NAME

Path::Tiny – File path utility

VERSION

version 0.108

SYNOPSIS

```
use Path::Tiny;

# creating Path::Tiny objects

$dir = path("/tmp");
$foo = path("foo.txt");

$subdir = $dir->child("foo");
$bar = $subdir->child("bar.txt");

# stringifies as cleaned up path

$file = path("./foo.txt");
print $file; # "foo.txt"

# reading files

$guts = $file->slurp;
$guts = $file->slurp_utf8;

@lines = $file->lines;
@lines = $file->lines_utf8;

($head) = $file->lines( {count => 1} );
($tail) = $file->lines( {count => -1} );

# writing files

$bar->spew( @data );
$bar->spew_utf8( @data );

# reading directories

for ( $dir->children ) { ... }

$iter = $dir->iterator;
while ( my $next = $iter->() ) { ... }
```

DESCRIPTION

This module provides a small, fast utility for working with file paths. It is friendlier to use than File::Spec and provides easy access to functions from several other core file handling modules. It aims to be smaller and faster than many alternatives on CPAN, while helping people do many common things in consistent and less error-prone ways.

Path::Tiny does not try to work for anything except Unix-like and Win32 platforms. Even then, it might break if you try something particularly obscure or tortuous. (Quick! What does this mean: `///../../..//./././a//b/../c/./.`? And how does it differ on Win32?)

All paths are forced to have Unix-style forward slashes. Stringifying the object gives you back the path (after some clean up).

File input/output methods `flock` handles before reading or writing, as appropriate (if supported by the platform and/or filesystem).

The `*_utf8` methods (`slurp_utf8`, `lines_utf8`, etc.) operate in raw mode. On Windows, that means they will not have CRLF translation from the `:crlf` IO layer. Installing Unicode::UTF8 0.58 or later will speed up `*_utf8` situations in many cases and is highly recommended. Alternatively, installing PerlIO::utf8_strict 0.003 or later will be used in place of the default `:encoding(UTF-8)`.

This module depends heavily on PerlIO layers for correct operation and thus requires Perl 5.008001 or later.

CONSTRUCTORS

path

```
$path = path("foo/bar");
$path = path("/tmp", "file.txt"); # list
$path = path(".");                # cwd
$path = path("~user/file.txt");   # tilde processing
```

Constructs a `Path::Tiny` object. It doesn't matter if you give a file or directory path. It's still up to you to call directory-like methods only on directories and file-like methods only on files. This function is exported automatically by default.

The first argument must be defined and have non-zero length or an exception will be thrown. This prevents subtle, dangerous errors with code like `path( maybe_undef() )->remove_tree`.

If the first component of the path is a tilde ('~') then the component will be replaced with the output of `glob('~')`. If the first component of the path is a tilde followed by a user name then the component will be replaced with output of `glob('~username')`. Behaviour for non-existent users depends on the output of `glob` on the system.

On Windows, if the path consists of a drive identifier without a path component (`C:` or `D:`), it will be expanded to the absolute path of the current directory on that volume using `Cwd::getdcwd()`.

If called with a single `Path::Tiny` argument, the original is returned unless the original is holding a temporary file or directory reference in which case a stringified copy is made.

```
$path = path("foo/bar");
$temp = Path::Tiny->tempfile;

$p2 = path($path); # like $p2 = $path
$t2 = path($temp); # like $t2 = path( "$temp" )
```

This optimizes copies without proliferating references unexpectedly if a copy is made by code outside your control.

Current API available since 0.017.

new

```
$path = Path::Tiny->new("foo/bar");
```

This is just like `path`, but with method call overhead. (Why would you do that?)

Current API available since 0.001.

cwd

```
$path = Path::Tiny->cwd; # path( Cwd::getcwd )
$path = cwd; # optional export
```

Gives you the absolute path to the current directory as a `Path::Tiny` object. This is slightly faster than `path(".")->absolute`.

`cwd` may be exported on request and used as a function instead of as a method.

Current API available since 0.018.

rootdir

```
$path = Path::Tiny->rootdir; # /
$path = rootdir;             # optional export
```

Gives you `File::Spec->rootdir` as a `Path::Tiny` object if you're too picky for `path("/")`.

`rootdir` may be exported on request and used as a function instead of as a method.

Current API available since 0.018.

tempfile, tempdir

```
$temp = Path::Tiny->tempfile( @options );
$temp = Path::Tiny->tempdir( @options );
$temp = tempfile( @options ); # optional export
$temp = tempdir( @options );  # optional export
```

`tempfile` passes the options to `File::Temp->new` and returns a `Path::Tiny` object with the file name. The `TMPDIR` option is enabled by default.

The resulting `File::Temp` object is cached. When the `Path::Tiny` object is destroyed, the `File::Temp` object will be as well.

`File::Temp` annoyingly requires you to specify a custom template in slightly different ways depending on which function or method you call, but `Path::Tiny` lets you ignore that and can take either a leading template or a `TEMPLATE` option and does the right thing.

```
$temp = Path::Tiny->tempfile( "customXXXXXXXX" );             # ok
$temp = Path::Tiny->tempfile( TEMPLATE => "customXXXXXXXX" ); # ok
```

The tempfile path object will be normalized to have an absolute path, even if created in a relative directory using `DIR`. If you want it to have the `realpath` instead, pass a leading options hash like this:

```
$real_temp = tempfile({realpath => 1}, @options);
```

`tempdir` is just like `tempfile`, except it calls `File::Temp->newdir` instead.

Both `tempfile` and `tempdir` may be exported on request and used as functions instead of as methods.

Note: for tempfiles, the filehandles from File::Temp are closed and not reused. This is not as secure as using File::Temp handles directly, but is less prone to deadlocks or access problems on some platforms. Think of what `Path::Tiny` gives you to be just a temporary file **name** that gets cleaned up.

Note 2: if you don't want these cleaned up automatically when the object is destroyed, File::Temp requires different options for directories and files. Use `CLEANUP => 0` for directories and `UNLINK => 0` for files.

Note 3: Don't lose the temporary object by chaining a method call instead of storing it:

```
my $lost = tempdir()->child("foo"); # tempdir cleaned up right away
```

Note 4: The cached object may be accessed with the ''cached_temp'' method. Keeping a reference to, or modifying the cached object may break the behavior documented above and is not supported. Use at your own risk.

Current API available since 0.097.

METHODS

absolute

```
$abs = path("foo/bar")->absolute;
$abs = path("foo/bar")->absolute("/tmp");
```

Returns a new `Path::Tiny` object with an absolute path (or itself if already absolute). If no argument is given, the current directory is used as the absolute base path. If an argument is given, it will be converted to an absolute path (if it is not already) and used as the absolute base path.

This will not resolve upward directories (''foo/../bar'') unless `canonpath` in File::Spec would normally do so on your platform. If you need them resolved, you must call the more expensive `realpath` method

instead.

On Windows, an absolute path without a volume component will have it added based on the current drive.

Current API available since 0.101.

append, append_raw, append_utf8

```
path("foo.txt")->append(@data);
path("foo.txt")->append(\@data);
path("foo.txt")->append({binmode => ":raw"}, @data);
path("foo.txt")->append_raw(@data);
path("foo.txt")->append_utf8(@data);
```

Appends data to a file. The file is locked with `flock` prior to writing and closed afterwards. An optional hash reference may be used to pass options. Valid options are:

- `binmode`: passed to `binmode()` on the handle used for writing.
- `truncate`: truncates the file after locking and before appending

The `truncate` option is a way to replace the contents of a file **in place**, unlike "spew" which writes to a temporary file and then replaces the original (if it exists).

`append_raw` is like `append` with a `binmode` of `:unix` for fast, unbuffered, raw write.

`append_utf8` is like `append` with a `binmode` of `:unix:encoding(UTF-8)` (or PerlIO::utf8_strict). If Unicode::UTF8 0.58+ is installed, a raw append will be done instead on the data encoded with `Unicode::UTF8`.

Current API available since 0.060.

assert

```
$path = path("foo.txt")->assert( sub { $_->exists } );
```

Returns the invocant after asserting that a code reference argument returns true. When the assertion code reference runs, it will have the invocant object in the `$_` variable. If it returns false, an exception will be thrown. The assertion code reference may also throw its own exception.

If no assertion is provided, the invocant is returned without error.

Current API available since 0.062.

basename

```
$name = path("foo/bar.txt")->basename;         # bar.txt
$name = path("foo.txt")->basename('.txt');     # foo
$name = path("foo.txt")->basename(qr/.txt/);   # foo
$name = path("foo.txt")->basename(@suffixes);
```

Returns the file portion or last directory portion of a path.

Given a list of suffixes as strings or regular expressions, any that match at the end of the file portion or last directory portion will be removed before the result is returned.

Current API available since 0.054.

canonpath

```
$canonical = path("foo/bar")->canonpath; # foo\bar on Windows
```

Returns a string with the canonical format of the path name for the platform. In particular, this means directory separators will be `\` on Windows.

Current API available since 0.001.

cached_temp

Returns the cached `File::Temp` or `File::Temp::Dir` object if the `Path::Tiny` object was created with `/tempfile` or `/tempdir`. If there is no such object, this method throws.

WARNING: Keeping a reference to, or modifying the cached object may break the behavior documented for

temporary files and directories created with `Path::Tiny` and is not supported. Use at your own risk.

Current API available since 0.101.

child

```
$file = path("/tmp")->child("foo.txt"); # "/tmp/foo.txt"
$file = path("/tmp")->child(@parts);
```

Returns a new `Path::Tiny` object relative to the original. Works like `catfile` or `catdir` from File::Spec, but without caring about file or directories.

WARNING: because the argument could contain `..` or refer to symlinks, there is no guarantee that the new path refers to an actual descendent of the original. If this is important to you, transform parent and child with "realpath" and check them with "subsumes".

Current API available since 0.001.

children

```
@paths = path("/tmp")->children;
@paths = path("/tmp")->children( qr/\.txt$/ );
```

Returns a list of `Path::Tiny` objects for all files and directories within a directory. Excludes "." and ".." automatically.

If an optional `qr//` argument is provided, it only returns objects for child names that match the given regular expression. Only the base name is used for matching:

```
@paths = path("/tmp")->children( qr/^foo/ );
# matches children like the glob foo*
```

Current API available since 0.028.

chmod

```
path("foo.txt")->chmod(0777);
path("foo.txt")->chmod("0755");
path("foo.txt")->chmod("go-w");
path("foo.txt")->chmod("a=r,u+wx");
```

Sets file or directory permissions. The argument can be a numeric mode, a octal string beginning with a "0" or a limited subset of the symbolic mode use by */bin/chmod*.

The symbolic mode must be a comma-delimited list of mode clauses. Clauses must match `qr/\A([augo]+)([=+-])([rwx]+)\z/`, which defines "who", "op" and "perms" parameters for each clause. Unlike */bin/chmod*, all three parameters are required for each clause, multiple ops are not allowed and permissions `stugoX` are not supported. (See File::chmod for more complex needs.)

Current API available since 0.053.

copy

```
path("/tmp/foo.txt")->copy("/tmp/bar.txt");
```

Copies the current path to the given destination using File::Copy's `copy` function. Upon success, returns the `Path::Tiny` object for the newly copied file.

Current API available since 0.070.

digest

```
$obj = path("/tmp/foo.txt")->digest;        # SHA-256
$obj = path("/tmp/foo.txt")->digest("MD5"); # user-selected
$obj = path("/tmp/foo.txt")->digest( { chunk_size => 1e6 }, "MD5" );
```

Returns a hexadecimal digest for a file. An optional hash reference of options may be given. The only option is `chunk_size`. If `chunk_size` is given, that many bytes will be read at a time. If not provided, the entire file will be slurped into memory to compute the digest.

Any subsequent arguments are passed to the constructor for Digest to select an algorithm. If no arguments

are given, the default is SHA-256.

Current API available since 0.056.

dirname (deprecated)

```
$name = path("/tmp/foo.txt")->dirname; # "/tmp/"
```

Returns the directory portion you would get from calling `File::Spec->splitpath( $path->stringify )` or `"."` for a path without a parent directory portion. Because File::Spec is inconsistent, the result might or might not have a trailing slash. Because of this, this method is **deprecated**.

A better, more consistently approach is likely `$path->parent->stringify`, which will not have a trailing slash except for a root directory.

Deprecated in 0.056.

edit, edit_raw, edit_utf8

```
path("foo.txt")->edit( \&callback, $options );
path("foo.txt")->edit_utf8( \&callback );
path("foo.txt")->edit_raw( \&callback );
```

These are convenience methods that allow "editing" a file using a single callback argument. They slurp the file using `slurp`, place the contents inside a localized `$_` variable, call the callback function (without arguments), and then write `$_` (presumably mutated) back to the file with `spew`.

An optional hash reference may be used to pass options. The only option is `binmode`, which is passed to `slurp` and `spew`.

`edit_utf8` and `edit_raw` act like their respective `slurp_*` and `spew_*` methods.

Current API available since 0.077.

edit_lines, edit_lines_utf8, edit_lines_raw

```
path("foo.txt")->edit_lines( \&callback, $options );
path("foo.txt")->edit_lines_utf8( \&callback );
path("foo.txt")->edit_lines_raw( \&callback );
```

These are convenience methods that allow "editing" a file's lines using a single callback argument. They iterate over the file: for each line, the line is put into a localized `$_` variable, the callback function is executed (without arguments) and then `$_` is written to a temporary file. When iteration is finished, the temporary file is atomically renamed over the original.

An optional hash reference may be used to pass options. The only option is `binmode`, which is passed to the method that open handles for reading and writing.

`edit_lines_utf8` and `edit_lines_raw` act like their respective `slurp_*` and `spew_*` methods.

Current API available since 0.077.

exists, is_file, is_dir

```
if ( path("/tmp")->exists ) { ... }      # -e
if ( path("/tmp")->is_dir ) { ... }      # -d
if ( path("/tmp")->is_file ) { ... }     # -e && ! -d
```

Implements file test operations, this means the file or directory actually has to exist on the filesystem. Until then, it's just a path.

Note: `is_file` is not `-f` because `-f` is not the opposite of `-d`. `-f` means "plain file", excluding symlinks, devices, etc. that often can be read just like files.

Use `-f` instead if you really mean to check for a plain file.

Current API available since 0.053.

filehandle

```
$fh = path("/tmp/foo.txt")->filehandle($mode, $binmode);
$fh = path("/tmp/foo.txt")->filehandle({ locked => 1 }, $mode, $binmode);
$fh = path("/tmp/foo.txt")->filehandle({ exclusive => 1  }, $mode, $binmode);
```

Returns an open file handle. The `$mode` argument must be a Perl-style read/write mode string ("<" ,">", ">>", etc.). If a `$binmode` is given, it is set during the `open` call.

An optional hash reference may be used to pass options.

The `locked` option governs file locking; if true, handles opened for writing, appending or read-write are locked with `LOCK_EX`; otherwise, they are locked with `LOCK_SH`. When using `locked`, ">" or "+>" modes will delay truncation until after the lock is acquired.

The `exclusive` option causes the **open()** call to fail if the file already exists. This corresponds to the O_EXCL flag to sysopen / **open** (2). `exclusive` implies `locked` and will set it for you if you forget it.

See `openr`, `openw`, `openrw`, and `opena` for sugar.

Current API available since 0.066.

is_absolute, is_relative

```
if ( path("/tmp")->is_absolute ) { ... }
if ( path("/tmp")->is_relative ) { ... }
```

Booleans for whether the path appears absolute or relative.

Current API available since 0.001.

is_rootdir

```
while ( ! $path->is_rootdir ) {
    $path = $path->parent;
    ...
}
```

Boolean for whether the path is the root directory of the volume. I.e. the `dirname` is `q[/]` and the `basename` is `q[]`.

This works even on `MSWin32` with drives and UNC volumes:

```
path("C:/")->is_rootdir;              # true
path("//server/share/")->is_rootdir; #true
```

Current API available since 0.038.

iterator

```
$iter = path("/tmp")->iterator( \%options );
```

Returns a code reference that walks a directory lazily. Each invocation returns a `Path::Tiny` object or undef when the iterator is exhausted.

```
$iter = path("/tmp")->iterator;
while ( $path = $iter->() ) {
    ...
}
```

The current and parent directory entries ("." and "..") will not be included.

If the `recurse` option is true, the iterator will walk the directory recursively, breadth-first. If the `follow_symlinks` option is also true, directory links will be followed recursively. There is no protection against loops when following links. If a directory is not readable, it will not be followed.

The default is the same as:

```
$iter = path("/tmp")->iterator( {
    recurse         => 0,
    follow_symlinks => 0,
} );
```

For a more powerful, recursive iterator with built-in loop avoidance, see Path::Iterator::Rule.

See also "visit".

Current API available since 0.016.

lines, lines_raw, lines_utf8

```
@contents = path("/tmp/foo.txt")->lines;
@contents = path("/tmp/foo.txt")->lines(\%options);
@contents = path("/tmp/foo.txt")->lines_raw;
@contents = path("/tmp/foo.txt")->lines_utf8;

@contents = path("/tmp/foo.txt")->lines( { chomp => 1, count => 4 } );
```

Returns a list of lines from a file. Optionally takes a hash-reference of options. Valid options are `binmode`, `count` and `chomp`.

If `binmode` is provided, it will be set on the handle prior to reading.

If a positive `count` is provided, that many lines will be returned from the start of the file. If a negative `count` is provided, the entire file will be read, but only `abs(count)` will be kept and returned. If `abs(count)` exceeds the number of lines in the file, all lines will be returned.

If `chomp` is set, any end-of-line character sequences (`CR`, `CRLF`, or `LF`) will be removed from the lines returned.

Because the return is a list, `lines` in scalar context will return the number of lines (and throw away the data).

```
$number_of_lines = path("/tmp/foo.txt")->lines;
```

`lines_raw` is like `lines` with a `binmode` of `:raw`. We use `:raw` instead of `:unix` so PerlIO buffering can manage reading by line.

`lines_utf8` is like `lines` with a `binmode` of `:raw:encoding(UTF-8)` (or PerlIO::utf8_strict). If Unicode::UTF8 0.58+ is installed, a raw UTF-8 slurp will be done and then the lines will be split. This is actually faster than relying on `:encoding(UTF-8)`, though a bit memory intensive. If memory use is a concern, consider `openr_utf8` and iterating directly on the handle.

Current API available since 0.065.

mkpath

```
path("foo/bar/baz")->mkpath;
path("foo/bar/baz")->mkpath( \%options );
```

Like calling `make_path` from File::Path. An optional hash reference is passed through to `make_path`. Errors will be trapped and an exception thrown. Returns the list of directories created or an empty list if the directories already exist, just like `make_path`.

Current API available since 0.001.

move

```
path("foo.txt")->move("bar.txt");
```

Move the current path to the given destination path using Perl's built-in rename function. Returns the result of the `rename` function (except it throws an exception if it fails).

Current API available since 0.001.

openr, openw, openrw, opena

```
$fh = path("foo.txt")->openr($binmode);  # read
$fh = path("foo.txt")->openr_raw;
$fh = path("foo.txt")->openr_utf8;

$fh = path("foo.txt")->openw($binmode);  # write
$fh = path("foo.txt")->openw_raw;
$fh = path("foo.txt")->openw_utf8;

$fh = path("foo.txt")->opena($binmode);  # append
$fh = path("foo.txt")->opena_raw;
$fh = path("foo.txt")->opena_utf8;

$fh = path("foo.txt")->openrw($binmode); # read/write
$fh = path("foo.txt")->openrw_raw;
$fh = path("foo.txt")->openrw_utf8;
```

Returns a file handle opened in the specified mode. The `openr` style methods take a single `binmode` argument. All of the `open*` methods have `open*_raw` and `open*_utf8` equivalents that use `:raw` and `:raw:encoding(UTF-8)`, respectively.

An optional hash reference may be used to pass options. The only option is `locked`. If true, handles opened for writing, appending or read-write are locked with `LOCK_EX`; otherwise, they are locked for `LOCK_SH`.

```
$fh = path("foo.txt")->openrw_utf8( { locked => 1 } );
```

See "filehandle" for more on locking.

Current API available since 0.011.

parent

```
$parent = path("foo/bar/baz")->parent; # foo/bar
$parent = path("foo/wibble.txt")->parent; # foo

$parent = path("foo/bar/baz")->parent(2); # foo
```

Returns a `Path::Tiny` object corresponding to the parent directory of the original directory or file. An optional positive integer argument is the number of parent directories upwards to return. `parent` by itself is equivalent to `parent(1)`.

Current API available since 0.014.

realpath

```
$real = path("/baz/foo/../bar")->realpath;
$real = path("foo/../bar")->realpath;
```

Returns a new `Path::Tiny` object with all symbolic links and upward directory parts resolved using Cwd's `realpath`. Compared to `absolute`, this is more expensive as it must actually consult the filesystem.

If the parent path can't be resolved (e.g. if it includes directories that don't exist), an exception will be thrown:

```
$real = path("doesnt_exist/foo")->realpath; # dies
```

However, if the parent path exists and only the last component (e.g. filename) doesn't exist, the realpath will be the realpath of the parent plus the non-existent last component:

```
$real = path("./aasdlfasdlf")->realpath; # works
```

The underlying Cwd module usually worked this way on Unix, but died on Windows (and some Unixes) if the full path didn't exist. As of version 0.064, it's safe to use anywhere.

Current API available since 0.001.

relative

```
$rel = path("/tmp/foo/bar")->relative("/tmp"); # foo/bar
```

Returns a `Path::Tiny` object with a path relative to a new base path given as an argument. If no argument is given, the current directory will be used as the new base path.

If either path is already relative, it will be made absolute based on the current directly before determining the new relative path.

The algorithm is roughly as follows:

- If the original and new base path are on different volumes, an exception will be thrown.
- If the original and new base are identical, the relative path is `"."`.
- If the new base subsumes the original, the relative path is the original path with the new base chopped off the front
- If the new base does not subsume the original, a common prefix path is determined (possibly the root directory) and the relative path will consist of updirs (`".."`) to reach the common prefix, followed by the original path less the common prefix.

Unlike `File::Spec::abs2rel`, in the last case above, the calculation based on a common prefix takes into account symlinks that could affect the updir process. Given an original path "/A/B" and a new base "/A/C", (where "A", "B" and "C" could each have multiple path components):

- Symlinks in "A" don't change the result unless the last component of A is a symlink and the first component of "C" is an updir.
- Symlinks in "B" don't change the result and will exist in the result as given.
- Symlinks and updirs in "C" must be resolved to actual paths, taking into account the possibility that not all path components might exist on the filesystem.

Current API available since 0.001. New algorithm (that accounts for symlinks) available since 0.079.

remove

```
path("foo.txt")->remove;
```

This is just like `unlink`, except for its error handling: if the path does not exist, it returns false; if deleting the file fails, it throws an exception.

Current API available since 0.012.

remove_tree

```
# directory
path("foo/bar/baz")->remove_tree;
path("foo/bar/baz")->remove_tree( \%options );
path("foo/bar/baz")->remove_tree( { safe => 0 } ); # force remove
```

Like calling `remove_tree` from File::Path, but defaults to `safe` mode. An optional hash reference is passed through to `remove_tree`. Errors will be trapped and an exception thrown. Returns the number of directories deleted, just like `remove_tree`.

If you want to remove a directory only if it is empty, use the built-in `rmdir` function instead.

```
rmdir path("foo/bar/baz/");
```

Current API available since 0.013.

sibling

```
$foo = path("/tmp/foo.txt");
$sib = $foo->sibling("bar.txt");         # /tmp/bar.txt
$sib = $foo->sibling("baz", "bam.txt"); # /tmp/baz/bam.txt
```

Returns a new `Path::Tiny` object relative to the parent of the original. This is slightly more efficient

than `$path->parent->child(...)`.

Current API available since 0.058.

slurp, slurp_raw, slurp_utf8

```
$data = path("foo.txt")->slurp;
$data = path("foo.txt")->slurp( {binmode => ":raw"} );
$data = path("foo.txt")->slurp_raw;
$data = path("foo.txt")->slurp_utf8;
```

Reads file contents into a scalar. Takes an optional hash reference which may be used to pass options. The only available option is `binmode`, which is passed to `binmode()` on the handle used for reading.

`slurp_raw` is like `slurp` with a `binmode` of `:unix` for a fast, unbuffered, raw read.

`slurp_utf8` is like `slurp` with a `binmode` of `:unix:encoding(UTF-8)` (or PerlIO::utf8_strict). If Unicode::UTF8 0.58+ is installed, a raw slurp will be done instead and the result decoded with `Unicode::UTF8`. This is just as strict and is roughly an order of magnitude faster than using `:encoding(UTF-8)`.

Note: `slurp` and friends lock the filehandle before slurping. If you plan to slurp from a file created with File::Temp, be sure to close other handles or open without locking to avoid a deadlock:

```
my $tempfile = File::Temp->new(EXLOCK => 0);
my $guts = path($tempfile)->slurp;
```

Current API available since 0.004.

spew, spew_raw, spew_utf8

```
path("foo.txt")->spew(@data);
path("foo.txt")->spew(\@data);
path("foo.txt")->spew({binmode => ":raw"}, @data);
path("foo.txt")->spew_raw(@data);
path("foo.txt")->spew_utf8(@data);
```

Writes data to a file atomically. The file is written to a temporary file in the same directory, then renamed over the original. An optional hash reference may be used to pass options. The only option is `binmode`, which is passed to `binmode()` on the handle used for writing.

`spew_raw` is like `spew` with a `binmode` of `:unix` for a fast, unbuffered, raw write.

`spew_utf8` is like `spew` with a `binmode` of `:unix:encoding(UTF-8)` (or PerlIO::utf8_strict). If Unicode::UTF8 0.58+ is installed, a raw spew will be done instead on the data encoded with `Unicode::UTF8`.

NOTE: because the file is written to a temporary file and then renamed, the new file will wind up with permissions based on your current umask. This is a feature to protect you from a race condition that would otherwise give different permissions than you might expect. If you really want to keep the original mode flags, use ''append'' with the `truncate` option.

Current API available since 0.011.

stat, lstat

```
$stat = path("foo.txt")->stat;
$stat = path("/some/symlink")->lstat;
```

Like calling `stat` or `lstat` from File::stat.

Current API available since 0.001.

stringify

```
$path = path("foo.txt");
say $path->stringify; # same as "$path"
```

Returns a string representation of the path. Unlike `canonpath`, this method returns the path standardized with Unix-style `/` directory separators.

Current API available since 0.001.

subsumes

```
path("foo/bar")->subsumes("foo/bar/baz"); # true
path("/foo/bar")->subsumes("/foo/baz");   # false
```

Returns true if the first path is a prefix of the second path at a directory boundary.

This **does not** resolve parent directory entries (`..`) or symlinks:

```
path("foo/bar")->subsumes("foo/bar/../baz"); # true
```

If such things are important to you, ensure that both paths are resolved to the filesystem with `realpath`:

```
my $p1 = path("foo/bar")->realpath;
my $p2 = path("foo/bar/../baz")->realpath;
if ( $p1->subsumes($p2) ) { ... }
```

Current API available since 0.048.

touch

```
path("foo.txt")->touch;
path("foo.txt")->touch($epoch_secs);
```

Like the Unix `touch` utility. Creates the file if it doesn't exist, or else changes the modification and access times to the current time. If the first argument is the epoch seconds then it will be used.

Returns the path object so it can be easily chained with other methods:

```
# won't die if foo.txt doesn't exist
$content = path("foo.txt")->touch->slurp;
```

Current API available since 0.015.

touchpath

```
path("bar/baz/foo.txt")->touchpath;
```

Combines `mkpath` and `touch`. Creates the parent directory if it doesn't exist, before touching the file. Returns the path object like `touch` does.

Current API available since 0.022.

visit

```
path("/tmp")->visit( \&callback, \%options );
```

Executes a callback for each child of a directory. It returns a hash reference with any state accumulated during iteration.

The options are the same as for "iterator" (which it uses internally): `recurse` and `follow_symlinks`. Both default to false.

The callback function will receive a `Path::Tiny` object as the first argument and a hash reference to accumulate state as the second argument. For example:

```
# collect files sizes
my $sizes = path("/tmp")->visit(
    sub {
        my ($path, $state) = @_;
        return if $path->is_dir;
        $state->{$path} = -s $path;
    },
    { recurse => 1 }
);
```

For convenience, the `Path::Tiny` object will also be locally aliased as the `$_` global variable:

```
# print paths matching /foo/
path("/tmp")->visit( sub { say if /foo/ }, { recurse => 1} );
```

If the callback returns a **reference** to a false scalar value, iteration will terminate. This is not the same as "pruning" a directory search; this just stops all iteration and returns the state hash reference.

```
# find up to 10 files larger than 100K
my $files = path("/tmp")->visit(
    sub {
        my ($path, $state) = @_;
        $state->{$path}++ if -s $path > 102400
        return \0 if keys %$state == 10;
    },
    { recurse => 1 }
);
```

If you want more flexible iteration, use a module like Path::Iterator::Rule.

Current API available since 0.062.

volume

```
$vol = path("/tmp/foo.txt")->volume;   # ""
$vol = path("C:/tmp/foo.txt")->volume; # "C:"
```

Returns the volume portion of the path. This is equivalent to what File::Spec would give from `splitpath` and thus usually is the empty string on Unix-like operating systems or the drive letter for an absolute path on `MSWin32`.

Current API available since 0.001.

EXCEPTION HANDLING

Simple usage errors will generally croak. Failures of underlying Perl functions will be thrown as exceptions in the class `Path::Tiny::Error`.

A `Path::Tiny::Error` object will be a hash reference with the following fields:

- `op` — a description of the operation, usually function call and any extra info
- `file` — the file or directory relating to the error
- `err` — hold `$!` at the time the error was thrown
- `msg` — a string combining the above data and a Carp-like short stack trace

Exception objects will stringify as the `msg` field.

ENVIRONMENT

PERL_PATH_TINY_NO_FLOCK

If the environment variable `PERL_PATH_TINY_NO_FLOCK` is set to a true value then flock will NOT be used when accessing files (this is not recommended).

CAVEATS

Subclassing not supported

For speed, this class is implemented as an array based object and uses many direct function calls internally. You must not subclass it and expect things to work properly.

File locking

If flock is not supported on a platform, it will not be used, even if locking is requested.

In situations where a platform normally would support locking, but the flock fails due to a filesystem limitation, Path::Tiny has some heuristics to detect this and will warn once and continue in an unsafe mode. If you want this failure to be fatal, you can fatalize the 'flock' warnings category:

```
use warnings FATAL => 'flock';
```

See additional caveats below.

NFS and BSD

On BSD, Perl's flock implementation may not work to lock files on an NFS filesystem. If detected, this situation will warn once, as described above.

Lustre

The Lustre filesystem does not support flock. If detected, this situation will warn once, as described above.

AIX and locking

AIX requires a write handle for locking. Therefore, calls that normally open a read handle and take a shared lock instead will open a read-write handle and take an exclusive lock. If the user does not have write permission, no lock will be used.

utf8 vs UTF-8

All the `*_utf8` methods by default use `:encoding(UTF-8)` — either as `:unix:encoding(UTF-8)` (unbuffered) or `:raw:encoding(UTF-8)` (buffered) — which is strict against the Unicode spec and disallows illegal Unicode codepoints or UTF-8 sequences.

Unfortunately, `:encoding(UTF-8)` is very, very slow. If you install Unicode::UTF8 0.58 or later, that module will be used by some `*_utf8` methods to encode or decode data after a raw, binary input/output operation, which is much faster. Alternatively, if you install PerlIO::utf8_strict, that will be used instead of `:encoding(UTF-8)` and is also very fast.

If you need the performance and can accept the security risk, `slurp({binmode => ":unix:utf8"})` will be faster than `:unix:encoding(UTF-8)` (but not as fast as `Unicode::UTF8`).

Note that the `*_utf8` methods read in **raw** mode. There is no CRLF translation on Windows. If you must have CRLF translation, use the regular input/output methods with an appropriate binmode:

```
$path->spew_utf8($data);                            # raw
$path->spew({binmode => ":encoding(UTF-8)"}, $data; # LF -> CRLF
```

Default IO layers and the open pragma

If you have Perl 5.10 or later, file input/output methods (`slurp`, `spew`, etc.) and high-level handle opening methods (`filehandle`, `openr`, `openw`, etc.) respect default encodings set by the `-C` switch or lexical open settings of the caller. For UTF-8, this is almost certainly slower than using the dedicated `_utf8` methods if you have Unicode::UTF8.

TYPE CONSTRAINTS AND COERCION

A standard MooseX::Types library is available at MooseX::Types::Path::Tiny. A Type::Tiny equivalent is available as Types::Path::Tiny.

SEE ALSO

These are other file/path utilities, which may offer a different feature set than `Path::Tiny`.

- File::chmod
- File::Fu
- IO::All
- Path::Class

These iterators may be slightly faster than the recursive iterator in `Path::Tiny`:

- Path::Iterator::Rule
- File::Next

There are probably comparable, non-Tiny tools. Let me know if you want me to add a module to the list.

This module was featured in the 2013 Perl Advent Calendar <http://www.perladvent.org/2013/2013-12-18.html>.

SUPPORT

Bugs / Feature Requests

Please report any bugs or feature requests through the issue tracker at <https://github.com/dagolden/Path-Tiny/issues>. You will be notified automatically of any progress on your issue.

Source Code

This is open source software. The code repository is available for public review and contribution under the terms of the license.

<https://github.com/dagolden/Path-Tiny>

```
git clone https://github.com/dagolden/Path-Tiny.git
```

AUTHOR

David Golden <dagolden@cpan.org>

CONTRIBUTORS

- Alex Efros <powerman@powerman.name>
- Aristotle Pagaltzis <pagaltzis@gmx.de>
- Chris Williams <bingos@cpan.org>
- Dave Rolsky <autarch@urth.org>
- David Steinbrunner <dsteinbrunner@pobox.com>
- Doug Bell <madcityzen@gmail.com>
- Gabor Szabo <szabgab@cpan.org>
- Gabriel Andrade <gabiruh@gmail.com>
- George Hartzell <hartzell@cpan.org>
- Geraud Continsouzas <geraud@scsi.nc>
- Goro Fuji <gfuji@cpan.org>
- Graham Knop <haarg@haarg.org>
- Graham Ollis <plicease@cpan.org>
- Ian Sillitoe <ian@sillit.com>
- James Hunt <james@niftylogic.com>
- John Karr <brainbuz@brainbuz.org>
- Karen Etheridge <ether@cpan.org>
- Mark Ellis <mark.ellis@cartridgesave.co.uk>
- Martin H. Sluka <fany@cpan.org>
- Martin Kjeldsen <mk@bluepipe.dk>
- Michael G. Schwern <mschwern@cpan.org>
- Nigel Gregoire <nigelgregoire@gmail.com>
- Philippe Bruhat (BooK) <book@cpan.org>
- Regina Verbae <regina-verbae@users.noreply.github.com>
- Roy Ivy III <rivy@cpan.org>
- Shlomi Fish <shlomif@shlomifish.org>
- Smylers <Smylers@stripey.com>
- Tatsuhiko Miyagawa <miyagawa@bulknews.net>

- Toby Inkster <tobyink@cpan.org>
- Yanick Champoux <yanick@babyl.dyndns.org>
- – Keedi Kim <keedi@cpan.org>

COPYRIGHT AND LICENSE