NAME

Monitoring::Plugin − A family of perl modules to streamline writing Naemon, Nagios, Icinga or Shinken (and compatible) plugins.

SYNOPSIS

```
# Constants OK, WARNING, CRITICAL, and UNKNOWN are exported by default
# See also Monitoring::Plugin::Functions for a functional interface
use Monitoring::Plugin;

# Constructor
$np = Monitoring::Plugin->new;                                  # OR
$np = Monitoring::Plugin->new( shortname => "PAGESIZE" );    # OR


# use Monitoring::Plugin::Getopt to process the @ARGV command line options:
#   --verbose, --help, --usage, --timeout and --host are defined automatically
$np = Monitoring::Plugin->new(
  usage => "Usage: %s [ -v|--verbose ]  [-H <host>] [-t <timeout>] "
    . "[ -c|--critical=<threshold> ] [ -w|--warning=<threshold> ]",
);

# add valid command line options and build them into your usage/help documenta
$np->add_arg(
  spec => 'warning|w=s',
  help => '-w, --warning=INTEGER:INTEGER .  See '
    . 'https://www.monitoring-plugins.org/doc/guidelines.html#THRESHOLDFORMAT
    . 'for the threshold format. ',
);

# Parse @ARGV and process standard arguments (e.g. usage, help, version)
$np->getopts;


# Exit/return value methods - plugin_exit( CODE, MESSAGE ),
#                             plugin_die( MESSAGE, [CODE])
$page = retrieve_page($page1)
    or $np->plugin_exit( UNKNOWN, "Could not retrieve page" );
    # Return code: 3;
    #   output: PAGESIZE UNKNOWN - Could not retrieve page
test_page($page)
    or $np->plugin_exit( CRITICAL, "Bad page found" );

# plugin_die() is just like plugin_exit(), but return code defaults
#   to UNKNOWN
$page = retrieve_page($page2)
  or $np->plugin_die( "Could not retrieve page" );
  # Return code: 3;
  #   output: PAGESIZE UNKNOWN - Could not retrieve page

# Threshold methods
$code = $np->check_threshold(
  check => $value,
  warning => $warning_threshold,
  critical => $critical_threshold,
);
$np->plugin_exit( $code, "Threshold check failed" ) if $code != OK;


# Message methods (EXPERIMENTAL AND SUBJECT TO CHANGE) -
```

```
#   add_message( CODE, $message ); check_messages()
for (@collection) {
  if (m/Error/) {
    $np->add_message( CRITICAL, $_ );
  } else {
    $np->add_message( OK, $_ );
  }
}
($code, $message) = $np->check_messages();
plugin_exit( $code, $message );
# If any items in collection matched m/Error/, returns CRITICAL and
#   the joined set of Error messages; otherwise returns OK and the
#   joined set of ok messages


# Perfdata methods
$np->add_perfdata(
  label => "size",
  value => $value,
  uom => "kB",
  threshold => $threshold,
);
$np->add_perfdata( label => "time", ... );
$np->plugin_exit( OK, "page size at http://... was ${value}kB" );
# Return code: 0;
#   output: PAGESIZE OK - page size at http://... was 36kB \
#   | size=36kB;10:25;25: time=...
```

DESCRIPTION

Monitoring::Plugin and its associated Monitoring::Plugin::* modules are a family of perl modules to streamline writing Monitoring plugins. The main end user modules are Monitoring::Plugin, providing an object-oriented interface to the entire Monitoring::Plugin::* collection, and Monitoring::Plugin::Functions, providing a simpler functional interface to a useful subset of the available functionality.

The purpose of the collection is to make it as simple as possible for developers to create plugins that conform the Monitoring Plugin guidelines (https://www.monitoring-plugins.org/doc/guidelines.html).

EXPORTS

Nagios status code constants are exported by default:

```
OK
WARNING
CRITICAL
UNKNOWN
DEPENDENT
```

The following variables are also exported on request:

`%ERRORS`

A hash mapping error strings (“CRITICAL”, “UNKNOWN”, etc.) to the corresponding status code.

`%STATUS_TEXT`

A hash mapping status code constants (OK, WARNING, CRITICAL, etc.) to the corresponding error string (“OK”, “WARNING, ”CRITICAL", etc.) i.e. the reverse of `%ERRORS`.

CONSTRUCTOR

```
Monitoring::Plugin->new;

Monitoring::Plugin->new( shortname => 'PAGESIZE' );

Monitoring::Plugin->new(
        usage => "Usage: %s [ -v|--verbose ]  [-H <host>] [-t <timeout>]
```

```
                    [ -c|--critical=<critical threshold> ] [ -w|--warning=<warni
                version => $VERSION,
                blurb   => $blurb,
                extra   => $extra,
                url     => $url,
                license => $license,
                plugin  => basename $0,
                timeout => 15,
        );
```

Instantiates a new Monitoring::Plugin object. Accepts the following named arguments:

shortname
: The 'shortname' for this plugin, used as the first token in the plugin output by the various exit methods. Default: uc basename `$0`.

usage ("Usage: `%s` --foo --bar")
: Passing a value for the *usage()* argument makes Monitoring::Plugin instantiate its own `Monitoring::Plugin::Getopt` object so you can start doing command line argument processing. See "CONSTRUCTOR" in Monitoring::Plugin::Getopt for more about "usage" and the following options:

version
url
blurb
license
extra
plugin
timeout

OPTION HANDLING METHODS

`Monitoring::Plugin` provides these methods for accessing the functionality in `Monitoring::Plugin::Getopt`.

add_arg
: Examples:

```
  # Define --hello argument (named parameters)
  $plugin->add_arg(
    spec => 'hello=s',
    help => "--hello\n   Hello string",
    required => 1,
  );

  # Define --hello argument (positional parameters)
  #   Parameter order is 'spec', 'help', 'default', 'required?'
  $plugin->add_arg('hello=s', "--hello\n   Hello string", undef, 1);
```

  See "ARGUMENTS" in Monitoring::Plugin::Getopt for more details.

getopts()
: Parses and processes the command line options you've defined, automatically doing the right thing with help/usage/version arguments.

  See "GETOPTS" in Monitoring::Plugin::Getopt for more details.

opts()
: Assuming you've instantiated it by passing 'usage' to *new()*, *opts()* returns the Monitoring::Plugin object's `Monitoring::Plugin::Getopt` object, with which you can do lots of great things.

  E.g.

```
  if ( $plugin->opts->verbose ) {
          print "yah yah YAH YAH YAH!!!";
  }
```

```
# start counting down to timeout
alarm $plugin->opts->timeout;
your_long_check_step_that_might_time_out();

# access any of your custom command line options,
# assuming you've done these steps above:
#   $plugin->add_arg('my_argument=s', '--my_argument [STRING]');
#   $plugin->getopts;
print $plugin->opts->my_argument;
```

Again, see Monitoring::Plugin::Getopt.

EXIT METHODS

plugin_exit(<CODE>, `$message`)

Exit with return code CODE, and a standard nagios message of the form ‘‘SHORTNAME CODE − `$message`’’.

plugin_die(`$message`, [<CODE>])

Same as *plugin_exit()*, except that CODE is optional, defaulting to UNKNOWN. NOTE: exceptions are not raised by default to calling code. Set `$_use_die` flag if this functionality is required (see test code).

nagios_exit(<CODE>, `$message`)

Alias for *plugin_die()*. Deprecated.

nagios_die(`$message`, [<CODE>])

Alias for *plugin_die()*. Deprecated.

die(`$message`, [<CODE>])

Alias for *plugin_die()*. Deprecated.

max_state, max_state_alt

These are wrapper function for Monitoring::Plugin::Functions::max_state and Monitoring::Plugin::Functions::max_state_alt.

THRESHOLD METHODS

These provide a top level interface to the `Monitoring::Plugin::Threshold` module; for more details, see Monitoring::Plugin::Threshold and Monitoring::Plugin::Range.

check_threshold(`$value`)

check_threshold(check => `$value`, warning => `$warn`, critical => `$crit`)

Evaluates `$value` against the thresholds and returns OK, CRITICAL, or WARNING constant. The thresholds may be:

1. explicitly set by passing ’warning’ and/or ’critical’ parameters to `check_threshold()`, or,
2. explicitly set by calling `set_thresholds()` before `check_threshold()`, or,
3. implicitly set by command-line parameters −w, −c, −−critical or −−warning, if you have run `$plugin->getopts()`.

You can specify `$value` as an array of values and each will be checked against the thresholds.

The return value is ready to pass to C <plugin_exit>, e . g .,

```
$p->plugin_exit(
      return_code => $p->check_threshold($result),
      message     => " sample result was $result"
);
```

set_thresholds(warning => ‘‘10:25’’, critical => ‘‘~:25’’)

Sets the acceptable ranges and creates the plugin’s Monitoring::Plugins::Threshold object. See https://www.monitoring−plugins.org/doc/guidelines.html#THRESHOLDFORMAT for details and examples of the threshold format.

threshold()

Returns the object's `Monitoring::Plugin::Threshold` object, if it has been defined by calling *set_thresholds()*. You can pass a new Threshold object to it to replace the old one too, but you shouldn't need to do that from a plugin script.

MESSAGE METHODS

EXPERIMENTAL AND SUBJECT TO CHANGE

add_messages and check_messages are higher-level convenience methods to add and then check a set of messages, returning an appropriate return code and/or result message. They are equivalent to maintaining a set of `@critical`, `@warning`, and and `@ok` message arrays (add_message), and then doing a final if test (check_messages) like this:

```
if (@critical) {
  plugin_exit( CRITICAL, join(' ', @critical) );
}
elsif (@warning) {
  plugin_exit( WARNING, join(' ', @warning) );
}
else {
  plugin_exit( OK, join(' ', @ok) );
}
```

add_message(<CODE>, `$message`)

Add a message with CODE status to the object. May be called multiple times. The messages added are checked by check_messages, following.

Only CRITICAL, WARNING, and OK are accepted as valid codes.

check_messages()

Check the current set of messages and return an appropriate nagios return code and/or a result message. In scalar context, returns only a return code; in list context returns both a return code and an output message, suitable for passing directly to *plugin_exit()* e.g.

```
$code = $np->check_messages;
($code, $message) = $np->check_messages;
```

check_messages returns CRITICAL if any critical messages are found, WARNING if any warning messages are found, and OK otherwise. The message returned in list context defaults to the joined set of error messages; this may be customised using the arguments below.

check_messages accepts the following named arguments (none are required):

join => SCALAR

A string used to join the relevant array to generate the message string returned in list context i.e. if the 'critical' array `@crit` is non-empty, check_messages would return:

```
join( $join, @crit )
```

as the result message. Default: ' ' (space).

join_all => SCALAR

By default, only one set of messages are joined and returned in the result message i.e. if the result is CRITICAL, only the 'critical' messages are included in the result; if WARNING, only the 'warning' messages are included; if OK, the 'ok' messages are included (if supplied) i.e. the default is to return an 'errors-only' type message.

If join_all is supplied, however, it will be used as a string to join the resultant critical, warning, and ok messages together i.e. all messages are joined and returned.

critical => ARRAYREF

Additional critical messages to supplement any passed in via *add_message()*.

warning => ARRAYREF

Additional warning messages to supplement any passed in via *add_message()*.

ok => ARRAYREF | SCALAR

Additional ok messages to supplement any passed in via *add_message()*.

PERFORMANCE DATA METHODS

add_perfdata(label => ‘‘size’’, value => `$value`, uom => ‘‘kB’’, threshold => `$threshold`)

Add a set of performance data to the object. May be called multiple times. The performance data is included in the standard plugin output messages by the various exit methods.

See the Monitoring::Plugin::Performance documentation for more information on performance data and the various field definitions, as well as the relevant section of the Monitoring Plugin guidelines (https://www.monitoring−plugins.org/doc/guidelines.html#AEN202).

EXAMPLES

‘‘Enough talk! Show me some examples!’’

See the file ’check_stuff.pl’ in the ’t’ directory included with the Monitoring::Plugin distribution for a complete working example of a plugin script.

VERSIONING

The Monitoring::Plugin::* modules are currently experimental and so the interfaces may change up until Monitoring::Plugin hits version 1.0, although every attempt will be made to keep them as backwards compatible as possible.

SEE ALSO

See Monitoring::Plugin::Functions for a simple functional interface to a subset of the available Monitoring::Plugin functionality.

See also Monitoring::Plugin::Getopt, Monitoring::Plugin::Range, Monitoring::Plugin::Performance, Monitoring::Plugin::Range, and Monitoring::Plugin::Threshold.

The Monitoring Plugin project page is at http://monitoring−plugins.org.

BUGS

Please report bugs in these modules to the Monitoring Plugin development team: devel@monitoring−plugins.org.

AUTHOR

Maintained by the Monitoring Plugin development team − https://www.monitoring−plugins.org.

Originally by Ton Voon, <ton DOT voon AT altinity DOT com>.

COPYRIGHT AND LICENSE