

---

# mutapath

'matfax'

Oct 26, 2019



# CONTENTS

<b>1</b>	<b>MutaPath Class</b>	<b>3</b>
<b>2</b>	<b>Path Class</b>	<b>5</b>
<b>3</b>	<b>Hashing</b>	<b>7</b>
3.1	Documentation . . . . .	7
3.2	Indices and tables . . . . .	86
	<b>Index</b>	<b>87</b>



This library is for you if you are also annoyed that there is no mutable pathlib wrapper for use cases where paths are often changed. mutapath solves this by wrapping the extended pathlib library [path.py](#) and updating the encapsulated object every time the path might be changed.

mutapath also adds the possibility to delimit file and path modifications to a safe fallback context.



## MUTAPATH CLASS

The MutaPath Class allows direct manipulation 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
```





## 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.



## 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
```

### 3.1 Documentation

<code>Path([contained, posix, string_repr])</code>	Immutable Path
<code>MutaPath([contained, posix, string_repr])</code>	Mutable Path
<code>PathException</code>	Exception about inconsistencies between the virtual path and the real file system.
<code>DummyFileLock(lock_file[, timeout])</code>	

#### 3.1.1 mutapath.Path

**class** `mutapath.Path` (*contained: Union[mutapath.immutapath.Path, path.Path, pathlib.PurePath, str]*  
= ", \*, posix: Optional[bool] = None, string\_repr: Optional[bool] = None)

Bases: `mashumaro.types.SerializableType`

Immutable Path

\_\_init\_\_ (*contained: Union[mutapath.immutapath.Path, path.Path, pathlib.PurePath, str] = ", \*,  
posix: Optional[bool] = None, string\_repr: Optional[bool] = None*)  
Initialize self. See help(type(self)) for accurate signature.

#### Methods

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

Continued on next page

Table 2 – continued from previous page

<code>format(*args, **kwargs)</code>	Return a formatted version of S, using substitutions from args and kwargs.
<code>format_map(mapping)</code>	Return a formatted version of S, using substitutions from mapping.
<code>get_owner()</code>	Return the name of the owner of this file or directory.
<code>getatime()</code>	<b>See also:</b> <code>atime, os.path.getatime()</code>
<code>getctime()</code>	<b>See also:</b> <code>ctime, os.path.getctime()</code>
<code>getcwd()</code>	<b>See also:</b> <code>pathlib.Path.cwd()</code>
<code>getmtime()</code>	<b>See also:</b> <code>mtime, os.path.getmtime()</code>
<code>getsize()</code>	<b>See also:</b> <code>size, os.path.getsize()</code>
<code>glob(pattern)</code>	Iterate over this subtree and yield all existing files (of any kind, including directories) matching the given relative pattern.
<code>group()</code>	Return the group name of the file gid.
<code>iglob(pattern)</code>	Return an iterator of Path objects that match the pattern.
<code>in_place([mode, buffering, encoding, ...])</code>	A context in which a file may be re-written in-place with new content.
<code>index(sub[, start[, end]])</code>	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end].
<code>is_absolute()</code>	True if the path is absolute (has both a root and, if applicable, a drive).
<code>is_block_device()</code>	Whether this path is a block device.
<code>is_char_device()</code>	Whether this path is a character device.
<code>is_dir()</code>	Whether this path is a directory.
<code>is_fifo()</code>	Whether this path is a FIFO.
<code>is_file()</code>	Whether this path is a regular file (also True for sym-links pointing to regular files).
<code>is_mount()</code>	Check if this path is a POSIX mount point
<code>is_reserved()</code>	Return True if the path contains one of the special names reserved by the system, if any.
<code>is_socket()</code>	Whether this path is a socket.
<code>is_symlink()</code>	Whether this path is a symbolic link.
<code>isabs()</code>	Test whether a path is absolute
<code>isalnum()</code>	Return True if the string is an alpha-numeric string, False otherwise.
<code>isalpha()</code>	Return True if the string is an alphabetic string, False otherwise.
<code>isascii()</code>	Return True if all characters in the string are ASCII, False otherwise.

Continued on next page

Table 2 – continued from previous page

<code>isdecimal()</code>	Return True if the string is a decimal string, False otherwise.
<code>isdigit()</code>	Return True if the string is a digit string, False otherwise.
<code>isdir()</code>	Return true if the pathname refers to an existing directory.
<code>isfile()</code>	Test whether a path is a regular file
<code>isidentifier()</code>	Return True if the string is a valid Python identifier, False otherwise.
<code>islink()</code>	Test whether a path is a symbolic link
<code>islower()</code>	Return True if the string is a lowercase string, False otherwise.
<code>ismount()</code>	Test whether a path is a mount point
<code>isnumeric()</code>	Return True if the string is a numeric string, False otherwise.
<code>isprintable()</code>	Return True if the string is printable, False otherwise.
<code>isspace()</code>	Return True if the string is a whitespace string, False otherwise.
<code>istitle()</code>	Return True if the string is a title-cased string, False otherwise.
<code>isupper()</code>	Return True if the string is an uppercase string, False otherwise.
<code>iterdir()</code>	Iterate over the files in this directory.
<code>join(iterable)</code>	Concatenate any number of strings.
<code>lchmod(mode)</code>	Like <code>chmod()</code> , except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.
<code>lines([encoding, errors, retain])</code>	Open this file, read all lines, return them in a list.
<code>link(newpath)</code>	Create a hard link at <i>newpath</i> , pointing to this file.
<code>listdir()</code>	Use <code>files()</code> or <code>dirs()</code> instead if you want a listing of just files or just subdirectories.
<code>ljust(width[, fillchar])</code>	Return a left-justified string of length width.
<code>lower()</code>	Return a copy of the string converted to lowercase.
<code>lstat()</code>	Like <code>stat()</code> , but do not follow symbolic links.
<code>lstrip([chars])</code>	Return a copy of the string with leading whitespace removed.
<code>makedirs(name [, mode, exist_ok])</code>	Super-mkdir; create a leaf directory and all intermediate ones.
<code>makedirs_p([mode])</code>	Like <code>makedirs()</code> , but does not raise an exception if the directory already exists.
<code>match(path_pattern)</code>	Return True if this path matches the given pattern.
<code>merge_tree(dst[, symlinks, update, ...])</code>	Copy entire contents of self to dst, overwriting existing contents in dst with those in self.
<code>mkdir([mode])</code>	Create a directory.
<code>mkdir_p([mode])</code>	Like <code>mkdir()</code> , but does not raise an exception if the directory already exists.
<code>move(dst[, copy_function])</code>	Recursively move a file or directory to another location.
<code>moving([lock, timeout, method])</code>	Create a moving context for this immutable path.
<code>mutate()</code>	Create a mutable context for this immutable path.
<code>normcase()</code>	Normalize case of pathname.

Continued on next page

Table 2 – continued from previous page

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

Continued on next page

Table 2 – continued from previous page

<code>rmtree(ignore_errors, onerror)</code>	Recursively delete a directory tree.
<code>rmtree_p()</code>	Like <code>rmtree()</code> , but does not raise an exception if the directory does not exist.
<code>rpartition(sep)</code>	Partition the string into three parts using the given separator.
<code>rsplit([sep, maxsplit])</code>	Return a list of the words in the string, using <code>sep</code> as the delimiter string.
<code>rstrip([chars])</code>	Return a copy of the string with trailing whitespace removed.
<code>samefile(other)</code>	Test whether two pathnames reference the same actual file
<code>split([sep, maxsplit])</code>	Return a list of the words in the string, using <code>sep</code> as the delimiter string.
<code>splitall()</code>	Return a list of the path components in this path.
<code>splitdrive()</code>	Split the drive specifier from this path.
<code>splitext()</code>	Split the filename extension from this path and return the two parts.
<code>splitlines([keepends])</code>	Return a list of the lines in the string, breaking at line boundaries.
<code>splitpath()</code>	<b>See also:</b> <code>parent, name, os.path.split()</code>
<code>splitunc()</code>	<b>See also:</b> <code>os.path.splitunc()</code>
<code>startswith(prefix[, start[, end]])</code>	Return True if S starts with the specified prefix, False otherwise.
<code>stat()</code>	Perform a <code>stat()</code> system call on this path.
<code>statvfs()</code>	Perform a <code>statvfs()</code> system call on this path.
<code>strip([chars])</code>	Return a copy of the string with leading and trailing whitespace remove.
<code>striptext()</code>	For example, <code>Path('/home/guido/python.tar.gz').striptext()</code> returns <code>Path('/home/guido/python.tar')</code> .
<code>swapcase()</code>	Convert uppercase characters to lowercase and lowercase characters to uppercase.
<code>symlink([newlink])</code>	Create a symbolic link at <code>newlink</code> , pointing here.
<code>symlink_to(target[, target_is_directory])</code>	Make this path a symlink pointing to the given path.
<code>text([encoding, errors])</code>	Open this file, read it in, return the content as a string.
<code>title()</code>	Return a version of the string where each word is titlecased.
<code>touch()</code>	Set the access/modified times of this file to the current time.
<code>translate(table)</code>	Replace each character in the string using the given translation table.
<code>unlink()</code>	Remove a file (same as <code>remove()</code> ).
<code>unlink_p()</code>	Like <code>unlink()</code> , but does not raise an exception if the file does not exist.
<code>upper()</code>	Return a copy of the string converted to uppercase.
<code>using_module()</code>	
<code>utime(times)</code>	Set the access and modified times of this file.

Continued on next page



Table 2 – continued from previous page

<code>walk()</code>	The iterator yields Path objects naming each child item of this directory and its descendants.
<code>walkdirs()</code>	
<code>walkfiles()</code>	
<code>with_base(base[, strip_length])</code>	Clone this path with a new base.
<code>with_name(new_name)</code>	<b>See also:</b> <code>pathlib.PurePath.with_name()</code>
<code>with_parent(new_parent)</code>	Clone this path with a new parent.
<code>with_posix_enabled([enable])</code>	Clone this path in posix format with posix-like separators (i.e., '/').
<code>with_stem(new_stem)</code>	Clone this path with a new stem.
<code>with_string_repr_enabled([enable])</code>	Clone this path in with string representation enabled.
<code>with_suffix(suffix)</code>	Return a new path with the file suffix changed (or added, if none)
<code>write_bytes(bytes[, append])</code>	Open this file and write the given bytes to it.
<code>write_lines(lines[, encoding, errors, ...])</code>	Write the given lines of text to this file.
<code>write_text(text[, encoding, errors, ...])</code>	Write the given text to this file.
<code>zfill(width)</code>	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(mode)**

Return True if current user has access to this path.

mode - One of the constants `os.F_OK`, `os.R_OK`, `os.W_OK`, `os.X_OK`**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.bytes**

`Path.bytes()`

Open this file, read all bytes, return them as a string.

**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](#).

**See also:**

`os.chmod()`

### mutapath.Path.chown

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

Change the owner and group by names rather than the uid or gid 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("CHANGES.rst").chunks(8192, mode='rb'):
...     hash.update(chunk)

```

This will read the file by chunks of 8192 bytes.

### mutapath.Path.clone

`Path.clone(contained) → mutapath.immutapath.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.

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: Callable[[Path, Path], 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 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)`

Recursively copy a directory tree.

The destination directory must not already exist. 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 () → 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()` → List of the files in this directory.

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 `self.module()`, which defaults to `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()` → `mutapath.immutapath.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)`

Iterate over this subtree and yield all existing files (of any kind, including directories) matching the given relative pattern.

**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()`

Test whether a path is absolute

**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.

Use `keyword.iskeyword()` to test for reserved identifiers such as “def” and “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()`

Test whether a path is a mount point

**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.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='strict', 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 the content of the file is read as 8-bit characters and returned as a list of (non-Unicode) str objects.

**errors** - How to handle Unicode errors; see `help(str.decode)` for the options. Default is `'strict'`.

**retain** - If `True`, retain newline characters; but all newline character combinations (`'\r'`, `'\n'`, `'\r\n'`) are translated to `'\n'`. If `False`, newline characters are stripped off. Default is `True`.

**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.listdir**

`Path.listdir()` → 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.

**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, *, update=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: 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 the file pointed by this path and return a file object, as the built-in `open()` function does.

### 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.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.read\_bytes

`Path.read_bytes()`

Open the file in bytes mode, read it, and close the file.

### 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 the file in text mode, read it, and close the file.

**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.abstractmethod()`.

### 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: 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

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

### mutapath.Path.splitdrive

`Path.splitdrive()` → Return “(p.drive, <the rest of p>)”.

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 “(p.stripxext(), p.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 “(p.parent, p.name)”.

**See also:**

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

**mutapath.Path.splitunc**

`Path.splitunc()`

**See also:**

`os.path.splitunc()`

**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.

**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 remove.

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 given path. Note the order of arguments (*self*, *target*) is the reverse of `os.symlink`'s.

### mutapath.Path.text

`Path.text (encoding=None, errors='strict')`

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

All newline sequences are converted to `'\n'`. Keyword arguments will be passed to `open()`.

**See also:**

`lines()`

### 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 `remove()`).

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 `unlink()`, 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()`

### mutapath.Path.utime

`Path.utime(times)`

Set the access and modified times of this file.

**See also:**

`os.utime()`

### mutapath.Path.walk

`Path.walk()` → 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()` → iterator over subdirs, recursively.

**mutapath.Path.walkfiles**

`Path.walkfiles()` → iterator over files in D, 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)` → `mutapath.immutapath.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='\n', 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** - 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 `file.writelines()`.

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

**Warning:** When you use this with Unicode data, if the encoding of the existing data in the file is different from the encoding you specify with the `encoding=` parameter, the result is mixed-encoding data, which can really confuse someone trying to read the file later.

### mutapath.Path.write\_text

`Path.write_text(text, encoding=None, errors='strict', newline='\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

- **str/unicode** – The text to be written. (*text*) –
- **str** – The Unicode encoding that will be used. (*encoding*) – This is ignored if *text* isn't a Unicode string.
- **str** – How to handle Unicode encoding errors. (*errors*) – Default is 'strict'. See `help(unicode.encode)` for the options. This is ignored if *text* isn't a Unicode string.
- **keyword argument** – **str/unicode** – The sequence of (*newline*) – characters to be used to mark end-of-line. The default is `os.newline`. You can also specify `None` to leave all newlines as they are in *text*.
- **keyword argument** – **bool** – Specifies what to do if (*append*) – 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.newline`; on Windows, for example, the end-of-line marker is `'\r\n'`).

If you don't like your platform's default, you can override it using the `newline=` keyword argument. If you specifically want `write_text()` to preserve the newlines as-is, use `newline=None`.

This applies to Unicode text the same as to 8-bit text, except there are three additional standard Unicode end-of-line sequences: `u'\x85'`, `u'\r\x85'`, and `u'\u2028'`.

(This is slightly different from when you open a file for writing with `fopen(filename, "w")` in C or `open(filename, 'w')` in Python.)

— 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**

<i>anchor</i>	The concatenation of the drive and root, or ‘’.
<i>atime</i>	Last access time of the file.
<i>base</i>	Get the path base (i.e., the parent of the file).
<i>ctime</i>	Creation time of the file.
<i>cwd</i>	Return a new path pointing to the current working directory (as returned by <code>os.getcwd()</code> ).
<i>dirname</i>	Returns the directory component of a pathname
<i>drive</i>	‘’.
<i>ext</i>	The file extension, for example ‘.py’.
<i>home</i>	Get the home path of the current path representation.
<i>joinpath</i>	
<i>lock</i>	Generate a cached file locker for this file with the additional suffix ‘.lock’.
<i>mtime</i>	Last-modified time of the file.
<i>name</i>	The final path component, if any.
<i>namebase</i>	
<i>parent</i>	The logical parent of the path.
<i>parents</i>	A sequence of this path’s logical parents.
<i>parts</i>	An object providing sequence-like access to the components in the filesystem path.
<i>posix_enabled</i>	If set to True, the the representation of this path will always follow the posix format, even on NT filesystems.
<i>posix_string</i>	
<i>root</i>	The root of the path, if any.
<i>size</i>	Size of the file, in bytes.
<i>stem</i>	The final path component, minus its last suffix.
<i>string_repr_enabled</i>	If set to True, the the representation of this path will always be returned unwrapped as the path’s string.
<i>suffix</i>	The final component’s last suffix, if any.
<i>suffixes</i>	A list of the final component’s suffixes, if any.
<i>to_pathlib</i>	Return the contained path as <code>pathlib.Path</code> 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.

**See also:**

`getatime()`, `os.path.getatime()`

### mutapath.Path.base

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

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

**See also:**

`parent`

### 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**

Returns the directory component of a pathname

### mutapath.Path.drive

**property** Path.**drive**

“.”

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

**Type** The drive specifier, for example “C”

### mutapath.Path.ext

**property** Path.**ext**

The file extension, for example ‘.py’.

### mutapath.Path.home

**property** Path.home

Get the home path of the current path representation.

**Returns** the home path

**Example**

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

### mutapath.Path.joinpath

Path.joinpath = functools.partial(<function Path.joinpath>, <class 'mutapath.immutapath.Path'>)

### 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.

**See also:**

`getmtime()`, `os.path.getmtime()`

### mutapath.Path.name

**property** Path.name

The final path component, if any.

### mutapath.Path.namebase

**property** Path.namebase



**mutapath.Path.parent****property** `Path.parent`

The logical parent of the path.

**mutapath.Path.parents****property** `Path.parents`

A sequence of this path's logical parents.

**mutapath.Path.parts****property** `Path.parts`

An object providing sequence-like access to the components in the filesystem path.

**mutapath.Path.posix\_enabled****property** `Path.posix_enabled`

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

**mutapath.Path.posix\_string**`Path.posix_string = functools.partial(<function Path.posix_string>, <class 'mutapath.i`**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`

The final path component, minus its last suffix.

**mutapath.Path.string\_repr\_enabled****property** `Path.string_repr_enabled`

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`

The final component's last suffix, if any.

**mutapath.Path.suffixes****property** `Path.suffixes`

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

**mutapath.Path.to\_pathlib****property** `Path.to_pathlib`Return the contained path as `pathlib.Path` representation. :return: the converted path

### 3.1.2 mutapath.MutaPath

```
class mutapath.MutaPath (contained: Union[mutapath.mutapath.MutaPath, mutapath.immutapath.Path, path.Path, pathlib.PurePath, str] = "", *, posix: Optional[bool] = None, string_repr: Optional[bool] = None)
```

Bases: `mutapath.immutapath.Path`

Mutable Path

```
__init__ (contained: Union[mutapath.mutapath.MutaPath, mutapath.immutapath.Path, path.Path, pathlib.PurePath, str] = "", *, posix: Optional[bool] = None, string_repr: Optional[bool] = None)
```

Initialize self. See `help(type(self))` for accurate signature.**Methods**

<code>absolute()</code>	Return an absolute version of this path.
<code>abspath()</code>	Return an absolute path.
<code>access(mode)</code>	Return <code>True</code> if current user has access to this path.
<code>as_posix()</code>	Return the string representation of the path with forward (/) slashes.
<code>as_uri()</code>	Return the path as a 'file' URI.
<code>basename()</code>	<b>See also:</b> <code>name, os.path.basename()</code>
<code>bytes()</code>	Open this file, read all bytes, return them as a string.
<code>capitalize()</code>	Return a capitalized version of the string.
<code>casefold()</code>	Return a version of the string suitable for caseless comparisons.
<code>cd()</code>	Change the current working directory to the specified path.
<code>center(width[, fillchar])</code>	Return a centered string of length width.
<code>chdir()</code>	Change the current working directory to the specified path.
<code>chmod(mode)</code>	Set the mode.

Continued on next page

Table 4 – continued from previous page

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

Continued on next page

Table 4 – continued from previous page

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

Continued on next page

Table 4 – continued from previous page

<code>istitle()</code>	Return True if the string is a title-cased string, False otherwise.
<code>isupper()</code>	Return True if the string is an uppercase string, False otherwise.
<code>iterdir()</code>	Iterate over the files in this directory.
<code>join(iterable)</code>	Concatenate any number of strings.
<code>joinpath(*others)</code>	<code>partial(func, *args, **keywords)</code> - new function with partial application of the given arguments and keywords.
<code>lchmod(mode)</code>	Like <code>chmod()</code> , except if the path points to a symlink, the symlink's permissions are changed, rather than its target's.
<code>lines([encoding, errors, retain])</code>	Open this file, read all lines, return them in a list.
<code>link(newpath)</code>	Create a hard link at <code>newpath</code> , pointing to this file.
<code>listdir()</code>	Use <code>files()</code> or <code>dirs()</code> instead if you want a listing of just files or just subdirectories.
<code>ljust(width[, fillchar])</code>	Return a left-justified string of length width.
<code>lower()</code>	Return a copy of the string converted to lowercase.
<code>lstat()</code>	Like <code>stat()</code> , but do not follow symbolic links.
<code>rstrip([chars])</code>	Return a copy of the string with leading whitespace removed.
<code>makedirs(name [, mode, exist_ok])</code>	Super-mkdir; create a leaf directory and all intermediate ones.
<code>makedirs_p([mode])</code>	Like <code>makedirs()</code> , but does not raise an exception if the directory already exists.
<code>match(path_pattern)</code>	Return True if this path matches the given pattern.
<code>merge_tree(other, *args, **kwargs)</code>	Move, merge and mutate this path to the given other path.
<code>mkdir([mode])</code>	Create a directory.
<code>mkdir_p([mode])</code>	Like <code>mkdir()</code> , but does not raise an exception if the directory already exists.
<code>move(dst[, copy_function])</code>	Recursively move a file or directory to another location.
<code>moving([lock, timeout, method])</code>	Create a moving context for this immutable path.
<code>mutate()</code>	Create a mutable context for this immutable path.
<code>normcase()</code>	Normalize case of pathname.
<code>normpath()</code>	Normalize path, eliminating double slashes, etc.
<code>open(*args, **kwargs)</code>	Open the file pointed by this path and return a file object, as the built-in <code>open()</code> function does.
<code>partition(sep)</code>	Partition the string into three parts using the given separator.
<code>pathconf(name)</code>	Return the configuration limit name for the file or directory path.
<code>read_bytes()</code>	Open the file in bytes mode, read it, and close the file.
<code>read_hash(hash_name)</code>	Calculate given hash for this file.
<code>read_hexhash(hash_name)</code>	Calculate given hash for this file, returning hexdigest.
<code>read_md5()</code>	Calculate the md5 hash for this file.
<code>read_text([encoding, errors])</code>	Open the file in text mode, read it, and close the file.
<code>readlink()</code>	Return the path to which this symbolic link points.

Continued on next page

Table 4 – continued from previous page

<code>readlinkabs()</code>	Return the path to which this symbolic link points.
<code>realpath()</code>	Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path.
<code>relative_to(*other)</code>	Return the relative path to another path identified by the passed arguments.
<code>relpath([start])</code>	Return this path as a relative path, based from <i>start</i> , which defaults to the current working directory.
<code>relpathto(dest)</code>	Return a relative path from <i>self</i> to <i>dest</i> .
<code>remove()</code>	Remove a file (same as <code>unlink()</code> ).
<code>remove_p()</code>	Like <code>remove()</code> , but does not raise an exception if the file does not exist.
<code>removedirs(name)</code>	Super-rmdir; remove a leaf directory and all empty intermediate ones.
<code>removedirs_p()</code>	Like <code>removedirs()</code> , but does not raise an exception if the directory is not empty or does not exist.
<code>rename(new)</code>	Rename a file or directory.
<code>renames(old, new)</code>	Super-rename; create directories as necessary and delete any left empty.
<code>renaming([lock, timeout, method])</code>	Create a renaming context for this immutable path.
<code>replace(old, new[, count])</code>	Return a copy with all occurrences of substring <i>old</i> replaced by <i>new</i> .
<code>resolve([strict])</code>	Make the path absolute, resolving all symlinks on the way and also normalizing it (for example turning slashes into backslashes under Windows).
<code>rfind(sub[, start[, end]])</code>	Return the highest index in <i>S</i> where substring <i>sub</i> is found, such that <i>sub</i> is contained within <i>S</i> [ <i>start</i> : <i>end</i> ].
<code>rglob(pattern)</code>	Recursively yield all existing files (of any kind, including directories) matching the given relative pattern, anywhere in this subtree.
<code>rindex(sub[, start[, end]])</code>	Return the highest index in <i>S</i> where substring <i>sub</i> is found, such that <i>sub</i> is contained within <i>S</i> [ <i>start</i> : <i>end</i> ].
<code>rjust(width[, fillchar])</code>	Return a right-justified string of length <i>width</i> .
<code>rmdir()</code>	Remove a directory.
<code>rmdir_p()</code>	Like <code>rmdir()</code> , but does not raise an exception if the directory is not empty or does not exist.
<code>rmtree([ignore_errors, onerror])</code>	Recursively delete a directory tree.
<code>rmtree_p()</code>	Like <code>rmtree()</code> , but does not raise an exception if the directory does not exist.
<code>rpartition(sep)</code>	Partition the string into three parts using the given separator.
<code>rsplit([sep, maxsplit])</code>	Return a list of the words in the string, using <i>sep</i> as the delimiter string.
<code>rstrip([chars])</code>	Return a copy of the string with trailing whitespace removed.
<code>samefile(other)</code>	Test whether two pathnames reference the same actual file
<code>split([sep, maxsplit])</code>	Return a list of the words in the string, using <i>sep</i> as the delimiter string.
<code>splitall()</code>	Return a list of the path components in this path.
<code>splitdrive()</code>	Split the drive specifier from this path.

Continued on next page

Table 4 – continued from previous page

<code>splitext()</code>	Split the filename extension from this path and return the two parts.
<code>splitlines([keepends])</code>	Return a list of the lines in the string, breaking at line boundaries.
<code>splitpath()</code>	<b>See also:</b> <code>parent, name, os.path.split()</code>
<code>splitunc()</code>	<b>See also:</b> <code>os.path.splitunc()</code>
<code>startswith(prefix[, start[, end]])</code>	Return True if S starts with the specified prefix, False otherwise.
<code>stat()</code>	Perform a <code>stat()</code> system call on this path.
<code>statvfs()</code>	Perform a <code>statvfs()</code> system call on this path.
<code>strip([chars])</code>	Return a copy of the string with leading and trailing whitespace remove.
<code>stripext()</code>	For example, <code>Path('/home/guido/python.tar.gz').stripext()</code> returns <code>Path('/home/guido/python.tar')</code> .
<code>swapcase()</code>	Convert uppercase characters to lowercase and lowercase characters to uppercase.
<code>symlink([newlink])</code>	Create a symbolic link at <i>newlink</i> , pointing here.
<code>symlink_to(target[, target_is_directory])</code>	Make this path a symlink pointing to the given path.
<code>text([encoding, errors])</code>	Open this file, read it in, return the content as a string.
<code>title()</code>	Return a version of the string where each word is titlecased.
<code>touch()</code>	Set the access/modified times of this file to the current time.
<code>translate(table)</code>	Replace each character in the string using the given translation table.
<code>unlink()</code>	Remove a file (same as <code>remove()</code> ).
<code>unlink_p()</code>	Like <code>unlink()</code> , but does not raise an exception if the file does not exist.
<code>upper()</code>	Return a copy of the string converted to uppercase.
<code>using_module()</code>	
<code>utime(times)</code>	Set the access and modified times of this file.
<code>walk()</code>	The iterator yields <code>Path</code> objects naming each child item of this directory and its descendants.
<code>walkdirs()</code>	
<code>walkfiles()</code>	
<code>with_base(base[, strip_length])</code>	Clone this path with a new base.
<code>with_name(new_name)</code>	<b>See also:</b> <code>pathlib.PurePath.with_name()</code>
<code>with_parent(new_parent)</code>	Clone this path with a new parent.
<code>with_poxis_enabled([enable])</code>	Clone this path in posix format with posix-like separators (i.e., <code>'/'</code> ).
<code>with_stem(new_stem)</code>	Clone this path with a new stem.
<code>with_string_repr_enabled([enable])</code>	Clone this path in with string representation enabled.
<code>with_suffix(suffix)</code>	Return a new path with the file suffix changed (or added, if none)

Continued on next page

Table 4 – continued from previous page

<code>write_bytes(bytes[, append])</code>	Open this file and write the given bytes to it.
<code>write_lines(lines[, encoding, errors, ...])</code>	Write the given lines of text to this file.
<code>write_text(text[, encoding, errors, ...])</code>	Write the given text to this file.
<code>zfill(width)</code>	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.abstractmethod**

`MutaPath.abstractmethod()`

Return an absolute path.

### **mutapath.MutaPath.access**

`MutaPath.access(mode)`

Return True if current user has access to this path.

mode - One of the constants `os.F_OK`, `os.R_OK`, `os.W_OK`, `os.X_OK`

**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.bytes

`MutaPath.bytes()`

Open this file, read all bytes, return them as a string.

### 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](#).

**See also:**

`os.chmod()`

### mutapath.MutaPath.chown

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

Change the owner and group by names rather than the uid or gid 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("CHANGES.rst").chunks(8192, mode='rb'):
...     hash.update(chunk)
```

This will read the file by chunks of 8192 bytes.

### mutapath.MutaPath.clone

MutaPath.**clone** (*contained*) → mutapath.immutapath.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*.

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: Callable[[Path, Path], 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 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*)

Recursively copy a directory tree.

The destination directory must not already exist. 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** () → 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()` → List of the files in this directory.

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 `self.module()`, which defaults to `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()` → `mutapath.immutapath.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*)

Iterate over this subtree and yield all existing files (of any kind, including directories) matching the given relative pattern.

### 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()`

Test whether a path is absolute

**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.

Use `keyword.iskeyword()` to test for reserved identifiers such as “def” and “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()`

Test whether a path is a mount point

**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='strict', 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 the content of the file is read as 8-bit characters and returned as a list of (non-Unicode) str objects.

**errors** - How to handle Unicode errors; see `help(str.decode)` for the options. Default is `'strict'`.

**retain** - If `True`, retain newline characters; but all newline character combinations (`'\r'`, `'\n'`, `'\r\n'`) are translated to `'\n'`. If `False`, newline characters are stripped off. Default is `True`.

#### 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.listdir

MutaPath.**listdir**() → 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.

**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: 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 the file pointed by this path and return a file object, as the built-in `open()` function does.

### 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.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.read\_bytes**

`MutaPath.read_bytes ()`

Open the file in bytes mode, read it, and close the file.

### **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 the file in text mode, read it, and close the file.



**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: 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.rename`)

### 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

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

### mutapath.MutaPath.splitdrive

MutaPath.**splitdrive** () → Return “(p.drive, <the rest of p>)”.

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 “(p.stripext(), p.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 “(p.parent, p.name)”.

**See also:**

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

### mutapath.MutaPath.splitunc

MutaPath.**splitunc**()

**See also:**

`os.path.splitunc()`

### 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.

**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 remove.

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 given path. Note the order of arguments (*self*, *target*) is the reverse of `os.symlink`'s.

### mutapath.MutaPath.text

MutaPath.**text**(*encoding=None, errors='strict'*)

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

All newline sequences are converted to `'\n'`. Keyword arguments will be passed to `open()`.

**See also:**

*lines()*

### **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 `remove()`).

**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 *unlink()*, 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()`



### mutapath.MutaPath.utime

MutaPath.**utime** (*times*)

Set the access and modified times of this file.

**See also:**

`os.utime()`

### mutapath.MutaPath.walk

MutaPath.**walk**() → 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**() → iterator over subdirs, recursively.

### mutapath.MutaPath.walkfiles

MutaPath.**walkfiles**() → iterator over files in D, 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*) → mutapath.immutapath.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='\n', 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** - 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 `file.writelines()`.

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

**Warning:** When you use this with Unicode data, if the encoding of the existing data in the file is different from the encoding you specify with the *encoding=* parameter, the result is mixed-encoding data, which can really confuse someone trying to read the file later.

### 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

- - **str/unicode** - The text to be written. (*text*) -

- **- str - The Unicode encoding that will be used.** (*encoding*) - This is ignored if *text* isn't a Unicode string.
- **- str - How to handle Unicode encoding errors.** (*errors*) - Default is 'strict'. See `help(unicode.encode)` for the options. This is ignored if *text* isn't a Unicode string.
- **- keyword argument - str/unicode - The sequence of (*linesep*) -** characters to be used to mark end-of-line. The default is `os.linesep`. You can also specify `None` to leave all newlines as they are in *text*.
- **- keyword argument - bool - Specifies what to do if (*append*) -** 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`').

If you don't like your platform's default, you can override it using the *linesep*= keyword argument. If you specifically want `write_text()` to preserve the newlines as-is, use *linesep*=`None`.

This applies to Unicode text the same as to 8-bit text, except there are three additional standard Unicode end-of-line sequences: `u'\x85'`, `u'\r\x85'`, and `u'\u2028'`.

(This is slightly different from when you open a file for writing with `fopen(filename, "w")` in C or `open(filename, 'w')` in Python.)

— 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

<i>anchor</i>	The concatenation of the drive and root, or ''.
<i>atime</i>	Last access time of the file.
<i>base</i>	Get the path base (i.e., the parent of the file).
<i>ctime</i>	Creation time of the file.
<i>cwd</i>	Return a new path pointing to the current working directory (as returned by <code>os.getcwd()</code> ).
<i>dirname</i>	Returns the directory component of a pathname
<i>drive</i>	““.
<i>ext</i>	The file extension, for example ' <code>.py</code> '.

Continued on next page

Table 5 – continued from previous page

<code>home</code>	Get the home path of the current path representation.
<code>lock</code>	Generate a cached file locker for this file with the additional suffix <code>‘.lock’</code> .
<code>mtime</code>	Last-modified time of the file.
<code>name</code>	The final path component, if any.
<code>namebase</code>	
<code>parent</code>	The logical parent of the path.
<code>parents</code>	A sequence of this path’s logical parents.
<code>parts</code>	An object providing sequence-like access to the components in the filesystem path.
<code>posix_enabled</code>	If set to True, the the representation of this path will always follow the posix format, even on NT filesystems.
<code>posix_string</code>	
<code>root</code>	The root of the path, if any.
<code>size</code>	Size of the file, in bytes.
<code>stem</code>	The final path component, minus its last suffix.
<code>string_repr_enabled</code>	If set to True, the the representation of this path will always be returned unwrapped as the path’s string.
<code>suffix</code>	The final component’s last suffix, if any.
<code>suffixes</code>	A list of the final component’s suffixes, if any.
<code>to_pathlib</code>	Return the contained path as <code>pathlib.Path</code> 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.

**See also:**`getatime()`, `os.path.getatime()`**mutapath.MutaPath.base****property** `MutaPath.base`

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

**See also:**`parent`**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`

Returns the directory component of a pathname

### **mutapath.MutaPath.drive**

**property** `MutaPath.drive`

““.

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

**Type** The drive specifier, for example ““C

### **mutapath.MutaPath.ext**

**property** `MutaPath.ext`

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

### **mutapath.MutaPath.home**

**property** `MutaPath.home`

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.

**See also:**

`getmtime()`, `os.path.getmtime()`

**mutapath.MutaPath.name**

**property** `MutaPath.name`

The final path component, if any.

**mutapath.MutaPath.namebase**

**property** `MutaPath.namebase`

**mutapath.MutaPath.parent**

**property** `MutaPath.parent`

The logical parent of the path.

**mutapath.MutaPath.parents**

**property** `MutaPath.parents`

A sequence of this path's logical parents.

**mutapath.MutaPath.parts**

**property** `MutaPath.parts`

An object providing sequence-like access to the components in the filesystem path.

**mutapath.MutaPath.posix\_enabled**

**property** `MutaPath.posix_enabled`

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

**mutapath.MutaPath.posix\_string**

`MutaPath.posix_string = functools.partial(<function Path.posix_string>, <class 'mutapath.MutaPath'>)`

**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`  
The final path component, minus its last suffix.

**mutapath.MutaPath.string\_repr\_enabled**

**property** `MutaPath.string_repr_enabled`  
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`  
The final component's last suffix, if any.

**mutapath.MutaPath.suffixes**

**property** `MutaPath.suffixes`  
A list of the final component's suffixes, if any.

**mutapath.MutaPath.to\_pathlib**

**property** `MutaPath.to_pathlib`  
Return the contained path as `pathlib.Path` representation. :return: the converted path

### 3.1.3 mutapath.exceptions.PathException

**class** `mutapath.exceptions.PathException`  
Bases: `BaseException`

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

**\_\_init\_\_()**  
Initialize self. See `help(type(self))` for accurate signature.



### 3.1.4 mutapath.lock\_dummy.DummyFileLock

**class** mutapath.lock\_dummy.DummyFileLock(*lock\_file*, *timeout=-1*)

Bases: filelock.BaseFileLock

**\_\_init\_\_**(*lock\_file*, *timeout=-1*)

#### Methods

<i>acquire</i> ([ <i>timeout</i> , <i>poll_intervall</i> ])	Doing nothing
<i>release</i> ([ <i>force</i> ])	Doing nothing

#### mutapath.lock\_dummy.DummyFileLock.acquire

DummyFileLock.**acquire**(*timeout=None*, *poll\_intervall=0.05*)

Doing nothing

#### mutapath.lock\_dummy.DummyFileLock.release

DummyFileLock.**release**(*force=False*)

Doing nothing

#### Attributes

<i>is_locked</i>	True, if the object holds the file lock.
<i>lock_file</i>	The path to the lock file.
<i>timeout</i>	You can set a default timeout for the filelock.

#### mutapath.lock\_dummy.DummyFileLock.is\_locked

**property** DummyFileLock.**is\_locked**

True, if the object holds the file lock.

Changed in version 2.0.0: This was previously a method and is now a property.

#### mutapath.lock\_dummy.DummyFileLock.lock\_file

**property** DummyFileLock.**lock\_file**

The path to the lock file.

#### mutapath.lock\_dummy.DummyFileLock.timeout

**property** DummyFileLock.**timeout**

You can set a default timeout for the filelock. 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.

New in version 2.0.0.

## 3.2 Indices and tables

- `genindex`

## Symbols

`__init__()` (*mutapath.MutaPath* method), 46  
`__init__()` (*mutapath.Path* method), 7  
`__init__()` (*mutapath.exceptions.PathException* method), 84  
`__init__()` (*mutapath.lock\_dummy.DummyFileLock* method), 85

## A

`absolute()` (*mutapath.MutaPath* method), 52  
`absolute()` (*mutapath.Path* method), 13  
`abspath()` (*mutapath.MutaPath* method), 52  
`abspath()` (*mutapath.Path* method), 13  
`access()` (*mutapath.MutaPath* method), 52  
`access()` (*mutapath.Path* method), 13  
`acquire()` (*mutapath.lock\_dummy.DummyFileLock* method), 85  
`anchor()` (*mutapath.MutaPath* property), 81  
`anchor()` (*mutapath.Path* property), 42  
`as_posix()` (*mutapath.MutaPath* method), 52  
`as_posix()` (*mutapath.Path* method), 13  
`as_uri()` (*mutapath.MutaPath* method), 52  
`as_uri()` (*mutapath.Path* method), 14  
`atime()` (*mutapath.MutaPath* property), 81  
`atime()` (*mutapath.Path* property), 43

## B

`base()` (*mutapath.MutaPath* property), 81  
`base()` (*mutapath.Path* property), 43  
`basename()` (*mutapath.MutaPath* method), 52  
`basename()` (*mutapath.Path* method), 14  
`bytes()` (*mutapath.MutaPath* method), 53  
`bytes()` (*mutapath.Path* method), 14

## C

`capitalize()` (*mutapath.MutaPath* method), 53  
`capitalize()` (*mutapath.Path* method), 14  
`casefold()` (*mutapath.MutaPath* method), 53  
`casefold()` (*mutapath.Path* method), 14  
`cd()` (*mutapath.MutaPath* method), 53  
`cd()` (*mutapath.Path* method), 14  
`center()` (*mutapath.MutaPath* method), 53

`center()` (*mutapath.Path* method), 14  
`chdir()` (*mutapath.MutaPath* method), 53  
`chdir()` (*mutapath.Path* method), 15  
`chmod()` (*mutapath.MutaPath* method), 53  
`chmod()` (*mutapath.Path* method), 15  
`chown()` (*mutapath.MutaPath* method), 54  
`chown()` (*mutapath.Path* method), 15  
`chroot()` (*mutapath.MutaPath* method), 54  
`chroot()` (*mutapath.Path* method), 15  
`chunks()` (*mutapath.MutaPath* method), 54  
`chunks()` (*mutapath.Path* method), 15  
`clone()` (*mutapath.MutaPath* method), 54  
`clone()` (*mutapath.Path* method), 15  
`copy()` (*mutapath.MutaPath* method), 54  
`copy()` (*mutapath.Path* method), 16  
`copy2()` (*mutapath.MutaPath* method), 54  
`copy2()` (*mutapath.Path* method), 16  
`copyfile()` (*mutapath.MutaPath* method), 55  
`copyfile()` (*mutapath.Path* method), 16  
`copying()` (*mutapath.MutaPath* method), 55  
`copying()` (*mutapath.Path* method), 16  
`copymode()` (*mutapath.MutaPath* method), 55  
`copymode()` (*mutapath.Path* method), 16  
`copystat()` (*mutapath.MutaPath* method), 55  
`copystat()` (*mutapath.Path* method), 17  
`copytree()` (*mutapath.MutaPath* method), 55  
`copytree()` (*mutapath.Path* method), 17  
`count()` (*mutapath.MutaPath* method), 56  
`count()` (*mutapath.Path* method), 17  
`ctime()` (*mutapath.MutaPath* property), 81  
`ctime()` (*mutapath.Path* property), 43  
`cwd()` (*mutapath.MutaPath* property), 82  
`cwd()` (*mutapath.Path* property), 43

## D

`dirname()` (*mutapath.MutaPath* property), 82  
`dirname()` (*mutapath.Path* property), 43  
`dirs()` (*mutapath.MutaPath* method), 56  
`dirs()` (*mutapath.Path* method), 17  
`drive()` (*mutapath.MutaPath* property), 82  
`drive()` (*mutapath.Path* property), 43  
`DummyFileLock` (class in *mutapath.lock\_dummy*), 85

## E

`encode()` (*mutapath.MutaPath* method), 56  
`encode()` (*mutapath.Path* method), 18  
`endswith()` (*mutapath.MutaPath* method), 56  
`endswith()` (*mutapath.Path* method), 18  
`exists()` (*mutapath.MutaPath* method), 57  
`exists()` (*mutapath.Path* method), 18  
`expand()` (*mutapath.MutaPath* method), 57  
`expand()` (*mutapath.Path* method), 18  
`expandtabs()` (*mutapath.MutaPath* method), 57  
`expandtabs()` (*mutapath.Path* method), 18  
`expanduser()` (*mutapath.MutaPath* method), 57  
`expanduser()` (*mutapath.Path* method), 18  
`expandvars()` (*mutapath.MutaPath* method), 57  
`expandvars()` (*mutapath.Path* method), 18  
`ext()` (*mutapath.MutaPath* property), 82  
`ext()` (*mutapath.Path* property), 43

## F

`files()` (*mutapath.MutaPath* method), 57  
`files()` (*mutapath.Path* method), 19  
`find()` (*mutapath.MutaPath* method), 57  
`find()` (*mutapath.Path* method), 19  
`fnmatch()` (*mutapath.MutaPath* method), 57  
`fnmatch()` (*mutapath.Path* method), 19  
`format()` (*mutapath.MutaPath* method), 58  
`format()` (*mutapath.Path* method), 19  
`format_map()` (*mutapath.MutaPath* method), 58  
`format_map()` (*mutapath.Path* method), 19

## G

`get_owner()` (*mutapath.MutaPath* method), 58  
`get_owner()` (*mutapath.Path* method), 19  
`getatime()` (*mutapath.MutaPath* method), 58  
`getatime()` (*mutapath.Path* method), 20  
`getctime()` (*mutapath.MutaPath* method), 58  
`getctime()` (*mutapath.Path* method), 20  
`getcwd()` (*mutapath.MutaPath* class method), 58  
`getcwd()` (*mutapath.Path* class method), 20  
`getmtime()` (*mutapath.MutaPath* method), 59  
`getmtime()` (*mutapath.Path* method), 20  
`getsize()` (*mutapath.MutaPath* method), 59  
`getsize()` (*mutapath.Path* method), 20  
`glob()` (*mutapath.MutaPath* method), 59  
`glob()` (*mutapath.Path* method), 20  
`group()` (*mutapath.MutaPath* method), 59  
`group()` (*mutapath.Path* method), 20

## H

`home()` (*mutapath.MutaPath* property), 82  
`home()` (*mutapath.Path* property), 44

## I

`iglob()` (*mutapath.MutaPath* method), 59

`iglob()` (*mutapath.Path* method), 21  
`in_place()` (*mutapath.MutaPath* method), 59  
`in_place()` (*mutapath.Path* method), 21  
`index()` (*mutapath.MutaPath* method), 60  
`index()` (*mutapath.Path* method), 21  
`is_absolute()` (*mutapath.MutaPath* method), 60  
`is_absolute()` (*mutapath.Path* method), 21  
`is_block_device()` (*mutapath.MutaPath* method), 60  
`is_block_device()` (*mutapath.Path* method), 22  
`is_char_device()` (*mutapath.MutaPath* method), 60  
`is_char_device()` (*mutapath.Path* method), 22  
`is_dir()` (*mutapath.MutaPath* method), 60  
`is_dir()` (*mutapath.Path* method), 22  
`is_fifo()` (*mutapath.MutaPath* method), 60  
`is_fifo()` (*mutapath.Path* method), 22  
`is_file()` (*mutapath.MutaPath* method), 61  
`is_file()` (*mutapath.Path* method), 22  
`is_locked()` (*mutapath.lock\_dummy.DummyFileLock* property), 85  
`is_mount()` (*mutapath.MutaPath* method), 61  
`is_mount()` (*mutapath.Path* method), 22  
`is_reserved()` (*mutapath.MutaPath* method), 61  
`is_reserved()` (*mutapath.Path* method), 22  
`is_socket()` (*mutapath.MutaPath* method), 61  
`is_socket()` (*mutapath.Path* method), 22  
`is_symlink()` (*mutapath.MutaPath* method), 61  
`is_symlink()` (*mutapath.Path* method), 22  
`isabs()` (*mutapath.MutaPath* method), 61  
`isabs()` (*mutapath.Path* method), 23  
`isalnum()` (*mutapath.MutaPath* method), 61  
`isalnum()` (*mutapath.Path* method), 23  
`isalpha()` (*mutapath.MutaPath* method), 61  
`isalpha()` (*mutapath.Path* method), 23  
`isascii()` (*mutapath.MutaPath* method), 62  
`isascii()` (*mutapath.Path* method), 23  
`isdecimal()` (*mutapath.MutaPath* method), 62  
`isdecimal()` (*mutapath.Path* method), 23  
`isdigit()` (*mutapath.MutaPath* method), 62  
`isdigit()` (*mutapath.Path* method), 23  
`isdir()` (*mutapath.MutaPath* method), 62  
`isdir()` (*mutapath.Path* method), 23  
`isfile()` (*mutapath.MutaPath* method), 62  
`isfile()` (*mutapath.Path* method), 24  
`isidentifier()` (*mutapath.MutaPath* method), 62  
`isidentifier()` (*mutapath.Path* method), 24  
`islink()` (*mutapath.MutaPath* method), 62  
`islink()` (*mutapath.Path* method), 24  
`islower()` (*mutapath.MutaPath* method), 62  
`islower()` (*mutapath.Path* method), 24  
`ismount()` (*mutapath.MutaPath* method), 63  
`ismount()` (*mutapath.Path* method), 24  
`isnumeric()` (*mutapath.MutaPath* method), 63

isnumeric() (*mutapath.Path* method), 24  
 isprintable() (*mutapath.MutaPath* method), 63  
 isprintable() (*mutapath.Path* method), 24  
 isspace() (*mutapath.MutaPath* method), 63  
 isspace() (*mutapath.Path* method), 24  
 istitle() (*mutapath.MutaPath* method), 63  
 istitle() (*mutapath.Path* method), 25  
 isupper() (*mutapath.MutaPath* method), 63  
 isupper() (*mutapath.Path* method), 25  
 iterdir() (*mutapath.MutaPath* method), 63  
 iterdir() (*mutapath.Path* method), 25

## J

join() (*mutapath.MutaPath* method), 64  
 join() (*mutapath.Path* method), 25  
 joinpath(*mutapath.Path* attribute), 44  
 joinpath() (*mutapath.MutaPath* method), 64

## L

lchmod() (*mutapath.MutaPath* method), 64  
 lchmod() (*mutapath.Path* method), 25  
 lines() (*mutapath.MutaPath* method), 64  
 lines() (*mutapath.Path* method), 25  
 link() (*mutapath.MutaPath* method), 64  
 link() (*mutapath.Path* method), 26  
 listdir() (*mutapath.MutaPath* method), 65  
 listdir() (*mutapath.Path* method), 26  
 ljust() (*mutapath.MutaPath* method), 65  
 ljust() (*mutapath.Path* method), 26  
 lock(*mutapath.MutaPath* attribute), 82  
 lock(*mutapath.Path* attribute), 44  
 lock\_file() (*mutapath.lock\_dummy.DummyFileLock* property), 85  
 lower() (*mutapath.MutaPath* method), 65  
 lower() (*mutapath.Path* method), 26  
 lstat() (*mutapath.MutaPath* method), 65  
 lstat() (*mutapath.Path* method), 26  
 lstrip() (*mutapath.MutaPath* method), 65  
 lstrip() (*mutapath.Path* method), 26

## M

makedirs() (*mutapath.MutaPath* method), 65  
 makedirs() (*mutapath.Path* method), 27  
 makedirs\_p() (*mutapath.MutaPath* method), 65  
 makedirs\_p() (*mutapath.Path* method), 27  
 match() (*mutapath.MutaPath* method), 66  
 match() (*mutapath.Path* method), 27  
 merge\_tree() (*mutapath.MutaPath* method), 66  
 merge\_tree() (*mutapath.Path* method), 27  
 mkdir() (*mutapath.MutaPath* method), 66  
 mkdir() (*mutapath.Path* method), 27  
 mkdir\_p() (*mutapath.MutaPath* method), 66  
 mkdir\_p() (*mutapath.Path* method), 27  
 move() (*mutapath.MutaPath* method), 66

move() (*mutapath.Path* method), 28  
 moving() (*mutapath.MutaPath* method), 67  
 moving() (*mutapath.Path* method), 28  
 mtime() (*mutapath.MutaPath* property), 83  
 mtime() (*mutapath.Path* property), 44  
 MutaPath (class in *mutapath*), 46  
 mutate() (*mutapath.MutaPath* method), 67  
 mutate() (*mutapath.Path* method), 28

## N

name() (*mutapath.MutaPath* property), 83  
 name() (*mutapath.Path* property), 44  
 namebase() (*mutapath.MutaPath* property), 83  
 namebase() (*mutapath.Path* property), 44  
 normcase() (*mutapath.MutaPath* method), 67  
 normcase() (*mutapath.Path* method), 29  
 normpath() (*mutapath.MutaPath* method), 67  
 normpath() (*mutapath.Path* method), 29

## O

open() (*mutapath.MutaPath* method), 67  
 open() (*mutapath.Path* method), 29

## P

parent() (*mutapath.MutaPath* property), 83  
 parent() (*mutapath.Path* property), 45  
 parents() (*mutapath.MutaPath* property), 83  
 parents() (*mutapath.Path* property), 45  
 partition() (*mutapath.MutaPath* method), 67  
 partition() (*mutapath.Path* method), 29  
 parts() (*mutapath.MutaPath* property), 83  
 parts() (*mutapath.Path* property), 45  
 Path (class in *mutapath*), 7  
 pathconf() (*mutapath.MutaPath* method), 68  
 pathconf() (*mutapath.Path* method), 29  
 PathException (class in *mutapath.exceptions*), 84  
 posix\_enabled() (*mutapath.MutaPath* property), 83  
 posix\_enabled() (*mutapath.Path* property), 45  
 posix\_string(*mutapath.MutaPath* attribute), 83  
 posix\_string(*mutapath.Path* attribute), 45

## R

read\_bytes() (*mutapath.MutaPath* method), 68  
 read\_bytes() (*mutapath.Path* method), 29  
 read\_hash() (*mutapath.MutaPath* method), 68  
 read\_hash() (*mutapath.Path* method), 29  
 read\_hexhash() (*mutapath.MutaPath* method), 68  
 read\_hexhash() (*mutapath.Path* method), 30  
 read\_md5() (*mutapath.MutaPath* method), 68  
 read\_md5() (*mutapath.Path* method), 30  
 read\_text() (*mutapath.MutaPath* method), 68  
 read\_text() (*mutapath.Path* method), 30  
 readlink() (*mutapath.MutaPath* method), 69



`readlink()` (*mutapath.Path* method), 30  
`readlinkabs()` (*mutapath.MutaPath* method), 69  
`readlinkabs()` (*mutapath.Path* method), 30  
`realpath()` (*mutapath.MutaPath* method), 69  
`realpath()` (*mutapath.Path* method), 30  
`relative_to()` (*mutapath.MutaPath* method), 69  
`relative_to()` (*mutapath.Path* method), 31  
`release()` (*mutapath.lock\_dummy.DummyFileLock* method), 85  
`relpath()` (*mutapath.MutaPath* method), 69  
`relpath()` (*mutapath.Path* method), 31  
`relpathto()` (*mutapath.MutaPath* method), 69  
`relpathto()` (*mutapath.Path* method), 31  
`remove()` (*mutapath.MutaPath* method), 69  
`remove()` (*mutapath.Path* method), 31  
`remove_p()` (*mutapath.MutaPath* method), 70  
`remove_p()` (*mutapath.Path* method), 31  
`removedirs()` (*mutapath.MutaPath* method), 70  
`removedirs()` (*mutapath.Path* method), 31  
`removedirs_p()` (*mutapath.MutaPath* method), 70  
`removedirs_p()` (*mutapath.Path* method), 31  
`rename()` (*mutapath.MutaPath* method), 70  
`rename()` (*mutapath.Path* method), 32  
`renames()` (*mutapath.MutaPath* method), 70  
`renames()` (*mutapath.Path* method), 32  
`renaming()` (*mutapath.MutaPath* method), 70  
`renaming()` (*mutapath.Path* method), 32  
`replace()` (*mutapath.MutaPath* method), 71  
`replace()` (*mutapath.Path* method), 32  
`resolve()` (*mutapath.MutaPath* method), 71  
`resolve()` (*mutapath.Path* method), 33  
`rfind()` (*mutapath.MutaPath* method), 71  
`rfind()` (*mutapath.Path* method), 33  
`rglob()` (*mutapath.MutaPath* method), 71  
`rglob()` (*mutapath.Path* method), 33  
`rindex()` (*mutapath.MutaPath* method), 71  
`rindex()` (*mutapath.Path* method), 33  
`rjust()` (*mutapath.MutaPath* method), 72  
`rjust()` (*mutapath.Path* method), 33  
`rmdir()` (*mutapath.MutaPath* method), 72  
`rmdir()` (*mutapath.Path* method), 33  
`rmdir_p()` (*mutapath.MutaPath* method), 72  
`rmdir_p()` (*mutapath.Path* method), 33  
`rmtree()` (*mutapath.MutaPath* method), 72  
`rmtree()` (*mutapath.Path* method), 34  
`rmtree_p()` (*mutapath.MutaPath* method), 72  
`rmtree_p()` (*mutapath.Path* method), 34  
`root()` (*mutapath.MutaPath* property), 84  
`root()` (*mutapath.Path* property), 45  
`rpartition()` (*mutapath.MutaPath* method), 72  
`rpartition()` (*mutapath.Path* method), 34  
`rsplit()` (*mutapath.MutaPath* method), 73  
`rsplit()` (*mutapath.Path* method), 34  
`rstrip()` (*mutapath.MutaPath* method), 73

`rstrip()` (*mutapath.Path* method), 34

## S

`samefile()` (*mutapath.MutaPath* method), 73  
`samefile()` (*mutapath.Path* method), 34  
`size()` (*mutapath.MutaPath* property), 84  
`size()` (*mutapath.Path* property), 45  
`split()` (*mutapath.MutaPath* method), 73  
`split()` (*mutapath.Path* method), 35  
`splitall()` (*mutapath.MutaPath* method), 73  
`splitall()` (*mutapath.Path* method), 35  
`splitdrive()` (*mutapath.MutaPath* method), 73  
`splitdrive()` (*mutapath.Path* method), 35  
`splitext()` (*mutapath.MutaPath* method), 74  
`splitext()` (*mutapath.Path* method), 35  
`splitlines()` (*mutapath.MutaPath* method), 74  
`splitlines()` (*mutapath.Path* method), 35  
`splitpath()` (*mutapath.MutaPath* method), 74  
`splitpath()` (*mutapath.Path* method), 35  
`splitunc()` (*mutapath.MutaPath* method), 74  
`splitunc()` (*mutapath.Path* method), 36  
`startswith()` (*mutapath.MutaPath* method), 74  
`startswith()` (*mutapath.Path* method), 36  
`stat()` (*mutapath.MutaPath* method), 74  
`stat()` (*mutapath.Path* method), 36  
`statvfs()` (*mutapath.MutaPath* method), 75  
`statvfs()` (*mutapath.Path* method), 36  
`stem()` (*mutapath.MutaPath* property), 84  
`stem()` (*mutapath.Path* property), 45  
`string_repr_enabled()` (*mutapath.MutaPath* property), 84  
`string_repr_enabled()` (*mutapath.Path* property), 45  
`strip()` (*mutapath.MutaPath* method), 75  
`strip()` (*mutapath.Path* method), 36  
`stripext()` (*mutapath.MutaPath* method), 75  
`stripext()` (*mutapath.Path* method), 36  
`suffix()` (*mutapath.MutaPath* property), 84  
`suffix()` (*mutapath.Path* property), 46  
`suffixes()` (*mutapath.MutaPath* property), 84  
`suffixes()` (*mutapath.Path* property), 46  
`swapcase()` (*mutapath.MutaPath* method), 75  
`swapcase()` (*mutapath.Path* method), 36  
`symlink()` (*mutapath.MutaPath* method), 75  
`symlink()` (*mutapath.Path* method), 37  
`symlink_to()` (*mutapath.MutaPath* method), 75  
`symlink_to()` (*mutapath.Path* method), 37

## T

`text()` (*mutapath.MutaPath* method), 75  
`text()` (*mutapath.Path* method), 37  
`timeout()` (*mutapath.lock\_dummy.DummyFileLock* property), 85  
`title()` (*mutapath.MutaPath* method), 76

`title()` (*mutapath.Path* method), 37  
`to_pathlib()` (*mutapath.MutaPath* property), 84  
`to_pathlib()` (*mutapath.Path* property), 46  
`touch()` (*mutapath.MutaPath* method), 76  
`touch()` (*mutapath.Path* method), 37  
`translate()` (*mutapath.MutaPath* method), 76  
`translate()` (*mutapath.Path* method), 37

## U

`unlink()` (*mutapath.MutaPath* method), 76  
`unlink()` (*mutapath.Path* method), 38  
`unlink_p()` (*mutapath.MutaPath* method), 76  
`unlink_p()` (*mutapath.Path* method), 38  
`upper()` (*mutapath.MutaPath* method), 76  
`upper()` (*mutapath.Path* method), 38  
`using_module()` (*mutapath.MutaPath* method), 76  
`using_module()` (*mutapath.Path* method), 38  
`utime()` (*mutapath.MutaPath* method), 77  
`utime()` (*mutapath.Path* method), 38

## W

`walk()` (*mutapath.MutaPath* method), 77  
`walk()` (*mutapath.Path* method), 38  
`walkdirs()` (*mutapath.MutaPath* method), 77  
`walkdirs()` (*mutapath.Path* method), 38  
`walkfiles()` (*mutapath.MutaPath* method), 77  
`walkfiles()` (*mutapath.Path* method), 39  
`with_base()` (*mutapath.MutaPath* method), 77  
`with_base()` (*mutapath.Path* method), 39  
`with_name()` (*mutapath.MutaPath* method), 78  
`with_name()` (*mutapath.Path* method), 39  
`with_parent()` (*mutapath.MutaPath* method), 78  
`with_parent()` (*mutapath.Path* method), 39  
`with_poxis_enabled()` (*mutapath.MutaPath* method), 78  
`with_poxis_enabled()` (*mutapath.Path* method), 39  
`with_stem()` (*mutapath.MutaPath* method), 78  
`with_stem()` (*mutapath.Path* method), 39  
`with_string_repr_enabled()` (*mutapath.MutaPath* method), 78  
`with_string_repr_enabled()` (*mutapath.Path* method), 40  
`with_suffix()` (*mutapath.MutaPath* method), 78  
`with_suffix()` (*mutapath.Path* method), 40  
`write_bytes()` (*mutapath.MutaPath* method), 79  
`write_bytes()` (*mutapath.Path* method), 40  
`write_lines()` (*mutapath.MutaPath* method), 79  
`write_lines()` (*mutapath.Path* method), 40  
`write_text()` (*mutapath.MutaPath* method), 79  
`write_text()` (*mutapath.Path* method), 41

## Z

`zfill()` (*mutapath.MutaPath* method), 80