NAME

PIR − Short alias for Path::Iterator::Rule

VERSION

version 1.014

SYNOPSIS

```
use PIR;

my $rule = PIR->new;          # match anything
$rule->file->size(">10k");    # add/chain rules

# iterator interface
my $next = $rule->iter( @dirs );
while ( defined( my $file = $next->() ) ) {
  ...
}

# list interface
for my $file ( $rule->all( @dirs ) ) {
  ...
}
```

DESCRIPTION

This is an empty subclass of Path::Iterator::Rule. It saves you from having to type the full name repeatedly, which is particularly handy for one-liners:

```
$ perl -MPIR -wE 'say for PIR->new->skip_dirs(".")->perl_module->all(@INC)'
```

USAGE

Constructors

new

```
my $rule = Path::Iterator::Rule->new;
```

Creates a new rule object that matches any file or directory. It takes no arguments. For convenience, it may also be called on an object, in which case it still returns a new object that matches any file or directory.

clone

```
my $common      = Path::Iterator::Rule->new->file->not_empty;
my $big_files   = $common->clone->size(">1M");
my $small_files = $common->clone->size("<10K");
```

Creates a copy of a rule object. Useful for customizing different rule objects against a common base.

Matching and iteration

iter

```
my $next = $rule->iter( @dirs, \%options);
while ( defined( my $file = $next->() ) ) {
  ...
}
```

Creates a subroutine reference iterator that returns a single result when dereferenced. This iterator is ''lazy'' — results are not pre-computed.

It takes as arguments a list of directories to search and an optional hash reference of control options. If no search directories are provided, the current directory is used (`"."`). Valid options include:

- `depthfirst` — Controls order of results. Valid values are ''1'' (post-order, depth-first search), ''0'' (breadth-first search) or ''−1'' (pre-order, depth-first search). Default is 0.

- `error_handler` — Catches errors during execution of rule tests. Default handler dies with the filename and error. If set to undef, error handling is disabled.
- `follow_symlinks` — Follow directory symlinks when true. Default is 1.
- `report_symlinks` — Includes symlinks in results when true. Default is equal to `follow_symlinks`.
- `loop_safe` — Prevents visiting the same directory more than once when true. Default is 1.
- `relative` — Return matching items relative to the search directory. Default is 0.
- `sorted` — Whether entries in a directory are sorted before processing. Default is 1.
- `visitor` — An optional coderef that will be called on items matching all rules.

Filesystem loops might exist from either hard or soft links. The `loop_safe` option prevents infinite loops, but adds some overhead by making `stat` calls. Because directories are visited only once when `loop_safe` is true, matches could come from a symlinked directory before the real directory depending on the search order.

To get only the real files, turn off `follow_symlinks`. You can have symlinks included in results, but not descend into symlink directories if you turn off `follow_symlinks`, but turn on `report_symlinks`.

Turning `loop_safe` off and leaving `follow_symlinks` on avoids `stat` calls and will be fastest, but with the risk of an infinite loop and repeated files. The default is slow, but safe.

The `error_handler` parameter must be a subroutine reference. It will be called when a rule test throws an exception. The first argument will be the file name being inspected and the second argument will be the exception.

The optional `visitor` parameter must be a subroutine reference. If set, it will be called for any result that matches. It is called the same way a custom rule would be (see "EXTENDING") but its return value is ignored. It is called when an item is first inspected — "postorder" is not respected.

The paths inspected and returned will be relative to the search directories provided. If these are absolute, then the paths returned will have absolute paths. If these are relative, then the paths returned will have relative paths.

If the search directories are absolute and the `relative` option is true, files returned will be relative to the search directory. Note that if the search directories are not mutually exclusive (whether containing subdirectories like `@INC` or symbolic links), files found could be returned relative to different initial search directories based on `depthfirst`, `follow_symlinks` or `loop_safe`.

When the iterator is exhausted, it will return undef.

iter_fast

This works just like `iter`, except that it optimizes for speed over safety. Don't do this unless you're sure you need it and accept the consequences. See "PERFORMANCE" for details.

all

```
my @matches = $rule->all( @dir, \%options );
```

Returns a list of paths that match the rule. It takes the same arguments and has the same behaviors as the `iter` method. The `all` method uses `iter` internally to fetch all results.

In scalar context, it will return the count of matched paths.

In void context, it is optimized to iterate over everything, but not store results. This is most useful with the `visitor` option:

```
$rule->all( $path, { visitor => \&callback } );
```

all_fast

This works just like `all`, except that it optimizes for speed over safety. Don't do this unless you're sure

you need it and accept the consequences. See ''PERFORMANCE'' for details.

test

```
if ( $rule->test( $path, $basename, $stash ) ) { ... }
```

Test a file path against a rule. Used internally, but provided should someone want to create their own, custom iteration algorithm.

Logic operations

`Path::Iterator::Rule` provides three logic operations for adding rules to the object. Rules may be either a subroutine reference with specific semantics (described below in ''EXTENDING'') or another `Path::Iterator::Rule` object.

and

```
$rule->and( sub { -r -w -x $_ } ); # stacked filetest example
$rule->and( @more_rules );
```

Adds one or more constraints to the current rule. E.g. ''old rule AND new1 AND new2 AND ...''. Returns the object to allow method chaining.

or

```
$rule->or(
  $rule->new->name("foo*"),
  $rule->new->name("bar*"),
  sub { -r -w -x $_ },
);
```

Takes one or more alternatives and adds them as a constraint to the current rule. E.g. ''old rule AND (new1 OR new2 OR ...)''. Returns the object to allow method chaining.

not

```
$rule->not( sub { -r -w -x $_ } );
```

Takes one or more alternatives and adds them as a negative constraint to the current rule. E.g. ''old rule AND NOT (new1 AND new2 AND ...)''. Returns the object to allow method chaining.

skip

```
$rule->skip(
  $rule->new->dir->not_writeable,
  $rule->new->dir->name("foo"),
);
```

Takes one or more alternatives and will prune a directory if any of the criteria match or if any of the rules already indicate the directory should be pruned. Pruning means the directory will not be returned by the iterator and will not be searched.

For files, it is equivalent to `$rule->not($rule->or(@rules))`. Returns the object to allow method chaining.

This method should be called as early as possible in the rule chain. See ''skip_dirs'' below for further explanation and an example.

RULE METHODS

Rule methods are helpers that add constraints. Internally, they generate a closure to accomplish the desired logic and add it to the rule object with the `and` method. Rule methods return the object to allow for method chaining.

File name rules

name

```
$rule->name( "foo.txt" );
$rule->name( qr/foo/, "bar.*");
```

The `name` method takes one or more patterns and creates a rule that is true if any of the patterns match the **basename** of the file or directory path. Patterns may be regular expressions or glob expressions (or literal names).

iname

```
$rule->iname( "foo.txt" );
$rule->iname( qr/foo/, "bar.*");
```

The `iname` method is just like the `name` method, but matches case-insensitively.

skip_dirs

```
$rule->skip_dirs( @patterns );
```

The `skip_dirs` method skips directories that match one or more patterns. Patterns may be regular expressions or globs (just like `name`). Directories that match will not be returned from the iterator and will be excluded from further search. **This includes the starting directories.** If that isn't what you want, see "skip_subdirs" instead.

Note: this rule should be specified early so that it has a chance to operate before a logical shortcut. E.g.

```
$rule->skip_dirs(".git")->file; # OK
$rule->file->skip_dirs(".git"); # Won't work
```

In the latter case, when a ".git" directory is seen, the `file` rule shortcuts the rule before the `skip_dirs` rule has a chance to act.

skip_subdirs

```
$rule->skip_subdirs( @patterns );
```

This works just like `skip_dirs`, except that the starting directories (depth 0) are not skipped and may be returned from the iterator unless excluded by other rules.

File test rules

Most of the `-X` style filetest are available as boolean rules. The table below maps the filetest to its corresponding method name.

Test	Method	Test	Method
-r	readable	-R	r_readable
-w	writeable	-W	r_writeable
-w	writable	-W	r_writable
-x	executable	-X	r_executable
-o	owned	-O	r_owned
-e	exists	-f	file
-z	empty	-d	directory, dir
-s	nonempty	-l	symlink
		-p	fifo
-u	setuid	-S	socket
-g	setgid	-b	block
-k	sticky	-c	character
		-t	tty
-T	ascii		
-B	binary		

For example:

```
$rule->file->nonempty; # -f -s $file
```

The -X operators for timestamps take a single argument in a form that Number::Compare can interpret.

Test	Method
-A	accessed
-M	modified
-C	changed

For example:

```
$rule->modified(">1"); # -M $file > 1
```

Stat test rules

All of the `stat` elements have a method that takes a single argument in a form understood by Number::Compare.

stat()	Method
0	dev
1	ino
2	mode
3	nlink
4	uid
5	gid
6	rdev
7	size
8	atime
9	mtime
10	ctime
11	blksize
12	blocks

For example:

```
$rule->size(">10K")
```

Depth rules

```
$rule->min_depth(3);
$rule->max_depth(5);
```

The `min_depth` and `max_depth` rule methods take a single argument and limit the paths returned to a minimum or maximum depth (respectively) from the starting search directory. A depth of 0 means the starting directory itself. A depth of 1 means its children. (This is similar to the Unix `find` utility.)

Perl file rules

```
# All perl rules
$rule->perl_file;

# Individual perl file rules
$rule->perl_module;     # .pm files
$rule->perl_pod;        # .pod files
$rule->perl_test;       # .t files
$rule->perl_installer;  # Makefile.PL or Build.PL
$rule->perl_script;     # .pl or 'perl' in the shebang
```

These rule methods match file names (or a shebang line) that are typical of Perl distribution files.

Version control file rules

```
# Skip all known VCS files
$rule->skip_vcs;

# Skip individual VCS files
```

```
$rule->skip_cvs;
$rule->skip_rcs;
$rule->skip_svn;
$rule->skip_git;
$rule->skip_bzr;
$rule->skip_hg;
$rule->skip_darcs;
```

Skips files and/or prunes directories related to a version control system. Just like `skip_dirs`, these rules should be specified early to get the correct behavior.

File content rules

contents_match

```
$rule->contents_match(qr/BEGIN .* END/xs);
```

The `contents_match` rule takes a list of regular expressions and returns files that match one of the expressions.

The expressions are applied to the file's contents as a single string. For large files, this is likely to take significant time and memory.

Files are assumed to be encoded in UTF-8, but alternative Perl IO layers can be passed as the first argument:

```
$rule->contents_match(":encoding(iso-8859-1)", qr/BEGIN .* END/xs);
```

See perlio for further details.

line_match

```
$rule->line_match(qr/^new/i, qr/^Addition/);
```

The `line_match` rule takes a list of regular expressions and returns files with at least one line that matches one of the expressions.

Files are assumed to be encoded in UTF-8, but alternative Perl IO layers can be passed as the first argument.

shebang

```
$rule->shebang(qr/#!.*\bperl\b/);
```

The `shebang` rule takes a list of regular expressions or glob patterns and checks them against the first line of a file.

Other rules

dangling

```
$rule->symlink->dangling;
$rule->not_dangling;
```

The `dangling` rule method matches dangling symlinks. Use it or its inverse to control how dangling symlinks should be treated.

Negated rules

Most rule methods have a negated form preceded by "not_".

```
$rule->not_name("foo.*")
```

Because this happens automatically, it includes somewhat silly ones like `not_nonempty` (which is thus a less efficient way of saying `empty`).

Rules that skip directories or version control files do not have a negated version.

EXTENDING

Custom rule subroutines

Rules are implemented as (usually anonymous) subroutine callbacks that return a value indicating whether or not the rule matches. These callbacks are called with three arguments. The first argument is a path, which is also locally aliased as the `$_` global variable for convenience in simple tests.

```
$rule->and( sub { -r -w -x $_ } ); # tests $_
```

The second argument is the basename of the path, which is useful for certain types of name checks:

```
$rule->and( sub { $_[1] =~ /foo|bar/ } ); "foo" or "bar" in basename;
```

The third argument is a hash reference that can be used to maintain state. Keys beginning with an underscore are **reserved** for `Path::Iterator::Rule` to provide additional data about the search in progress. For example, the `_depth` key is used to support minimum and maximum depth checks.

The custom rule subroutine must return one of four values:

- A true value — indicates the constraint is satisfied
- A false value — indicates the constraint is not satisfied
- `\1` — indicate the constraint is satisfied, and prune if it's a directory
- `\0` — indicate the constraint is not satisfied, and prune if it's a directory

A reference is a special flag that signals that a directory should not be searched recursively, regardless of whether the directory should be returned by the iterator or not.

The legacy "0 but true" value used previously for pruning is no longer valid and will throw an exception if it is detected.

Here is an example. This is equivalent to the "max_depth" rule method with a depth of 3:

```
$rule->and(
  sub {
    my ($path, $basename, $stash) = @_;
    return 1 if $stash->{_depth} < 3;
    return \1 if $stash->{_depth} == 3;
    return \0; # should never get here
  }
);
```

Files and directories and directories up to depth 3 will be returned and directories will be searched. Files of depth 3 will be returned. Directories of depth 3 will be returned, but their contents will not be added to the search.

Returning a reference is "sticky" — they will propagate through "and" and "or" logic.

```
0 && \0 = \0    \0 && 0 = \0    0 || \0 = \0    \0 || 0 = \0
0 && \1 = \0    \0 && 1 = \0    0 || \1 = \1    \0 || 1 = \1
1 && \0 = \0    \1 && 0 = \0    1 || \0 = \1    \1 || 0 = \1
1 && \1 = \1    \1 && 1 = \1    1 || \1 = \1    \1 || 1 = \1
```

Once a directory is flagged to be pruned, it will be pruned regardless of subsequent rules.

```
$rule->max_depth(3)->name(qr/foo/);
```

This will return files or directories with "foo" in the name, but all directories at depth 3 will be pruned, regardless of whether they match the name rule.

Generally, if you want to do directory pruning, you are encouraged to use the "skip" method instead of writing your own logic using `\0` and `\1`.

Extension modules and custom rule methods

One of the strengths of File::Find::Rule is the many CPAN modules that extend it. `Path::Iterator::Rule` provides the `add_helper` method to provide a similar mechanism for extensions.

The `add_helper` class method takes three arguments, a `name` for the rule method, a closure-generating callback, and a flag for not generating a negated form of the rule. Unless the flag is true, an inverted "not_*" method is generated automatically. Extension classes should call this as a class method to install new rule methods. For example, this adds a "foo" method that checks if the filename is "foo":

```
package Path::Iterator::Rule::Foo;

use Path::Iterator::Rule;

Path::Iterator::Rule->add_helper(
  foo => sub {
    my @args = @_; # do this to customize closure with arguments
    return sub {
      my ($item, $basename) = @_;
      return if -d "$item";
      return $basename =~ /^foo$/;
    }
  }
);

1;
```

This allows the following rule methods:

```
$rule->foo;
$fule->not_foo;
```

The `add_helper` method will warn and ignore a helper with the same name as an existing method.

Subclassing

Instead of processing and returning strings, this module may be subclassed to operate on objects that represent files. Such objects **must** stringify to a file path.

The following private implementation methods must be overridden:

- _objectify — given a path, return an object
- _children — given a directory, return an (unsorted) list of [basename, full path] entries within it, excluding "." and ".."

Note that `_children` should return a *list* of *tuples*, where the tuples are array references containing basename and full path.

See Path::Class::Rule source for an example.

LEXICAL WARNINGS

If you run with lexical warnings enabled, `Path::Iterator::Rule` will issue warnings in certain circumstances (such as an unreadable directory that must be skipped). To disable these categories, put the following statement at the correct scope:

```
no warnings 'Path::Iterator::Rule';
```

PERFORMANCE

By default, `Path::Iterator::Rule` iterator options are "slow but safe". They ensure uniqueness, return files in sorted order, and throw nice error messages if something goes wrong.

If you want speed over safety, set these options:

```
%options = (
    loop_safe => 0,
    sorted => 0,
    depthfirst => -1,
    error_handler => undef
);
```

Alternatively, use the `iter_fast` and `all_fast` methods instead, which set these options for you.

```
$iter = $rule->iter( @dirs, \%options );
```

```
$iter = $rule->iter_fast( @dirs ); # same thing
```

Depending on the file structure being searched, `depthfirst => -1` may or may not be a good choice. If you have lots of nested directories and all the files at the bottom, a depth first search might do less work or use less memory, particularly if the search will be halted early (e.g. finding the first N matches.)

Rules will shortcut on failure, so be sure to put rules likely to fail early in a rule chain.

Consider:

```
$r1 = Path::Iterator::Rule->new->name(qr/foo/)->file;
$r2 = Path::Iterator::Rule->new->file->name(qr/foo/);
```

If there are lots of files, but only a few containing ''foo'', then `$r1` above will be faster.

Rules are implemented as code references, so long chains have some overhead. Consider testing with a custom coderef that combines several tests into one.

Consider:

```
$r3 = Path::Iterator::Rule->new->and( sub { -x -w -r $_ } );
$r4 = Path::Iterator::Rule->new->executable->writeable->readable;
```

Rule `$r3` above will be much faster, not only because it stacks the file tests, but because it requires only a single code reference.

CAVEATS

Some features are still unimplemented:

- Untainting options
- Some File::Find::Rule helpers (e.g. `grep`)
- Extension class loading via `import()`

Filetest operators and stat rules are subject to the usual portability considerations. See perlport for details.

SEE ALSO

There are many other file finding modules out there. They all have various features/deficiencies, depending on your preferences and needs. Here is an (incomplete) list of alternatives, with some comparison commentary.

Path::Class::Rule and IO::All::Rule are subclasses of `Path::Iterator::Rule` and operate on Path::Class and IO::All objects, respectively. Because of this, they are substantially slower on large directory trees than just using this module directly.

File::Find is part of the Perl core. It requires the user to write a callback function to process each node of the search. Callbacks must use global variables to determine the current node. It only supports depth-first search (both pre- and post-order). It supports pre- and post-processing callbacks; the former is required for sorting files to process in a directory. File::Find::Closures can be used to help create a callback for File::Find.

File::Find::Rule is an object-oriented wrapper around File::Find. It provides a number of helper functions and there are many more `File::Find::Rule::*` modules on CPAN with additional helpers. It provides an iterator interface, but precomputes all the results.

File::Next provides iterators for file, directories or ''everything''. It takes two callbacks, one to match files and one to decide which directories to descend. It does not allow control over breadth/depth order, though it does provide means to sort files for processing within a directory. Like File::Find, it requires callbacks to use global variables.

Path::Class::Iterator walks a directory structure with an iterator. It is implemented as Path::Class subclasses, which adds a degree of extra complexity. It takes a single callback to define ''interesting'' paths to return. The callback gets a Path::Class::Iterator::File or Path::Class::Iterator::Dir object for evaluation.

File::Find::Object and companion File::Find::Object::Rule are like File::Find and File::Find::Rule, but without File::Find inside. They use an iterator that does not precompute results. They can return

File::Find::Object::Result objects, which give a subset of the utility of Path::Class objects. File::Find::Object::Rule appears to be a literal translation of File::Find::Rule, including oddities like making `-M` into a boolean.

File::chdir::WalkDir recursively descends a tree, calling a callback on each file. No iterator. Supports exclusion patterns. Depth-first post-order by default, but offers pre-order option. Does not process symlinks.

File::Find::Iterator is based on iterator patterns in Higher Order Perl. It allows a filtering callback. Symlinks are followed automatically without infinite loop protection. No control over order. It offers a "state file" option for resuming interrupted work.

File::Find::Declare has declarative helper rules, no iterator, is Moose-based and offers no control over ordering or following symlinks.

File::Find::Node has no iterator, does matching via callback and offers no control over ordering.

File::Set builds up a set of files to operate on from a list of directories to include or exclude, with control over recursion. A callback is applied to each file (or directory) in the set. There is no iterator. There is no control over ordering. Symlinks are not followed. It has several extra features for checksumming the set and creating tarballs with */bin/tar*.

THANKS

Thank you to Ricardo Signes (rjbs) for inspiring me to write yet another file finder module, for writing file finder optimization benchmarks, and tirelessly running my code over and over to see if it got faster.

- See the speed of Perl file finders <http://rjbs.manxome.org/rubric/entry/1981>

AUTHOR

David Golden <dagolden@cpan.org>

COPYRIGHT AND LICENSE