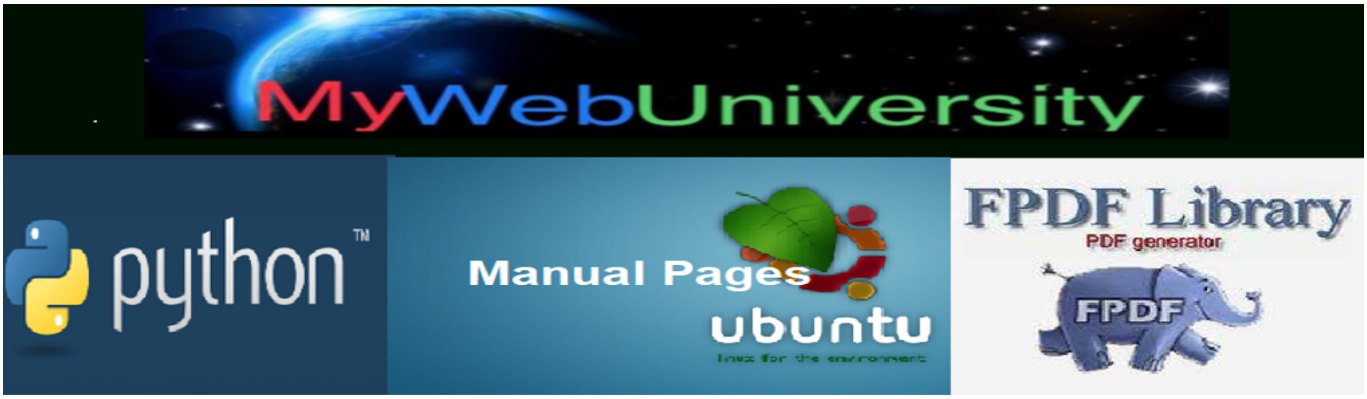

Rocky Enterprise Linux 9.2 Manual Pages on command 'Path::Class::Dir.3pm'

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

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

NAME

Path::Class::Dir - Objects representing directories

VERSION

version 0.37

SYNOPSIS

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

```
print $dir->as_foreign('Mac');   # :foo:bar:
print $dir->as_foreign('Win32'); #  foo\bar
# Iterate with IO::Dir methods:
my $handle = $dir->open;
while (my $file = $handle->read) {
  $file = $dir->file($file);  # Turn into Path::Class::File object
  ...
}
# Iterate with Path::Class methods:
while (my $file = $dir->next) {
  # $file is a Path::Class::File or Path::Class::Dir object
  ...
}
```

DESCRIPTION

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

METHODS

$dir = Path::Class::Dir->new(<dir1>, <dir2>, ...)

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

Creates a new "Path::Class::Dir" object and returns it. The arguments specify names of directories which will be joined to create a single directory object. A volume may also be specified as the first argument, or as part of the first argument. You can use platform-neutral syntax:

```
my $dir = dir( 'foo', 'bar', 'baz' );
```

or platform-native syntax:

```
my $dir = dir( 'foo/bar/baz' );
```

or a mixture of the two:

```
my $dir = dir( 'foo/bar', 'baz' );
```

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

```
my $dir = dir( '/var/tmp' );
```

or use an empty string as the first argument:

```
my $dir = dir( '', 'var', 'tmp' );
```

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 (many non-Unix platforms don't have a notion of a "root directory"), so they probably shouldn't appear in your code if you're trying to be cross-platform. The first form is perfectly natural, because paths like this may come from config files, user input, or whatever.

As a special case, since it doesn't otherwise mean anything useful and it's convenient to define this way, "Path::Class::Dir->new()" (or "dir()") refers to the current directory ("File::Spec->curdir"). To get the current directory as an absolute path, do "dir()->absolute".

Finally, as another special case "dir(undef)" will return undef, since that's usually an accident on the part of the caller, and returning the root directory would be a nasty surprise just asking for trouble a few lines later.

$dir->stringify

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

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

$dir->volume

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

$dir->basename

Returns the last directory name of the path as a string.

$dir->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.

$dir->is_absolute

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

$dir->is_relative

Returns true or false depending on whether the directory refers to a relative path specifier (like "lib/foo" or "./dir").

$dir->cleanup

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

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

$dir->resolve

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

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

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

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

Returns a Path::Class::File object representing an entry in $dir or one of its subdirectories. Internally, this just calls "Path::Class::File->new(@_)".

$subdir = $dir->subdir(<dir1>, <dir2>, ...)

Returns a new "Path::Class::Dir" object representing a subdirectory of $dir.

$parent = $dir->parent

Returns the parent directory of $dir. Note that this is the logical parent, not necessarily the physical parent. It really means we just chop off entries from the end of the directory list until we cain't chop no more. If the directory is relative, we start using the relative forms of parent directories.

The following code demonstrates the behavior on absolute and relative directories:

```
$dir = dir('/foo/bar');
for (1..6) {
  print "Absolute: $dir\n";
  $dir = $dir->parent;
}
$dir = dir('foo/bar');
for (1..6) {
  print "Relative: $dir\n";
  $dir = $dir->parent;
}
############ Output on Unix #################
Absolute: /foo/bar
Absolute: /foo
```

Absolute: /

Absolute: /

Absolute: /

Absolute: /

Relative: foo/bar

Relative: foo

Relative: .

Relative: ..

Relative: ../..

Relative: ../../..

@list = $dir->children

Returns a list of Path::Class::File and/or "Path::Class::Dir" objects listed in this directory, or in scalar context the number of such objects. Obviously, it is necessary for $dir to exist and be readable in order to find its children.

Note that the children are returned as subdirectories of $dir, i.e. the children of foo will be foo/bar and foo/baz, not bar and baz.

Ordinarily "children()" will not include the self and parent entries "." and ".." (or their equivalents on non-Unix systems), because that's like I'm-my-own-grandpa business. If you do want all directory entries including these special ones, pass a true value for the "all" parameter:

```
@c = $dir->children(); # Just the children
@c = $dir->children(all => 1); # All entries
```

In addition, there's a "no_hidden" parameter that will exclude all normally "hidden" entries - on Unix this means excluding all entries that begin with a dot ("."):

```
@c = $dir->children(no_hidden => 1); # Just normally-visible entries
```

$abs = $dir->absolute

Returns a "Path::Class::Dir" object representing $dir 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 = $dir->relative

Returns a "Path::Class::Dir" object representing $dir 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.

$boolean = $dir->subsumes($other)

Returns true if this directory spec subsumes the other spec, and false otherwise.

Think of "subsumes" as "contains", but we only look at the specs, not whether $dir actually contains $other on the filesystem.

The $other argument may be a "Path::Class::Dir" object, a Path::Class::File object, or a string. In the latter case, we assume it's a directory.

```
# Examples:
dir('foo/bar' )->subsumes(dir('foo/bar/baz'))  # True
dir('/foo/bar')->subsumes(dir('/foo/bar/baz')) # True
dir('foo/..')->subsumes(dir('foo/../bar'))      # True
dir('foo/bar' )->subsumes(dir('bar/baz'))      # False
dir('/foo/bar')->subsumes(dir('foo/bar'))      # False
dir('foo/..')->subsumes(dir('bar'))            # False! Use C<contains> to resolve ".."
```

$boolean = $dir->contains($other)

Returns true if this directory actually contains $other on the filesystem. $other doesn't have to be a direct child of $dir, it just has to be subsumed after both paths have been resolved.

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

Returns a "Path::Class::Dir" object representing $dir 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.

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

Returns a "Path::Class::Dir" object representing $dir 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()".

@list = $dir->dir_list([OFFSET, [LENGTH]])

Returns the list of strings internally representing this directory structure. Each successive member of the list is understood to be an entry in its predecessor's

directory list. By contract, "Path::Class->new($dir->dir_list)" should be equivalent to $dir.

The semantics of this method are similar to Perl's "splice" or "substr" functions; they return "LENGTH" elements starting at "OFFSET". If "LENGTH" is omitted, returns all the elements starting at "OFFSET" up to the end of the list. If "LENGTH" is negative, returns the elements from "OFFSET" onward except for "-LENGTH" elements at the end. If "OFFSET" is negative, it counts backward "OFFSET" elements from the end of the list. If "OFFSET" and "LENGTH" are both omitted, the entire list is returned.

In a scalar context, "dir_list()" with no arguments returns the number of entries in the directory list; "dir_list(OFFSET)" returns the single element at that offset; "dir_list(OFFSET, LENGTH)" returns the final element that would have been returned in a list context.

$dir->components

Identical to "dir_list()". It exists because there's an analogous method "dir_list()" in the "Path::Class::File" class that also returns the basename string, so this method lets someone call "components()" without caring whether the object is a file or a directory.

$fh = $dir->open()

Passes $dir to "IO::Dir->open" and returns the result as an IO::Dir object. If the opening fails, "undef" is returned and $! is set.

$dir->mkpath($verbose, $mode)

Passes all arguments, including $dir, to "File::Path::mkpath()" and returns the result (a list of all directories created).

$dir->rmtree($verbose, $cautious)

Passes all arguments, including $dir, to "File::Path::rmtree()" and returns the result (the number of files successfully deleted).

$dir->remove()

Removes the directory, which must be empty. Returns a boolean value indicating whether or not the directory was successfully removed. This method is mainly provided for consistency with "Path::Class::File"'s "remove()" method.

$dir->tempfile(...)

An interface to File::Temp's "tempfile()" function. Just like that function, if you call this in a scalar context, the return value is the filehandle and the file is

"unlink"ed as soon as possible (which is immediately on Unix-like platforms). If called in a list context, the return values are the filehandle and the filename.

The given directory is passed as the "DIR" parameter.

Here's an example of pretty good usage which doesn't allow race conditions, won't leave yucky tempfiles around on your filesystem, etc.:

```
my $fh = $dir->tempfile;
print $fh "Here's some data...\n";
seek($fh, 0, 0);
while (<$fh>) { do something... }
```

Or in combination with a "fork":

```
my $fh = $dir->tempfile;
print $fh "Here's some more data...\n";
seek($fh, 0, 0);
if ($pid=fork()) {
  wait;
} else {
  something($_) while <$fh>;
}
```

$dir_or_file = $dir->next()

A convenient way to iterate through directory contents. The first time "next()" is called, it will "open()" the directory and read the first item from it, returning the result as a "Path::Class::Dir" or Path::Class::File object (depending, of course, on its actual type). Each subsequent call to "next()" will simply iterate over the directory's contents, until there are no more items in the directory, and then the undefined value is returned. For example, to iterate over all the regular files in a directory:

```
while (my $file = $dir->next) {
  next unless -f $file;
  my $fh = $file->open('r') or die "Can't read $file: $!";
  ...
}
```

If an error occurs when opening the directory (for instance, it doesn't exist or isn't readable), "next()" will throw an exception with the value of $!.

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

Calls the given callback for the root, passing it a continuation function which, when called, will call this recursively on each of its children. The callback function should be of the form:

```
sub {
  my ($child, $cont, @args) = @_;
  # ...
}
```

For instance, to calculate the number of files in a directory, you can do this:

```
my $nfiles = $dir->traverse(sub {
  my ($child, $cont) = @_;
  return sum($cont->(), ($child->is_dir ? 0 : 1));
});
```

or to calculate the maximum depth of a directory:

```
my $depth = $dir->traverse(sub {
  my ($child, $cont, $depth) = @_;
  return max($cont->($depth + 1), $depth);
}, 0);
```

You can also choose not to call the callback in certain situations:

```
$dir->traverse(sub {
  my ($child, $cont) = @_;
  return if -l $child; # don't follow symlinks
  # do something with $child
  return $cont->();
});
```

$dir->traverse_if(sub { ... }, sub { ... }, @args)

traverse with additional "should I visit this child" callback. Particularly useful in case examined tree contains inaccessible directories.

Canonical example:

```
$dir->traverse_if(
  sub {
    my ($child, $cont) = @_;
    # do something with $child
```

```
        return $cont->();
      },
      sub {
        my ($child) = @_;
        # Process only readable items
        return -r $child;
      });
```

Second callback gets single parameter: child. Only children for which it returns true will be processed by the first callback.

Remaining parameters are interpreted as in traverse, in particular "traverse_if(callback, sub { 1 }, @args" is equivalent to "traverse(callback, @args)".

$dir->recurse(callback => sub {...})

Iterates through this directory and all of its children, and all of its children's children, etc., calling the "callback" subroutine for each entry. This is a lot like what the File::Find module does, and of course "File::Find" will work fine on Path::Class objects, but the advantage of the "recurse()" method is that it will also feed your callback routine "Path::Class" objects rather than just pathname strings.

The "recurse()" method requires a "callback" parameter specifying the subroutine to invoke for each entry. It will be passed the "Path::Class" object as its first argument.

"recurse()" also accepts two boolean parameters, "depthfirst" and "preorder" that control the order of recursion. The default is a preorder, breadth-first search, i.e. "depthfirst => 0, preorder => 1". At the time of this writing, all combinations of these two parameters are supported except "depthfirst => 0, preorder => 0".

"callback" is normally not required to return any value. If it returns special constant "Path::Class::Entity::PRUNE()" (more easily available as "$item->PRUNE"), no children of analyzed item will be analyzed (mostly as if you set "$File::Find::prune=1"). Of course pruning is available only in "preorder", in postorder return value has no effect.

$st = $file->stat()

Invokes "File::stat::stat()" on this directory 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 directory the link points to.

$class = $file->file_class()

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

Generally overridden whenever this class is subclassed.

AUTHOR

Ken Williams, kwilliams@cpan.org

SEE ALSO

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