively.
walkdirs(*args, **kwargs)	Iterator over subdirs, recursively.
walkfiles(*args, **kwargs)	Iterator over files, recursively.
with_base(base[, strip_length])	Clone this path with a new base.
with_name(new_name)	**See also:** `pathlib.PurePath.with_name()`
with_parent(new_parent)	Clone this path with a new parent.

continues on next page

Table 2 – continued from previous page

with_poxis_enabled([enable])	Clone this path in posix format with posix-like separators (i.e., '/').
with_stem(new_stem)	Clone this path with a new stem.
with_string_repr_enabled([enable])	Clone this path in with string representation enabled.
with_suffix(suffix)	Return a new path with the file suffix changed (or added, if none)
write_bytes(bytes[, append])	Open this file and write the given bytes to it.
write_lines(lines[, encoding, errors, ...])	Write the given lines of text to this file.
write_text(text[, encoding, errors, ...])	Write the given text to this file.
zfill(width, /)	Pad a numeric string with zeros on the left, to fill a field of the given width.

mutapath.MutaPath.absolute

MutaPath.**absolute**()

Return an absolute version of this path. This function works even if the path doesn't point to anything.

No normalization is done, i.e. all '.' and '..' will be kept along. Use resolve() to get the canonical path to a file.

mutapath.MutaPath.abspath

MutaPath.**abspath**()

Return an absolute path.

mutapath.MutaPath.access

MutaPath.**access**(**args*, ***kwargs*)

Return does the real user have access to this path.

```
>>> Path('.').access(os.F_OK)
True
```

See also:

`os.access()`

mutapath.MutaPath.as_posix

MutaPath.**as_posix**()

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

mutapath.MutaPath.as_uri

MutaPath.**as_uri**()

Return the path as a 'file' URI.

mutapath.MutaPath.basename

MutaPath.**basename**()

See also:

name, os.path.basename()

mutapath.MutaPath.capitalize

MutaPath.**capitalize**()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

mutapath.MutaPath.casefold

MutaPath.**casefold**()

Return a version of the string suitable for caseless comparisons.

mutapath.MutaPath.cd

MutaPath.**cd**()

Change the current working directory to the specified path.

path may always be specified as a string. On some platforms, path may also be specified as an open file descriptor.

If this functionality is unavailable, using it raises an exception.

mutapath.MutaPath.center

MutaPath.**center**(*width*, *fillchar=' '*, */*)

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

mutapath.MutaPath.chdir

MutaPath.**chdir**()

Change the current working directory to the specified path.

path may always be specified as a string. On some platforms, path may also be specified as an open file descriptor.

If this functionality is unavailable, using it raises an exception.

mutapath.MutaPath.chmod

MutaPath.**chmod**(*mode*)

Set the mode. May be the new mode (os.chmod behavior) or a symbolic mode.

```
>>> a_file = Path(getfixture('tmp_path')).joinpath('afile.txt').touch()
>>> a_file.chmod(0o700)
Path(...
>>> a_file.chmod('u+x')
Path(...
```

See also:

os.chmod()

mutapath.MutaPath.chown

MutaPath.**chown**(*uid=-1*, *gid=-1*)

Change the owner and group by names or numbers.

See also:

os.chown()

mutapath.MutaPath.chroot

MutaPath.**chroot**()

Change root directory to path.

mutapath.MutaPath.chunks

MutaPath.**chunks**(*size*, **args*, ***kwargs*)

Returns a generator yielding chunks of the file, so it can
be read piece by piece with a simple for loop.

Any argument you pass after *size* will be passed to *open()*.

Example

```
>>> hash = hashlib.md5()
>>> for chunk in Path("NEWS.rst").chunks(8192, mode='rb'):
...     hash.update(chunk)
```

This will read the file by chunks of 8192 bytes.

mutapath.MutaPath.clone

MutaPath.**clone**(*contained*) → *Path*

Clone this path with a new given wrapped path representation, having the same remaining attributes. :param contained: the new contained path element :return: the cloned path

mutapath.MutaPath.copy

MutaPath.**copy**(*dst*, *, *follow_symlinks=True*)

Copy data and mode bits ("cp src dst"). Return the file's destination.

The destination may be a directory.

If follow_symlinks is false, symlinks won't be followed. This resembles GNU's "cp -P src dst".

If source and destination are the same file, a SameFileError will be raised.

mutapath.MutaPath.copy2

MutaPath.**copy2**(*dst*, *, *follow_symlinks=True*)

Copy data and metadata. Return the file's destination.

Metadata is copied with copystat(). Please see the copystat function for more information.

The destination may be a directory.

If follow_symlinks is false, symlinks won't be followed. This resembles GNU's "cp -P src dst".

mutapath.MutaPath.copyfile

MutaPath.**copyfile**(*dst*, *, *follow_symlinks=True*)

Copy data from src to dst in the most efficient way possible.

If follow_symlinks is not set and src is a symbolic link, a new symlink will be created instead of copying the file it points to.

mutapath.MutaPath.copying

MutaPath.**copying**(*lock=True, timeout=1, method: ~typing.Callable[[~mutapath.immutapath.Path, ~mutapath.immutapath.Path], ~mutapath.immutapath.Path] = <function copy>*)

Create a copying context for this immutable path. The external value is only changed if the copying succeeds.

Parameters

- **timeout** – the timeout in seconds how long the lock file should be acquired
- **lock** – if the source file should be locked as long as this context is open
- **method** – an alternative method that copies the path and returns the new path (e.g., shutil.copy2)

Example

```
>>> with Path('/home/doe/folder/a.txt').copying() as mut:
...     mut.stem = "b"
Path('/home/doe/folder/b.txt')
```

mutapath.MutaPath.copymode

MutaPath.**copymode**(*dst*, *, *follow_symlinks=True*)

Copy mode bits from src to dst.

If follow_symlinks is not set, symlinks aren't followed if and only if both *src* and *dst* are symlinks. If *lchmod* isn't available (e.g. Linux) this method does nothing.

mutapath.MutaPath.copystat

MutaPath.**copystat**(*dst*, *, *follow_symlinks=True*)

Copy file metadata

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

If the optional flag *follow_symlinks* is not set, symlinks aren't followed if and only if both *src* and *dst* are symlinks.

mutapath.MutaPath.copytree

MutaPath.**copytree**(*dst*, *symlinks=False*, *ignore=None*, *copy_function=<function copy2>*, *ignore_dangling_symlinks=False*, *dirs_exist_ok=False*)

Recursively copy a directory tree and return the destination directory.

dirs_exist_ok dictates whether to raise an exception in case dst or any missing parent directory already exists.

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

If the optional symlinks flag is true, symbolic links in the source tree result in symbolic links in the destination tree; if it is false, the contents of the files pointed to by symbolic links are copied. If the file pointed by the symlink doesn't exist, an exception will be added in the list of errors raised in an Error exception at the end of the copy process.

You can set the optional ignore_dangling_symlinks flag to true if you want to silence this exception. Notice that this has no effect on platforms that don't support os.symlink.

The optional ignore argument is a callable. If given, it is called with the *src* parameter, which is the directory being visited by copytree(), and *names* which is the list of *src* contents, as returned by os.listdir():

> callable(src, names) -> ignored_names

Since copytree() is called recursively, the callable will be called once for each directory that is copied. It returns a list of names relative to the *src* directory that should not be copied.

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

mutapath.MutaPath.count

MutaPath.**count**(*sub*[, *start*[, *end*]]) → int

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

mutapath.MutaPath.dirs

MutaPath.**dirs**(**args*, ***kwargs*)

List of this directory's subdirectories.

The elements of the list are Path objects. This does not walk recursively into subdirectories (but see *walkdirs()*).

Accepts parameters to *listdir()*.

mutapath.MutaPath.encode

MutaPath.**encode**(*encoding='utf-8'*, *errors='strict'*)

Encode the string using the codec registered for encoding.

encoding
The encoding in which to encode the string.

errors
The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

mutapath.MutaPath.endswith

MutaPath.**endswith**(*suffix*[, *start*[, *end*]]) → bool

Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

mutapath.MutaPath.exists

MutaPath.**exists**()

Test whether a path exists. Returns False for broken symbolic links

mutapath.MutaPath.expand

MutaPath.**expand**()

Clean up a filename by calling *expandvars()*, *expanduser()*, and *normpath()* on it.

This is commonly everything needed to clean up a filename read from a configuration file, for example.

mutapath.MutaPath.expandtabs

MutaPath.**expandtabs**(*tabsize=8*)

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

mutapath.MutaPath.expanduser

MutaPath.**expanduser**()

Expand ~ and ~user constructions. If user or $HOME is unknown, do nothing.

mutapath.MutaPath.expandvars

MutaPath.**expandvars**()

Expand shell variables of form $var and ${var}. Unknown variables are left unchanged.

mutapath.MutaPath.files

MutaPath.**files**(**args*, ***kwargs*)

List of the files in self.

The elements of the list are Path objects. This does not walk into subdirectories (see *walkfiles()*).

Accepts parameters to *listdir()*.

mutapath.MutaPath.find

MutaPath.**find**(*sub*[, *start*[, *end*]]) → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

mutapath.MutaPath.fnmatch

MutaPath.**fnmatch**(*pattern*, *normcase=None*)

Return `True` if *self.name* matches the given *pattern*.

***pattern* - A filename pattern with wildcards,**
for example `'*.py'`. If the pattern contains a *normcase* attribute, it is applied to the name and path prior to comparison.

***normcase* - (optional) A function used to normalize the pattern and**
filename before matching. Defaults to normcase from `self.module`, `os.path.normcase()`.

See also:

`fnmatch.fnmatch()`

mutapath.MutaPath.format

MutaPath.**format**(**args*, ***kwargs*) → str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

mutapath.MutaPath.format_map

MutaPath.**format_map**(*mapping*) → str

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

mutapath.MutaPath.get_owner

MutaPath.**get_owner**()

Return the name of the owner of this file or directory. Follow symbolic links.

See also:

owner

mutapath.MutaPath.getatime

MutaPath.**getatime**()

See also:

atime, os.path.getatime()

mutapath.MutaPath.getctime

MutaPath.**getctime**()

See also:

ctime, os.path.getctime()

mutapath.MutaPath.getcwd

classmethod MutaPath.**getcwd**() → *Path*

See also:

pathlib.Path.cwd()

mutapath.MutaPath.getmtime

MutaPath.**getmtime**()

See also:

mtime, os.path.getmtime()

mutapath.MutaPath.getsize

MutaPath.**getsize**()

See also:

size, os.path.getsize()

mutapath.MutaPath.glob

MutaPath.**glob**(*pattern*) → Iterable[*Path*]

See also:

pathlib.Path.glob()

mutapath.MutaPath.group

MutaPath.**group**()

Return the group name of the file gid.

mutapath.MutaPath.iglob

MutaPath.**iglob**(*pattern*)

Return an iterator of Path objects that match the pattern.

pattern - a path relative to this directory, with wildcards.

For example, `Path('/users').iglob('*/bin/*')` returns an iterator of all the files users have in their `bin` directories.

See also:

glob.iglob()

Note: Glob is **not** recursive, even when using **. To do recursive globbing see *walk()*, *walkdirs()* or *walkfiles()*.

---

mutapath.MutaPath.in_place

MutaPath.**in_place**(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*, *backup_extension=None*)

A context in which a file may be re-written in-place with new content.

Yields a tuple of `(readable, writable)` file objects, where *writable* replaces *readable*.

If an exception occurs, the old file is restored, removing the written data.

Mode *must not* use `'w'`, `'a'`, or `'+'`; only read-only-modes are allowed. A `ValueError` is raised on invalid modes.

For example, to add line numbers to a file:

```
p = Path(filename)
assert p.isfile()
with p.in_place() as (reader, writer):
    for number, line in enumerate(reader, 1):
        writer.write('{0:3}: '.format(number)))
        writer.write(line)
```

Thereafter, the file at *filename* will have line numbers in it.

mutapath.MutaPath.index

MutaPath.**index**(*sub*[, *start*[, *end*]]) → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

mutapath.MutaPath.is_absolute

MutaPath.**is_absolute**()

True if the path is absolute (has both a root and, if applicable, a drive).

mutapath.MutaPath.is_block_device

MutaPath.**is_block_device**()

Whether this path is a block device.

mutapath.MutaPath.is_char_device

MutaPath.**is_char_device**()

Whether this path is a character device.

mutapath.MutaPath.is_dir

MutaPath.**is_dir**()

Whether this path is a directory.

mutapath.MutaPath.is_fifo

MutaPath.**is_fifo**()

Whether this path is a FIFO.

mutapath.MutaPath.is_file

MutaPath.**is_file**()

Whether this path is a regular file (also True for symlinks pointing to regular files).

mutapath.MutaPath.is_mount

MutaPath.**is_mount**()

Check if this path is a POSIX mount point

mutapath.MutaPath.is_reserved

MutaPath.**is_reserved**()

Return True if the path contains one of the special names reserved by the system, if any.

mutapath.MutaPath.is_socket

MutaPath.**is_socket**()

Whether this path is a socket.

mutapath.MutaPath.is_symlink

MutaPath.**is_symlink**()

Whether this path is a symbolic link.

mutapath.MutaPath.isabs

MutaPath.**isabs**()

```
>>> Path('.').isabs()
False
```

See also:

os.path.isabs()

mutapath.MutaPath.isalnum

MutaPath.**isalnum**()

Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

mutapath.MutaPath.isalpha

MutaPath.**isalpha**()

Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

mutapath.MutaPath.isascii

MutaPath.**isascii**()

Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

mutapath.MutaPath.isdecimal

MutaPath.**isdecimal**()

Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

mutapath.MutaPath.isdigit

MutaPath.**isdigit**()

Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

mutapath.MutaPath.isdir

MutaPath.**isdir**()

Return true if the pathname refers to an existing directory.

mutapath.MutaPath.isfile

MutaPath.**isfile**()

Test whether a path is a regular file

mutapath.MutaPath.isidentifier

MutaPath.**isidentifier**()

Return True if the string is a valid Python identifier, False otherwise.

Call keyword.iskeyword(s) to test whether string s is a reserved identifier, such as “def” or “class”.

mutapath.MutaPath.islink

MutaPath.**islink**()

Test whether a path is a symbolic link

mutapath.MutaPath.islower

MutaPath.**islower**()

Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

mutapath.MutaPath.ismount

MutaPath.**ismount**()

```
>>> Path('.').ismount()
False
```

See also:

`os.path.ismount()`

mutapath.MutaPath.isnumeric

MutaPath.**isnumeric**()

Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

mutapath.MutaPath.isprintable

MutaPath.**isprintable**()

Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

mutapath.MutaPath.isspace

MutaPath.**isspace**()

Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

mutapath.MutaPath.istitle

MutaPath.**istitle**()

Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

mutapath.MutaPath.isupper

MutaPath.**isupper**()

Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

mutapath.MutaPath.iterdir

MutaPath.**iterdir**()

Iterate over the files in this directory. Does not yield any result for the special paths '.' and '..'.

mutapath.MutaPath.join

MutaPath.**join**(*iterable*, /)

Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

mutapath.MutaPath.joinpath

MutaPath.**joinpath**(**others*)

partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.

mutapath.MutaPath.lchmod

MutaPath.**lchmod**(*mode*)

Like chmod(), except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.

mutapath.MutaPath.lines

MutaPath.**lines**(*encoding=None*, *errors=None*, *retain=True*)

Open this file, read all lines, return them in a list.

Optional arguments:

***encoding* - The Unicode encoding (or character set) of**
the file. The default is `None`, meaning use `locale.getpreferredencoding()`.

***errors* - How to handle Unicode errors; see**
open for the options. Default is `None` meaning "strict".

***retain* - If `True` (default), retain newline characters,**
but translate all newline characters to `\n`. If `False`, newline characters are omitted.

See also:

text()

mutapath.MutaPath.link

MutaPath.**link**(*newpath*)

Create a hard link at *newpath*, pointing to this file.

See also:

`os.link()`

mutapath.MutaPath.link_to

MutaPath.**link_to**(*target*)

Make the target path a hard link pointing to this path.

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

mutapath.MutaPath.listdir

MutaPath.**listdir**(*match=None*)

List of items in this directory.

Use *files()* or *dirs()* instead if you want a listing of just files or just subdirectories.

The elements of the list are Path objects.

With the optional *match* argument, a callable, only return items whose names match the given pattern.

See also:

files(), *dirs()*

mutapath.MutaPath.ljust

MutaPath.**ljust**(*width*, *fillchar=' '*, */*)

Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

mutapath.MutaPath.lower

MutaPath.**lower**()

Return a copy of the string converted to lowercase.

mutapath.MutaPath.lstat

MutaPath.**lstat**()

Like *stat()*, but do not follow symbolic links.

```
>>> Path('.').lstat() == Path('.').stat()
True
```

See also:

stat(), os.lstat()

mutapath.MutaPath.lstrip

MutaPath.**lstrip**(*chars=None*, */*)

Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

mutapath.MutaPath.makedirs

MutaPath.**makedirs**(*name [, mode=0o777][, exist_ok=False]*)

Super-mkdir; create a leaf directory and all intermediate ones. Works like mkdir, except that any intermediate path segment (not just the rightmost) will be created if it does not exist. If the target directory already exists, raise an OSError if exist_ok is False. Otherwise no exception is raised. This is recursive.

mutapath.MutaPath.makedirs_p

MutaPath.**makedirs_p**(*mode=511*)

Like *makedirs()*, but does not raise an exception if the directory already exists.

mutapath.MutaPath.match

MutaPath.**match**(*path_pattern*)

Return True if this path matches the given pattern.

mutapath.MutaPath.merge_tree

MutaPath.**merge_tree**(*other*, **args*, ***kwargs*)

Move, merge and mutate this path to the given other path.

mutapath.MutaPath.mkdir

MutaPath.**mkdir**(*mode=511*)

Create a directory.

If dir_fd is not None, it should be a file descriptor open to a directory,
and path should be relative; path will then be relative to that directory.

dir_fd may not be implemented on your platform.
If it is unavailable, using it will raise a NotImplementedError.

The mode argument is ignored on Windows.

mutapath.MutaPath.mkdir_p

MutaPath.**mkdir_p**(*mode=511*)

Like *mkdir()*, but does not raise an exception if the directory already exists.

mutapath.MutaPath.move

MutaPath.**move**(*dst*, *copy_function=<function copy2>*)

Recursively move a file or directory to another location. This is similar to the Unix "mv" command. Return the file or directory's destination.

If the destination is a directory or a symlink to a directory, the source is moved inside the directory. The destination path must not already exist.

If the destination already exists but is not a directory, it may be overwritten depending on os.rename() semantics.

If the destination is on our current filesystem, then rename() is used. Otherwise, src is copied to the destination and then removed. Symlinks are recreated under the new name if os.rename() fails because of cross filesystem renames.

The optional *copy_function* argument is a callable that will be used to copy the source or it will be delegated to *copytree*. By default, copy2() is used, but any function that supports the same signature (like copy()) can be used.

A lot more could be done here... A look at a mv.c shows a lot of the issues this implementation glosses over.

mutapath.MutaPath.moving

MutaPath.**moving**(*lock=True, timeout=1, method: ~typing.Callable[[~os.PathLike, ~os.PathLike], str] = <function move>*)

Create a moving context for this immutable path. The external value is only changed if the moving succeeds.

Parameters

- **timeout** – the timeout in seconds how long the lock file should be acquired
- **lock** – if the source file should be locked as long as this context is open
- **method** – an alternative method that moves the path and returns the new path

Example

```
>>> with Path('/home/doe/folder/a.txt').moving() as mut:
...     mut.stem = "b"
Path('/home/doe/folder/b.txt')
```

mutapath.MutaPath.mutate

MutaPath.**mutate**()

Create a mutable context for this immutable path.

Example

```
>>> with Path('/home/doe/folder/sub').mutate() as mut:
...     mut.name = "top"
Path('/home/doe/folder/top')
```

mutapath.MutaPath.normcase

MutaPath.**normcase**()

Normalize case of pathname. Has no effect under Posix

mutapath.MutaPath.normpath

MutaPath.**normpath**()

Normalize path, eliminating double slashes, etc.

mutapath.MutaPath.open

MutaPath.**open**(**args*, ***kwargs*)

Open file and return a stream. Raise OSError upon failure.

file is either a text or byte string giving the name (and the path if the file isn't in the current working directory) of the file to be opened or an integer file descriptor of the file to be wrapped. (If a file descriptor is given, it is closed when the returned I/O object is closed, unless closefd is set to False.)

mode is an optional string that specifies the mode in which the file is opened. It defaults to 'r' which means open for reading in text mode. Other common values are 'w' for writing (truncating the file if it already exists), 'x' for creating and writing to a new file, and 'a' for appending (which on some Unix systems, means that all writes append to the end of the file regardless of the current seek position). In text mode, if encoding is not specified the encoding used is platform dependent: locale.getpreferredencoding(False) is called to get the current locale encoding. (For reading and writing raw bytes use binary mode and leave encoding unspecified.) The available modes are:

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	create a new file and open it for writing
'a'	open for writing, appending to the end of the file if it exists
'b'	binary mode
't'	text mode (default)
'+'	open a disk file for updating (reading and writing)
'U'	universal newline mode (deprecated)

The default mode is 'rt' (open for reading text). For binary random access, the mode 'w+b' opens and truncates the file to 0 bytes, while 'r+b' opens the file without truncation. The 'x' mode implies 'w' and raises an *FileExistsError* if the file already exists.

Python distinguishes between files opened in binary and text modes, even when the underlying operating system doesn't. Files opened in binary mode (appending 'b' to the mode argument) return contents as bytes objects without any decoding. In text mode (the default, or when 't' is appended to the mode argument), the contents of the file are returned as strings, the bytes having been first decoded using a platform-dependent encoding or using the specified encoding if given.

'U' mode is deprecated and will raise an exception in future versions of Python. It has no effect in Python 3. Use newline to control universal newlines mode.

buffering is an optional integer used to set the buffering policy. Pass 0 to switch buffering off (only allowed in binary mode), 1 to select line buffering (only usable in text mode), and an integer > 1 to indicate the size of a fixed-size chunk buffer. When no buffering argument is given, the default buffering policy works as follows:

- Binary files are buffered in fixed-size chunks; the size of the buffer is chosen using a heuristic trying to determine the underlying device's "block size" and falling back on *io.DEFAULT_BUFFER_SIZE*. On many systems, the buffer will typically be 4096 or 8192 bytes long.

- "Interactive" text files (files for which isatty() returns True) use line buffering. Other text files use the policy described above for binary files.

encoding is the name of the encoding used to decode or encode the file. This should only be used in text mode. The default encoding is platform dependent, but any encoding supported by Python can be passed. See the codecs module for the list of supported encodings.

errors is an optional string that specifies how encoding errors are to be handled—this argument should not be used in binary mode. Pass 'strict' to raise a ValueError exception if there is an encoding error (the default of None has the same effect), or pass 'ignore' to ignore errors. (Note that ignoring encoding errors can lead to data loss.) See the documentation for codecs.register or run 'help(codecs.Codec)' for a list of the permitted encoding error strings.

newline controls how universal newlines works (it only applies to text mode). It can be None, '', 'n', 'r', and 'rn'. It works as follows:

- On input, if newline is None, universal newlines mode is enabled. Lines in the input can end in 'n', 'r', or 'rn', and these are translated into 'n' before being returned to the caller. If it is '', universal newline mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- On output, if newline is None, any 'n' characters written are translated to the system default line separator, os.linesep. If newline is '' or 'n', no translation takes place. If newline is any of the other legal values, any 'n' characters written are translated to the given string.

If closefd is False, the underlying file descriptor will be kept open when the file is closed. This does not work when a file name is given and must be True in that case.

A custom opener can be used by passing a callable as *opener*. The underlying file descriptor for the file object is then obtained by calling *opener* with (*file*, *flags*). *opener* must return an open file descriptor (passing os.open as *opener* results in functionality similar to passing None).

open() returns a file object whose type depends on the mode, and through which the standard file operations such as reading and writing are performed. When open() is used to open a file in a text mode ('w', 'r', 'wt', 'rt', etc.), it returns a TextIOWrapper. When used to open a file in a binary mode, the returned class varies: in read binary mode, it returns a BufferedReader; in write binary and append binary modes, it returns a BufferedWriter, and in read/write mode, it returns a BufferedRandom.

It is also possible to use a string or bytearray as a file for both reading and writing. For strings StringIO can be used like a file opened in a text mode, and for bytes a BytesIO can be used like a file opened in a binary mode.

mutapath.MutaPath.partition

MutaPath.**partition**(*sep*, /)

Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

mutapath.MutaPath.parts

MutaPath.**parts**()

```
>>> Path('/foo/bar/baz').parts()
(Path('/'), 'foo', 'bar', 'baz')
```

mutapath.MutaPath.pathconf

MutaPath.**pathconf**(*name*)

Return the configuration limit name for the file or directory path.

If there is no limit, return -1. On some platforms, path may also be specified as an open file descriptor.

If this functionality is unavailable, using it raises an exception.

mutapath.MutaPath.posix_string

MutaPath.**posix_string**() → str

Get this path as string with posix-like separators (i.e., '/').

Example

```
>>> Path("\home\\doe/folder\sub").with_poxis_enabled()
'/home/joe/doe/folder/sub'
```

mutapath.MutaPath.read_bytes

MutaPath.**read_bytes**()

Return the contents of this file as bytes.

mutapath.MutaPath.read_hash

MutaPath.**read_hash**(*hash_name*)

Calculate given hash for this file.

List of supported hashes can be obtained from `hashlib` package. This reads the entire file.

See also:

`hashlib.hash.digest()`

mutapath.MutaPath.read_hexhash

MutaPath.**read_hexhash**(*hash_name*)

Calculate given hash for this file, returning hexdigest.

List of supported hashes can be obtained from `hashlib` package. This reads the entire file.

See also:

`hashlib.hash.hexdigest()`

mutapath.MutaPath.read_md5

MutaPath.**read_md5**()

Calculate the md5 hash for this file.

This reads through the entire file.

See also:

read_hash()

mutapath.MutaPath.read_text

MutaPath.**read_text**(*encoding=None*, *errors=None*)

Open this file, read it in, return the content as a string.

Optional parameters are passed to *open()*.

See also:

lines()

mutapath.MutaPath.readlink

MutaPath.**readlink**()

Return the path to which this symbolic link points.

The result may be an absolute or a relative path.

See also:

readlinkabs(), os.readlink()

mutapath.MutaPath.readlinkabs

MutaPath.**readlinkabs**()

Return the path to which this symbolic link points.

The result is always an absolute path.

See also:

readlink(), os.readlink()

mutapath.MutaPath.realpath

MutaPath.**realpath**()

Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path.

mutapath.MutaPath.relative_to

MutaPath.**relative_to**(**other*)

Return the relative path to another path identified by the passed arguments. If the operation is not possible (because this is not a subpath of the other path), raise ValueError.

mutapath.MutaPath.relpath

MutaPath.**relpath**(*start='.'*)

Return this path as a relative path, based from *start*, which defaults to the current working directory.

mutapath.MutaPath.relpathto

MutaPath.**relpathto**(*dest*)

Return a relative path from *self* to *dest*.

If there is no relative path from *self* to *dest*, for example if they reside on different drives in Windows, then this returns `dest.abspath()`.

mutapath.MutaPath.remove

MutaPath.**remove**()

Remove a file (same as unlink()).

If dir_fd is not None, it should be a file descriptor open to a directory,
and path should be relative; path will then be relative to that directory.

dir_fd may not be implemented on your platform.
If it is unavailable, using it will raise a NotImplementedError.

mutapath.MutaPath.remove_p

MutaPath.**remove_p**()

Like *remove()*, but does not raise an exception if the file does not exist.

mutapath.MutaPath.removedirs

MutaPath.**removedirs**(*name*)

Super-rmdir; remove a leaf directory and all empty intermediate ones. Works like rmdir except that, if the leaf directory is successfully removed, directories corresponding to rightmost path segments will be pruned away until either the whole path is consumed or an error occurs. Errors during this latter phase are ignored – they generally mean that a directory was not empty.

mutapath.MutaPath.removedirs_p

MutaPath.**removedirs_p**()

Like *removedirs()*, but does not raise an exception if the directory is not empty or does not exist.

mutapath.MutaPath.rename

MutaPath.**rename**(*new*)

Rename a file or directory.

If either src_dir_fd or dst_dir_fd is not None, it should be a file
descriptor open to a directory, and the respective path string (src or dst) should be relative; the path will then be relative to that directory.

src_dir_fd and dst_dir_fd, may not be implemented on your platform.
If they are unavailable, using them will raise a NotImplementedError.

mutapath.MutaPath.renames

MutaPath.**renames**(*old*, *new*)

Super-rename; create directories as necessary and delete any left empty. Works like rename, except creation of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned until either the whole path is consumed or a nonempty directory is found.

Note: this function can fail with the new directory structure made if you lack permissions needed to unlink the leaf directory or file.

mutapath.MutaPath.renaming

MutaPath.**renaming**(*lock=True, timeout=1, method: ~typing.Callable[[str, str], None] = <built-in function rename>*)

Create a renaming context for this immutable path. The external value is only changed if the renaming succeeds.

Parameters

- **timeout** – the timeout in seconds how long the lock file should be acquired
- **lock** – if the source file should be locked as long as this context is open
- **method** – an alternative method that renames the path (e.g., os.renames)

Example

```
>>> with Path('/home/doe/folder/a.txt').renaming() as mut:
...     mut.stem = "b"
Path('/home/doe/folder/b.txt')
```

mutapath.MutaPath.replace

MutaPath.**replace**(*old*, *new*, *count=-1*, */*)

Return a copy with all occurrences of substring old replaced by new.

count
Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

mutapath.MutaPath.resolve

MutaPath.**resolve**(*strict=False*)

Make the path absolute, resolving all symlinks on the way and also normalizing it (for example turning slashes into backslashes under Windows).

mutapath.MutaPath.rfind

MutaPath.**rfind**(*sub*[, *start*[, *end*]]) → int

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

mutapath.MutaPath.rglob

MutaPath.**rglob**(*pattern*)

Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.

mutapath.MutaPath.rindex

MutaPath.**rindex**(*sub*[, *start*[, *end*]]) → int

Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

mutapath.MutaPath.rjust

MutaPath.**rjust**(*width*, *fillchar=' '*, */*)

Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

mutapath.MutaPath.rmdir

MutaPath.**rmdir**()

Remove a directory.

If dir_fd is not None, it should be a file descriptor open to a directory,
and path should be relative; path will then be relative to that directory.

dir_fd may not be implemented on your platform.
If it is unavailable, using it will raise a NotImplementedError.

mutapath.MutaPath.rmdir_p

MutaPath.**rmdir_p**()

Like *rmdir()*, but does not raise an exception if the directory is not empty or does not exist.

mutapath.MutaPath.rmtree

MutaPath.**rmtree**(*ignore_errors=False*, *onerror=None*)

Recursively delete a directory tree.

If ignore_errors is set, errors are ignored; otherwise, if onerror is set, it is called to handle the error with arguments (func, path, exc_info) where func is platform and implementation dependent; path is the argument to that function that caused it to fail; and exc_info is a tuple returned by sys.exc_info(). If ignore_errors is false and onerror is None, an exception is raised.

mutapath.MutaPath.rmtree_p

MutaPath.**rmtree_p**()

Like *rmtree()*, but does not raise an exception if the directory does not exist.

mutapath.MutaPath.rpartition

MutaPath.**rpartition**(*sep*, */*)

Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

mutapath.MutaPath.rsplit

MutaPath.**rsplit**(*sep=None*, *maxsplit=-1*)

Return a list of the words in the string, using sep as the delimiter string.

sep
The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit
Maximum number of splits to do. -1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

mutapath.MutaPath.rstrip

MutaPath.**rstrip**(*chars=None*, */*)

Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

mutapath.MutaPath.samefile

MutaPath.**samefile**(*other*)

Test whether two pathnames reference the same actual 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.

mutapath.MutaPath.set_atime

MutaPath.**set_atime**(*value*)

mutapath.MutaPath.set_mtime

MutaPath.**set_mtime**(*value*)

mutapath.MutaPath.split

MutaPath.**split**(*sep=None*, *maxsplit=-1*)

Return a list of the words in the string, using sep as the delimiter string.

sep
: The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit
: Maximum number of splits to do. -1 (the default value) means no limit.

mutapath.MutaPath.splitall

MutaPath.**splitall**()

Return a list of the path components in this path.

The first item in the list will be a Path. Its value will be either `os.curdir`, `os.pardir`, empty, or the root directory of this path (for example, `'/'` or `'C:\\'`). The other items in the list will be strings.

`Path.joinpath(*result)` will yield the original path.

```
>>> Path('/foo/bar/baz').splitall()
[Path('/'), 'foo', 'bar', 'baz']
```

mutapath.MutaPath.splitdrive

MutaPath.**splitdrive**()

Return two-tuple of `.drive` and rest without drive.

Split the drive specifier from this path. If there is no drive specifier, *`p.drive`* is empty, so the return value is simply `(Path(''), p)`. This is always the case on Unix.

See also:

`os.path.splitdrive()`

mutapath.MutaPath.splitext

MutaPath.**splitext**()

Return two-tuple of `.stripext()` and `.ext`.

Split the filename extension from this path and return the two parts. Either part may be empty.

The extension is everything from `'.'` to the end of the last path segment. This has the property that if `(a, b) == p.splitext()`, then `a + b == p`.

See also:

`os.path.splitext()`

mutapath.MutaPath.splitlines

MutaPath.**splitlines**(*keepends=False*)

Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

mutapath.MutaPath.splitpath

MutaPath.**splitpath**()

Return two-tuple of `.parent`, `.name`.

See also:

`parent`, *`name`*, `os.path.split()`

mutapath.MutaPath.startfile

MutaPath.**startfile**()

Open this path in a platform-dependant manner. This method follows the best practice from Openstack.

See also:

`os.startfile()`

mutapath.MutaPath.startswith

MutaPath.**startswith**(*prefix*[, *start*[, *end*]]) → bool

Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

mutapath.MutaPath.stat

MutaPath.**stat**()

Perform a `stat()` system call on this path.

```
>>> Path('.').stat()
os.stat_result(...)
```

See also:

lstat(), `os.stat()`

mutapath.MutaPath.statvfs

MutaPath.**statvfs**()

Perform a `statvfs()` system call on this path.

See also:

`os.statvfs()`

mutapath.MutaPath.strip

MutaPath.**strip**(*chars=None*, */*)

Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

mutapath.MutaPath.stripext

MutaPath.**stripext**()

Remove one file extension from the path.

For example, `Path('/home/guido/python.tar.gz').stripext()` returns `Path('/home/guido/python.tar')`.

mutapath.MutaPath.swapcase

MutaPath.**swapcase**()

Convert uppercase characters to lowercase and lowercase characters to uppercase.

mutapath.MutaPath.symlink

MutaPath.**symlink**(*newlink=None*)

Create a symbolic link at *newlink*, pointing here.

If newlink is not supplied, the symbolic link will assume the name self.basename(), creating the link in the cwd.

See also:

os.symlink()

mutapath.MutaPath.symlink_to

MutaPath.**symlink_to**(*target*, *target_is_directory=False*)

Make this path a symlink pointing to the target path. Note the order of arguments (link, target) is the reverse of os.symlink.

mutapath.MutaPath.title

MutaPath.**title**()

Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

mutapath.MutaPath.touch

MutaPath.**touch**()

Set the access/modified times of this file to the current time. Create the file if it does not exist.

mutapath.MutaPath.translate

MutaPath.**translate**(*table*, */*)

Replace each character in the string using the given translation table.

table
Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

mutapath.MutaPath.unlink

MutaPath.**unlink**()

Remove a file (same as unlink()).

If dir_fd is not None, it should be a file descriptor open to a directory,
and path should be relative; path will then be relative to that directory.

dir_fd may not be implemented on your platform.
If it is unavailable, using it will raise a NotImplementedError.

mutapath.MutaPath.unlink_p

MutaPath.**unlink_p**()

Like *remove()*, but does not raise an exception if the file does not exist.

mutapath.MutaPath.upper

MutaPath.**upper**()

Return a copy of the string converted to uppercase.

mutapath.MutaPath.using_module

MutaPath.**using_module**(*module*)

mutapath.MutaPath.utime

MutaPath.**utime**(**args*, ***kwargs*)

Set the access and modified times of this file.

See also:

os.utime()

mutapath.MutaPath.walk

MutaPath.**walk**(*match=None*, *errors='strict'*)

Iterator over files and subdirs, recursively.

The iterator yields Path objects naming each child item of this directory and its descendants. This requires that D.isdir().

This performs a depth-first traversal of the directory tree. Each directory is returned just before all its children.

The *errors=* keyword argument controls behavior when an error occurs. The default is 'strict', which causes an exception. Other allowed values are 'warn' (which reports the error via warnings.warn()), and 'ignore'. *errors* may also be an arbitrary callable taking a msg parameter.

mutapath.MutaPath.walkdirs

MutaPath.**walkdirs**(**args*, ***kwargs*)

Iterator over subdirs, recursively.

mutapath.MutaPath.walkfiles

MutaPath.**walkfiles**(**args*, ***kwargs*)

Iterator over files, recursively.

mutapath.MutaPath.with_base

MutaPath.**with_base**(*base*, *strip_length: int = 0*)

Clone this path with a new base.

The given path is used in its full length as base of this path, if strip_length is not specified.

Example

```
>>> Path('/home/doe/folder/sub').with_base("/home/joe")
Path('/home/joe/folder/sub')
```

If strip_length is specified, the given number of path elements are stripped from the left side, and the given base is prepended.

Example

```
>>> Path('/home/doe/folder/sub').with_base("/home/joe", strip_length=1)
Path('/home/joe/doe/folder/sub')
```

mutapath.MutaPath.with_name

MutaPath.**with_name**(*new_name*) → *Path*

See also:

`pathlib.PurePath.with_name()`

mutapath.MutaPath.with_parent

MutaPath.**with_parent**(*new_parent*) → *Path*

Clone this path with a new parent.

mutapath.MutaPath.with_poxis_enabled

MutaPath.**with_poxis_enabled**(*enable: bool = True*) → *Path*

Clone this path in posix format with posix-like separators (i.e., '/').

Example

```
>>> Path("\home\\doe/folder\sub").with_poxis_enabled()
Path('/home/joe/doe/folder/sub')
```

mutapath.MutaPath.with_stem

MutaPath.**with_stem**(*new_stem*) → *Path*

Clone this path with a new stem.

mutapath.MutaPath.with_string_repr_enabled

MutaPath.**with_string_repr_enabled**(*enable: bool = True*) → *Path*

Clone this path in with string representation enabled.

Example

```
>>> Path("/home/doe/folder/sub").with_string_repr_enabled()
'/home/joe/doe/folder/sub'
```

mutapath.MutaPath.with_suffix

MutaPath.**with_suffix**(*suffix*)

Return a new path with the file suffix changed (or added, if none)

```
>>> Path('/home/guido/python.tar.gz').with_suffix(".foo")
Path('/home/guido/python.tar.foo')
```

```
>>> Path('python').with_suffix('.zip')
Path('python.zip')
```

```
>>> Path('filename.ext').with_suffix('zip')
Traceback (most recent call last):
...
ValueError: Invalid suffix 'zip'
```

mutapath.MutaPath.write_bytes

MutaPath.**write_bytes**(*bytes*, *append=False*)

Open this file and write the given bytes to it.

Default behavior is to overwrite any existing file. Call `p.write_bytes(bytes, append=True)` to append instead.

mutapath.MutaPath.write_lines

MutaPath.**write_lines**(*lines*, *encoding=None*, *errors='strict'*, *linesep=<object object>*, *append=False*)

Write the given lines of text to this file.

By default this overwrites any existing file at this path.

This puts a platform-specific newline sequence on every line. See *linesep* below.

lines - A list of strings.

***encoding* - A Unicode encoding to use. This applies only if** *lines* contains any Unicode strings.

***errors* - How to handle errors in Unicode encoding. This** also applies only to Unicode strings.

linesep - (deprecated) The desired line-ending. This line-ending is applied to every line. If a line already has any standard line ending (`'\r'`, `'\n'`, `'\r\n'`, `u'\x85'`, `u'\r\x85'`, `u'\u2028'`), that will be stripped off and this will be used instead. The default is os.linesep, which is platform-dependent (`'\r\n'` on Windows, `'\n'` on Unix, etc.). Specify `None` to write the lines as-is, like `.writelines` on a file object.

Use the keyword argument `append=True` to append lines to the file. The default is to overwrite the file.

mutapath.MutaPath.write_text

MutaPath.**write_text**(*text*, *encoding=None*, *errors='strict'*, *linesep='\n'*, *append=False*)

Write the given text to this file.

The default behavior is to overwrite any existing file; to append instead, use the *append=True* keyword argument.

There are two differences between *write_text()* and *write_bytes()*: newline handling and Unicode handling. See below.

Parameters

- **written.** (*text - str/bytes - The text to be*) –
- **used.** (*encoding - str - The text encoding*) –
- **errors.** (*errors - str - How to handle Unicode encoding*) – Default is `'strict'`. See `help(unicode.encode)` for the options. Ignored if *text* isn't a Unicode string.
- **of** (*linesep - keyword argument - str/unicode - The sequence*) – characters to be used to mark end-of-line. The default is `os.linesep`. Specify `None` to use newlines unmodified.
- **if** (*append - keyword argument - bool - Specifies what to do*) – the file already exists (`True`: append to the end of it; `False`: overwrite it). The default is `False`.

— Newline handling.

`write_text()` converts all standard end-of-line sequences (`'\n'`, `'\r'`, and `'\r\n'`) to your platform's default end-of-line sequence (see `os.linesep`; on Windows, for example, the end-of-line marker is `'\r\n'`).

To override the platform's default, pass the *linesep=* keyword argument. To preserve the newlines as-is, pass `linesep=None`.

This handling applies to Unicode text and bytes, except with Unicode, additional non-ASCII newlines are recognized: `\x85`, `\r\x85`, and `\u2028`.

— Unicode

If *text* isn't Unicode, then apart from newline handling, the bytes are written verbatim to the file. The *encoding* and *errors* arguments are not used and must be omitted.

If *text* is Unicode, it is first converted to *bytes()* using the specified *encoding* (or the default encoding if *encoding* isn't specified). The *errors* argument applies only to this conversion.

mutapath.MutaPath.zfill

MutaPath.**zfill**(*width*, /)

Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

Attributes

anchor	The concatenation of the drive and root, or ''.
atime	Last access time of the file.
base	Get the path base (i.e., the parent of the file).
bytes	Read the file as bytes stream and return its content.
ctime	Creation time of the file.
cwd	Return a new path pointing to the current working directory (as returned by os.getcwd()).
dirname	Returns the directory component of a pathname
drive	The drive specifier, for example `'C:'`.
ext	The file extension, for example `'.py'`.
home	Get the home path of the current path representation.
lock	Generate a cached file locker for this file with the additional suffix '.lock'.
mtime	Last modified time of the file.
name	The final path component, if any.
parent	The logical parent of the path.
parents	A sequence of this path's logical parents.
permissions	Permissions.
posix_enabled	If set to True, the the representation of this path will always follow the posix format, even on NT filesystems.
root	The root of the path, if any.
size	Size of the file, in bytes.
stem	The final path component, minus its last suffix.
string_repr_enabled	If set to True, the the representation of this path will always be returned unwrapped as the path's string.
suffix	The final component's last suffix, if any.
suffixes	A list of the final component's suffixes, if any.
text	Read the file as text stream and return its content.
to_pathlib	Return the contained path as pathlib.Path representation.

mutapath.MutaPath.anchor

property MutaPath.**anchor**

The concatenation of the drive and root, or ‘’.

mutapath.MutaPath.atime

property MutaPath.**atime**

Last access time of the file.

```
>>> Path('.').atime > 0
True
```

Allows setting:

```
>>> some_file = Path(getfixture('tmp_path')).joinpath('file.txt').touch()
>>> MST = datetime.timezone(datetime.timedelta(hours=-7))
>>> some_file.atime = datetime.datetime(1976, 5, 7, 10, tzinfo=MST)
>>> some_file.atime
200336400.0
```

See also:

getatime(), os.path.getatime()

mutapath.MutaPath.base

property MutaPath.**base:** *Path*

Get the path base (i.e., the parent of the file).

See also:

parent

mutapath.MutaPath.bytes

MutaPath.**bytes**

Read the file as bytes stream and return its content. This property caches the returned value. Clone this object to have a new path with a cleared cache or simply use read_bytes().

See also:

pathlib.Path.read_bytes()

mutapath.MutaPath.ctime

property MutaPath.**ctime**

Creation time of the file.

See also:

getctime(), os.path.getctime()

mutapath.MutaPath.cwd

property MutaPath.**cwd**

Return a new path pointing to the current working directory (as returned by os.getcwd()).

mutapath.MutaPath.dirname

property MutaPath.**dirname:** *Path*

Returns the directory component of a pathname

mutapath.MutaPath.drive

property MutaPath.**drive**

The drive specifier, for example `'C:'`.

This is always empty on systems that don't use drive specifiers.

mutapath.MutaPath.ext

property MutaPath.**ext**

The file extension, for example `'.py'`.

mutapath.MutaPath.home

property MutaPath.**home:** *Path*

Get the home path of the current path representation.

Returns
the home path

Example

```
>>> Path("/home/doe/folder/sub").home
Path("home")
```

mutapath.MutaPath.lock

MutaPath.**lock**

Generate a cached file locker for this file with the additional suffix '.lock'. If this path refers not to an existing file or to an existing folder, a dummy lock is returned that does not do anything.

Once this path is modified (cloning != modifying), the lock is released and regenerated for the new path.

Example

```
>>> my_path = Path('/home/doe/folder/sub')
>>> with my_path.lock:
...     my_path.write_text("I can write")
```

See also:

SoftFileLock, *DummyFileLock*

mutapath.MutaPath.mtime

property MutaPath.**mtime**

Last modified time of the file.

Allows setting:

```
>>> some_file = Path(getfixture('tmp_path')).joinpath('file.txt').touch()
>>> MST = datetime.timezone(datetime.timedelta(hours=-7))
>>> some_file.mtime = datetime.datetime(1976, 5, 7, 10, tzinfo=MST)
>>> some_file.mtime
200336400.0
```

See also:

getmtime(), os.path.getmtime()

mutapath.MutaPath.name

property MutaPath.**name:** *Path*

The final path component, if any.

mutapath.MutaPath.parent

property MutaPath.**parent:** *Path*

The logical parent of the path.

mutapath.MutaPath.parents

property MutaPath.**parents**

A sequence of this path's logical parents.

mutapath.MutaPath.permissions

property MutaPath.**permissions**

Permissions.

```
>>> perms = Path('.').permissions
>>> isinstance(perms, int)
True
>>> set(perms.symbolic) <= set('rwx-')
True
>>> perms.symbolic
'r...'
```

mutapath.MutaPath.posix_enabled

property MutaPath.**posix_enabled**: bool

If set to True, the the representation of this path will always follow the posix format, even on NT filesystems.

mutapath.MutaPath.root

property MutaPath.**root**

The root of the path, if any.

mutapath.MutaPath.size

property MutaPath.**size**

Size of the file, in bytes.

See also:

getsize(), os.path.getsize()

mutapath.MutaPath.stem

property MutaPath.**stem**: str

The final path component, minus its last suffix.

mutapath.MutaPath.string_repr_enabled

property MutaPath.**string_repr_enabled**: bool

If set to True, the the representation of this path will always be returned unwrapped as the path's string.

mutapath.MutaPath.suffix

property MutaPath.**suffix**: str

The final component's last suffix, if any.

This includes the leading period. For example: '.txt'

mutapath.MutaPath.suffixes

property MutaPath.**suffixes**

A list of the final component's suffixes, if any.

These include the leading periods. For example: ['.tar', '.gz']

mutapath.MutaPath.text

MutaPath.**text**

Read the file as text stream and return its content. This property caches the returned value. Clone this object to have a new path with a cleared cache or simply use `read_text()`.

See also:

`pathlib.Path.read_text()`

mutapath.MutaPath.to_pathlib

property MutaPath.**to_pathlib**: `Path`

Return the contained path as pathlib.Path representation. :return: the converted path

4.1.3 mutapath.exceptions.PathException

class mutapath.exceptions.**PathException**

Bases: `BaseException`

Exception about inconsistencies between the virtual path and the real file system.

__init__(**args*, ***kwargs*)

4.1.4 mutapath.lock_dummy.DummyFileLock

class mutapath.lock_dummy.**DummyFileLock**(*lock_file: str | os.PathLike[Any]*, *timeout: float = -1*, *mode: int = 420*, *thread_local: bool = True*)

Bases: `BaseFileLock`

Create a new lock object.

Parameters

- **lock_file** – path to the file
- **timeout** – default timeout when acquiring the lock, in seconds. It will be used as fallback value in

the acquire method, if no timeout value (`None`) is given. If you want to disable the timeout, set it to a negative value. A timeout of 0 means, that there is exactly one attempt to acquire the file lock. :param mode: file permissions for the lockfile. :param thread_local: Whether this object's internal context should be thread local or not. If this is set to `False` then the lock will be reentrant across threads.

__init__(*lock_file: str | os.PathLike[Any]*, *timeout: float = -1*, *mode: int = 420*, *thread_local: bool = True*) → None

Create a new lock object.

Parameters

- **lock_file** – path to the file
- **timeout** – default timeout when acquiring the lock, in seconds. It will be used as fallback value in

the acquire method, if no timeout value (`None`) is given. If you want to disable the timeout, set it to a negative value. A timeout of 0 means, that there is exactly one attempt to acquire the file lock. :param mode: file permissions for the lockfile. :param thread_local: Whether this object's internal context should be thread local or not. If this is set to `False` then the lock will be reentrant across threads.

Methods

acquire([timeout, poll_intervall])	Doing nothing
is_thread_local()	**return** a flag indicating if this lock is thread local or not
release([force])	Doing nothing

mutapath.lock_dummy.DummyFileLock.acquire

DummyFileLock.**acquire**(*timeout=None*, *poll_intervall=0.05*)

Doing nothing

mutapath.lock_dummy.DummyFileLock.is_thread_local

DummyFileLock.**is_thread_local**() → bool

Returns
a flag indicating if this lock is thread local or not

mutapath.lock_dummy.DummyFileLock.release

DummyFileLock.**release**(*force=False*)

Doing nothing

Attributes

is_locked	A boolean indicating if the lock file is holding the lock currently.
lock_counter	The number of times this lock has been acquired (but not yet released).
lock_file	path to the lock file
timeout	the default timeout value, in seconds

mutapath.lock_dummy.DummyFileLock.is_locked

property DummyFileLock.**is_locked**: bool

A boolean indicating if the lock file is holding the lock currently.

Changed in version 2.0.0: This was previously a method and is now a property.

Type
return

mutapath.lock_dummy.DummyFileLock.lock_counter

property DummyFileLock.**lock_counter**: int

The number of times this lock has been acquired (but not yet released).

Type
return

mutapath.lock_dummy.DummyFileLock.lock_file

property DummyFileLock.**lock_file**: str

path to the lock file

Type
return

mutapath.lock_dummy.DummyFileLock.timeout

property DummyFileLock.**timeout**: float

the default timeout value, in seconds

New in version 2.0.0.

Type
return

4.2 Indices and tables

- genindex

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W

Z