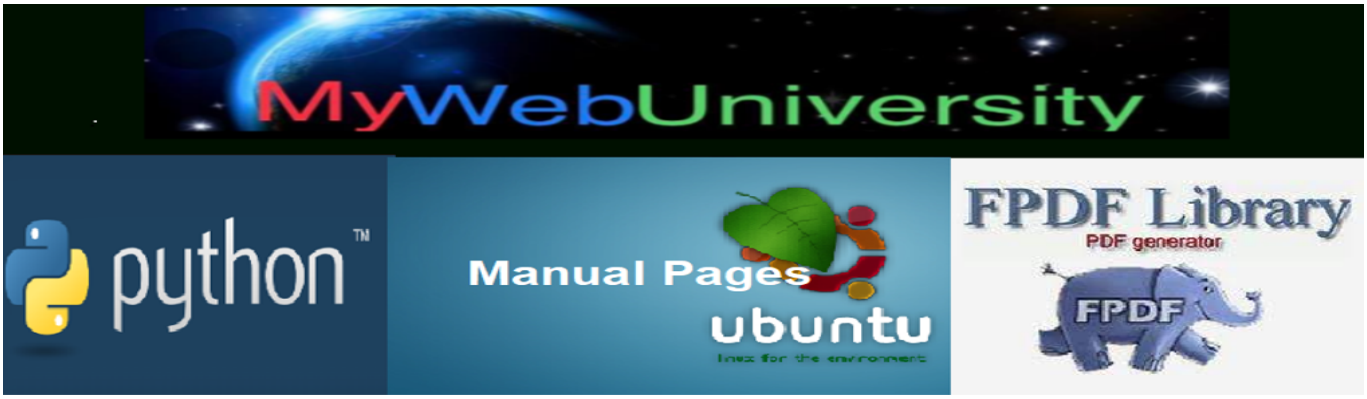

Full credit is given to the above companies including the Operating System (OS) that this PDF file was generated!

Rocky Enterprise Linux 9.2 Manual Pages on command 'Path::Class::File.3pm'

$ man Path::Class::File.3pm

Path::Class::File(3pm) User Contributed Perl Documentation Path::Class::File(3pm)

NAME

Path::Class::File - Objects representing files

VERSION

version 0.37

SYNOPSIS

```
use Path::Class;  # Exports file() by default
my $file = file('foo', 'bar.txt');  # Path::Class::File object
my $file = Path::Class::File->new('foo', 'bar.txt'); # Same thing
# Stringifies to 'foo/bar.txt' on Unix, 'foo\bar.txt' on Windows, etc.
print "file: $file\n";
if ($file->is_absolute) { ... }
if ($file->is_relative) { ... }
my $v = $file->volume; # Could be 'C:' on Windows, empty string
                       # on Unix, 'Macintosh HD:' on Mac OS
$file->cleanup; # Perform logical cleanup of pathname
$file->resolve; # Perform physical cleanup of pathname
my $dir = $file->dir;  # A Path::Class::Dir object
my $abs = $file->absolute; # Transform to absolute path
my $rel = $file->relative; # Transform to relative path
```

DESCRIPTION

The "Path::Class::File" class contains functionality for manipulating file names in a cross-platform way.

METHODS

$file = Path::Class::File->new(<dir1>, <dir2>, ..., <file>)

$file = file(<dir1>, <dir2>, ..., <file>)

Creates a new "Path::Class::File" object and returns it. The arguments specify the path to the file. Any volume may also be specified as the first argument, or as part of the first argument. You can use platform-neutral syntax:

```
my $file = file( 'foo', 'bar', 'baz.txt' );
```

or platform-native syntax:

```
my $file = file( 'foo/bar/baz.txt' );
```

or a mixture of the two:

```
my $file = file( 'foo/bar', 'baz.txt' );
```

All three of the above examples create relative paths. To create an absolute path, either use the platform native syntax for doing so:

```
my $file = file( '/var/tmp/foo.txt' );
```

or use an empty string as the first argument:

```
my $file = file( '', 'var', 'tmp', 'foo.txt' );
```

If the second form seems awkward, that's somewhat intentional - paths like "/var/tmp" or "\Windows" aren't cross-platform concepts in the first place, so they probably shouldn't appear in your code if you're trying to be cross-platform. The first form is perfectly fine, because paths like this may come from config files, user input, or whatever.

$file->stringify

This method is called internally when a "Path::Class::File" object is used in a string context, so the following are equivalent:

```
$string = $file->stringify;
$string = "$file";
```

$file->volume

Returns the volume (e.g. "C:" on Windows, "Macintosh HD:" on Mac OS, etc.) of the object, if any. Otherwise, returns the empty string.

$file->basename

Returns the name of the file as a string, without the directory portion (if any).

$file->components

Returns a list of the directory components of this file, followed by the basename.

Note: unlike "$dir->components", this method currently does not accept any arguments to select which elements of the list will be returned. It may do so in the future. Currently it throws an exception if such arguments are present.

$file->is_dir

Returns a boolean value indicating whether this object represents a directory. Not surprisingly, "Path::Class::File" objects always return false, and Path::Class::Dir objects always return true.

$file->is_absolute

Returns true or false depending on whether the file refers to an absolute path specifier (like "/usr/local/foo.txt" or "\Windows\Foo.txt").

$file->is_relative

Returns true or false depending on whether the file refers to a relative path specifier (like "lib/foo.txt" or ".\Foo.txt").

$file->cleanup

Performs a logical cleanup of the file path. For instance:

```
my $file = file('/foo//baz/./foo.txt')->cleanup;
# $file now represents '/foo/baz/foo.txt';
```

$dir->resolve

Performs a physical cleanup of the file path. For instance:

```
my $file = file('/foo/baz/../foo.txt')->resolve;
# $file now represents '/foo/foo.txt', assuming no symlinks
```

This actually consults the filesystem to verify the validity of the path.

$dir = $file->dir

Returns a "Path::Class::Dir" object representing the directory containing this file.

$dir = $file->parent

A synonym for the "dir()" method.

$abs = $file->absolute

Returns a "Path::Class::File" object representing $file as an absolute path. An optional argument, given as either a string or a Path::Class::Dir object, specifies the directory to use as the base of relativity - otherwise the current working directory will be used.

$rel = $file->relative

Returns a "Path::Class::File" object representing $file as a relative path. An

optional argument, given as either a string or a "Path::Class::Dir" object, specifies the directory to use as the base of relativity - otherwise the current working directory will be used.

$foreign = $file->as_foreign($type)

Returns a "Path::Class::File" object representing $file as it would be specified on a system of type $type. Known types include "Unix", "Win32", "Mac", "VMS", and "OS2", i.e. anything for which there is a subclass of "File::Spec".

Any generated objects (subdirectories, files, parents, etc.) will also retain this type.

$foreign = Path::Class::File->new_foreign($type, @args)

Returns a "Path::Class::File" object representing a file as it would be specified on a system of type $type. Known types include "Unix", "Win32", "Mac", "VMS", and "OS2", i.e. anything for which there is a subclass of "File::Spec".

The arguments in @args are the same as they would be specified in "new()".

$fh = $file->open($mode, $permissions)

Passes the given arguments, including $file, to "IO::File->new" (which in turn calls "IO::File->open" and returns the result as an IO::File object. If the opening fails, "undef" is returned and $! is set.

$fh = $file->openr()

A shortcut for

```
$fh = $file->open('r') or croak "Can't read $file: $!";
```

$fh = $file->openw()

A shortcut for

```
$fh = $file->open('w') or croak "Can't write to $file: $!";
```

$fh = $file->opena()

A shortcut for

```
$fh = $file->open('a') or croak "Can't append to $file: $!";
```

$file->touch

Sets the modification and access time of the given file to right now, if the file exists. If it doesn't exist, "touch()" will make it exist, and - YES! - set its modification and access time to now.

$file->slurp()

In a scalar context, returns the contents of $file in a string. In a list context,

returns the lines of $file (according to how $/ is set) as a list. If the file can't be read, this method will throw an exception.

If you want "chomp()" run on each line of the file, pass a true value for the "chomp" or "chomped" parameters:

```
my @lines = $file->slurp(chomp => 1);
```

You may also use the "iomode" parameter to pass in an IO mode to use when opening the file, usually IO layers (though anything accepted by the MODE argument of "open()" is accepted here). Just make sure it's a reading mode.

```
my @lines = $file->slurp(iomode => ':crlf');
my $lines = $file->slurp(iomode => '<:encoding(UTF-8)');
```

The default "iomode" is "r".

Lines can also be automatically split, mimicking the perl command-line option "-a" by using the "split" parameter. If this parameter is used, each line will be returned as an array ref.

```
my @lines = $file->slurp( chomp => 1, split => qr/\s*,\s*/ );
```

The "split" parameter can only be used in a list context.

$file->spew($content);

The opposite of "slurp", this takes a list of strings and prints them to the file in write mode. If the file can't be written to, this method will throw an exception.

The content to be written can be either an array ref or a plain scalar. If the content is an array ref then each entry in the array will be written to the file.

You may use the "iomode" parameter to pass in an IO mode to use when opening the file, just like "slurp" supports.

```
$file->spew(iomode => '>:raw', $content);
```

The default "iomode" is "w".

$file->spew_lines($content);

Just like "spew", but, if $content is a plain scalar, appends $/ to it, or, if $content is an array ref, appends $/ to each element of the array.

Can also take an "iomode" parameter like "spew". Again, the default "iomode" is "w".

$file->traverse(sub { ... }, @args)

Calls the given callback on $file. This doesn't do much on its own, but see the associated documentation in Path::Class::Dir.

$file->remove()

This method will remove the file in a way that works well on all platforms, and returns a boolean value indicating whether or not the file was successfully removed. "remove()" is better than simply calling Perl's "unlink()" function, because on some platforms (notably VMS) you actually may need to call "unlink()" several times before all versions of the file are gone - the "remove()" method handles this process for you.

$st = $file->stat()

Invokes "File::stat::stat()" on this file and returns a File::stat object representing the result.

$st = $file->lstat()

Same as "stat()", but if $file is a symbolic link, "lstat()" stats the link instead of the file the link points to.

$class = $file->dir_class()

Returns the class which should be used to create directory objects.

Generally overridden whenever this class is subclassed.

$copy = $file->copy_to($dest);

Copies the $file to $dest. It returns a Path::Class::File object when successful, "undef" otherwise.

$moved = $file->move_to($dest);

Moves the $file to $dest, and updates $file accordingly.

It returns $file is successful, "undef" otherwise.

AUTHOR

Ken Williams, kwilliams@cpan.org

SEE ALSO

Path::Class, Path::Class::Dir, File::Spec