lcd047 Manual: clarification about quiet_messages.   b10c59b 16 days ago

17 contributors

1021 lines (870 sloc)   48.9 KB   Raw   Blame   History

```
*syntastic.txt*   Syntax checking on the fly has never been so pimp.
*syntastic*


                    It's a bird! It's a plane! ZOMG It's ... ~

                   _____             __             __  _      ~
                  / ___/__  ______  / /_____ ______/ /_(_)____ ~
                  \__ \/ / / / __ \/ __/ __ `/ ___/ __/ / ___/ ~
                 ___/ / /_/ / / / / /_/ /_/ (__  ) /_/ / /__   ~
                /____/\__, /_/ /_/\__/\__,_/____/\__/_/\___/   ~
                     /____/                                    ~



                              Reference Manual~


==============================================================================
CONTENTS                                                  *syntastic-contents*

    1.Intro........................................|syntastic-intro|
        1.1.Quick start............................|syntastic-quickstart|
        1.2.Recommended settings...................|syntastic-recommended|
    2.Functionality provided.......................|syntastic-functionality|
        2.1.The statusline flag....................|syntastic-statusline-flag|
        2.2.Error signs............................|syntastic-error-signs|
        2.3.Error window...........................|syntastic-error-window|
        2.4.Error highlighting.....................|syntastic-highlighting|
        2.5.Aggregating errors.....................|syntastic-aggregating-errors|
        2.6.Filtering errors.......................|syntastic-filtering-errors|
    3.Commands.....................................|syntastic-commands|
    4.Global Options...............................|syntastic-global-options|
    5.Checker Options..............................|syntastic-checker-options|
        5.1.Choosing which checkers to use.........|syntastic-filetype-checkers|
        5.2.Choosing the executable................|syntastic-config-exec|
        5.3.Configuring specific checkers..........|syntastic-config-makeprg|
        5.4.Sorting errors.........................|syntastic-config-sort|
    6.Notes........................................|syntastic-notes|
        6.1.Handling of composite filetypes........|syntastic-composite|
        6.2.Editing files over network.............|syntastic-netrw|
        6.3.The 'shellslash' option................|syntastic-shellslash|
        6.4.Saving Vim sessions....................|syntastic-sessions|
    7.Compatibility with other software............|syntastic-compatibility|
        7.1.The csh and tcsh shells................|syntastic-csh|
        7.2.Eclim..................................|syntastic-eclim|
        7.3.The fish shell.........................|syntastic-fish|
        7.4.The fizsh shell........................|syntastic-fizsh|
        7.5.flagship...............................|syntastic-flagship|
        7.6.powerline..............................|syntastic-powerline|
        7.7.The PowerShell shell...................|syntastic-powershell|
        7.8.python-mode............................|syntastic-pymode|
        7.9.vim-auto-save..........................|syntastic-vim-auto-save|
        7.10.vim-go................................|syntastic-vim-go|
        7.11.vim-virtualenv........................|syntastic-vim-virtualenv|
        7.12.YouCompleteMe.........................|syntastic-ycm|
        7.13.The zsh shell and MacVim..............|syntastic-zsh|
    8.About........................................|syntastic-about|
    9.License......................................|syntastic-license|


==============================================================================
1. Intro                                                     *syntastic-intro*

Syntastic is a syntax checking plugin that runs files through external syntax
checkers. This can be done on demand, or automatically as files are saved and
opened.  If syntax errors are detected, the user is notified and is happy
because they didn't have to compile their code or execute their script to find
them.

Syntastic comes in two parts: the syntax checker plugins, and the core. The
syntax checker plugins are defined on a per-filetype basis where each one wraps
up an external syntax checking program. The core script delegates off to these
plugins and uses their output to provide the syntastic functionality.

Take a look at the wiki for a list of supported filetypes and checkers:

    https://github.com/scrooloose/syntastic/wiki/Syntax-Checkers

Note: This doc only deals with using syntastic. To learn how to write syntax
checker integrations, see the guide on the GitHub wiki:

    https://github.com/scrooloose/syntastic/wiki/Syntax-Checker-Guide

------------------------------------------------------------------------------
1.1. Quick start                                        *syntastic-quickstart*

Syntastic comes preconfigured with a default list of enabled checkers per
|filetype|. This list is kept reasonably short to prevent slowing down Vim or
trying to use conflicting checkers.

You can see the list of checkers available for the current filetype with the
|:SyntasticInfo| command.

You probably want to override the configured list of checkers for the
filetypes you use, and also change the arguments passed to specific checkers
to suit your needs. See |syntastic-checker-options| below for details.

Use |:SyntasticCheck| to manually check right now. Use |:Errors| to open the
|location-list| window, and |:lclose| to close it. You can clear the error
list with |:SyntasticReset|, and you can use |:SyntasticToggleMode| to switch
between active (checking on writing the buffer) and passive (manual) checking.

You don't have to switch focus to the |location-list| window to jump to the
different errors.  Vim provides several built-in commands for this, for
example |:lnext| and |:lprevious|.  You may want to add shortcut mappings for
these commands, or perhaps install a plugin such as Tim Pope's 'unimpaired'
(see https://github.com/tpope/vim-unimpaired) that provides such mappings.

------------------------------------------------------------------------------
1.2. Recommended settings                              *syntastic-recommended*

Syntastic has numerous options that can be configured, and the defaults are
not particularly well suitable for new users. It is recommended that you start
by adding the following lines to your vimrc, and return to them later as
needed: >
    set statusline+=%#warningmsg#
    set statusline+=%{SyntasticStatuslineFlag()}
    set statusline+=%*

    let g:syntastic_always_populate_loc_list = 1
    let g:syntastic_auto_loc_list = 1
    let g:syntastic_check_on_open = 1
    let g:syntastic_check_on_wq = 0
<
==============================================================================
2. Functionality provided                            *syntastic-functionality*
```

```

Syntax checking can be done automatically or on demand (see
|'syntastic_mode_map'| and |:SyntasticToggleMode| for configuring this).

When syntax checking is done, the features below can be used to notify the
user of errors. See |syntastic-global-options| for how to configure and
activate/deactivate these features.

    * A statusline flag
    * Signs beside lines with errors
    * The |location-list| can be populated with the errors for the associated
      buffer
    * Erroneous parts of lines can be highlighted (this functionality is only
      provided by some syntax checkers)
    * Balloons (if the |+balloon_eval| feature is compiled in) can be used to
      display error messages for erroneous lines when hovering the mouse over
      them
    * Error messages from multiple checkers can be aggregated in a single list

------------------------------------------------------------------------------
2.1. The statusline flag                           *syntastic-statusline-flag*

To use the statusline flag, this must appear in your |'statusline'| setting >
    %{SyntasticStatuslineFlag()}
<
Something like this could be more useful: >
    set statusline+=%#warningmsg#
    set statusline+=%{SyntasticStatuslineFlag()}
    set statusline+=%*
<
When syntax errors are detected a flag will be shown. The content of the flag
is derived from the |syntastic_stl_format| option.

------------------------------------------------------------------------------
2.2. Error signs                                       *syntastic-error-signs*

Syntastic uses the |:sign| commands (provided that the |+signs| feature is
compiled in) to mark lines with errors and warnings in the sign column. To
enable this feature, use the |'syntastic_enable_signs'| option.

Signs are colored using the Error and Todo syntax highlight groups by default
(see |group-name|). If you wish to customize the colors for the signs, you
can use the following groups:
    SyntasticErrorSign - For syntax errors, links to 'error' by default
    SyntasticWarningSign - For syntax warnings, links to 'todo' by default
    SyntasticStyleErrorSign - For style errors, links to 'SyntasticErrorSign'
                              by default
    SyntasticStyleWarningSign - For style warnings, links to
                                'SyntasticWarningSign' by default

Example: >
    highlight SyntasticErrorSign guifg=white guibg=red
<
To set up highlighting for the line where a sign resides, you can use the
following highlight groups:
    SyntasticErrorLine
    SyntasticWarningLine
    SyntasticStyleErrorLine - Links to 'SyntasticErrorLine' by default
    SyntasticStyleWarningLine - Links to 'SyntasticWarningLine' by default

Example: >
    highlight SyntasticErrorLine guibg=#2f0000
<
------------------------------------------------------------------------------
2.3. The error window                                   *syntastic-error-window*

You can use the |:Errors| command to display the errors for the current buffer
in the |location-list|.

By default syntastic doesn't fill the |location-list| with the errors found by
the checkers, in order to reduce clashes with other plugins. Consequently, if
you run |:lopen| or |:lwindow| rather than |:Errors| to open the error window
you wouldn't see syntastic's list of errors. If you insist on using |:lopen|

or |:lwindow| you should either run |:SyntasticSetLoclist| after running the
checks, or set |syntastic_always_populate_loc_list| which tells syntastic to
update the |location-list| automatically.

------------------------------------------------------------------------------
2.4. Error highlighting                               *syntastic-highlighting*

Some checkers provide enough information for syntastic to be able to highlight
errors. By default the SpellBad syntax highlight group is used to color errors,
and the SpellCap group is used for warnings. If you wish to customize the
colors for highlighting you can use the following groups:
    SyntasticError - Links to 'SpellBad' by default
    SyntasticWarning - Links to 'SpellCap' by default
    SyntasticStyleError - Links to SyntasticError by default
    SyntasticStyleWarning - Links to SyntasticWarning by default

Example: >
    highlight SyntasticError guibg=#2f0000
<
------------------------------------------------------------------------------
2.5. Aggregating errors                         *syntastic-aggregating-errors*

By default, namely if |'syntastic_aggregate_errors'| is unset, syntastic runs
in turn the checkers corresponding to the filetype of the current file (see
|syntastic-filetype-checkers|), and stops as soon as a checker reports any
errors. It then notifies you of the errors using the notification mechanisms
above. In this mode error lists are always produced by a single checker, and,
if you open the error window, the name of the checker that generated the errors
is shown on the statusline of the error window.

If |'syntastic_aggregate_errors'| is set, syntastic runs all checkers that
apply (still cf. |syntastic-filetype-checkers|), then aggregates errors found
by all checkers in a single list, and notifies you. In this mode each error
message is labeled with the name of the checker that generated it, but you can
disable generation of these labels by turning off '|syntastic_id_checkers|'.

If |'syntastic_sort_aggregated_errors'| is set (which is the default), messages
in the aggregated list are grouped by file, then sorted by line number, then
type, then column number.  Otherwise messages produced by the same checker are
grouped together, and sorting within each group is decided by the variables
|'syntastic_<filetype>_<checker>_sort'|.

------------------------------------------------------------------------------
2.6 Filtering errors                              *syntastic-filtering-errors*

You can selectively disable some of the errors found by checkers either
using |'syntastic_quiet_messages'|, or by specifying a list of patterns in
|'syntastic_ignore_files'|.

See also: |'syntastic_<filetype>_<checker>_quiet_messages'| and
|'b:syntastic_skip_checks'|.

==============================================================================
3. Commands                                               *syntastic-commands*

:Errors                                                              *:Errors*

When errors have been detected, use this command to pop up the |location-list|
and display the error messages.

Please note that the |:Errors| command overwrites the current location list with
syntastic's own location list.

:SyntasticToggleMode                                    *:SyntasticToggleMode*

Toggles syntastic between active and passive mode. See |'syntastic_mode_map'|
for more info.

:SyntasticCheck                                              *:SyntasticCheck*

Manually cause a syntax check to be done.  By default the checkers in the
```

```
|'g:syntastic_<filetype>_checkers'| or |'b:syntastic_checkers'| lists are run,
cf. |syntastic-filetype-checkers|.  If |'syntastic_aggregate_errors'| is unset
(which is the default), checking stops the first time a checker reports any
errors; if |'syntastic_aggregate_errors'| is set, all checkers that apply are
run in turn, and all errors found are aggregated in a single list.

The command may be followed by a (space separated) list of checkers.  In this
case |'g:syntastic_<filetype>_checkers'| and |'b:syntastic_checkers'| are
ignored, and the checkers named by the command's arguments are run instead, in
the order specified.  The set by |'syntastic_aggregate_errors'| still apply.

Example: >
    :SyntasticCheck flake8 pylint
<
:SyntasticInfo                                                 *:SyntasticInfo*

The command takes an optional argument, and outputs information about the
checkers available for the filetype named by said argument, or for the current
filetype if no argument was provided.

:SyntasticReset                                               *:SyntasticReset*

Resets the list of errors and turns off all error notifiers.

:SyntasticSetLoclist                                     *:SyntasticSetLoclist*

If |'syntastic_always_populate_loc_list'| is not set, the |location-list| is
not filled in automatically with the list of errors detected by the checkers.
This is useful if you run syntastic along with other plugins that use location
lists. The |:SyntasticSetLoclist| command allows you to stick the errors into
the location list explicitly.

==============================================================================
4. Global Options                                   *syntastic-global-options*

                                                   *'syntastic_check_on_open'*
Default: 0
If this variable is enabled, syntastic in active mode will run syntax checks
when buffers are first loaded, as well as on saving: >
    let g:syntastic_check_on_open = 1
<
                                                     *'syntastic_check_on_wq'*
Default: 1
In active mode syntax checks are normally run whenever buffers are written to
disk, even when the writes happen just before quitting Vim.  If you want to
skip checks when you issue |:wq|, |:x|, and |:ZZ|, set this variable to 0: >
    let g:syntastic_check_on_wq = 0
<
                                                *'syntastic_aggregate_errors'*
Default: 0
When enabled, syntastic runs all checkers that apply to the current filetype,
then aggregates errors found by all checkers and displays them. When disabled,
syntastic runs each checker in turn, and stops to display the results the first
time a checker finds any errors. >
    let g:syntastic_aggregate_errors = 1
<
                                                     *'syntastic_id_checkers'*
Default: 1
When results from multiple checkers are aggregated in a single error list
(that is either when |'syntastic_aggregate_errors'| is enabled, or when
checking a file with a composite filetype), it might not be immediately
obvious which checker has produced a given error message. This variable
instructs syntastic to label error messages with the names of the checkers
that created them. >
    let g:syntastic_id_checkers = 0
<
                                          *'syntastic_sort_aggregated_errors'*
Default: 1
By default, when results from multiple checkers are aggregated in a single
error list (that is either when |'syntastic_aggregate_errors'| is enabled,
or when checking a file with a composite filetype), errors are grouped by
file, then sorted by line number, then grouped by type (namely errors take
precedence over warnings), then they are sorted by column number.  If you want
to leave messages grouped by checker output, set this variable to 0: >
    let g:syntastic_sort_aggregated_errors = 0
<
                                              *'syntastic_echo_current_error'*
Default: 1
If enabled, syntastic will echo current error to the command window. If
multiple errors are found on the same line, |'syntastic_cursor_columns'| is
used to decide which one is shown. >
    let g:syntastic_echo_current_error = 1
<
                                                  *'syntastic_cursor_columns'*
Default: 1
This option controls which errors are echoed to the command window if
|'syntastic_echo_current_error'| is set and multiple errors are found on the
same line. When the option is enabled, the first error corresponding to the
current column is shown. Otherwise, the first error on the current line is
echoed, regardless of the cursor position on the current line.

When dealing with very large lists of errors, disabling this option can speed
up navigation significantly: >
    let g:syntastic_cursor_column = 0
<
                                                    *'syntastic_enable_signs'*
Default: 1
Use this option to tell syntastic whether to use the |:sign| interface to mark
syntax errors: >
    let g:syntastic_enable_signs = 1
<
                     *'syntastic_error_symbol'* *'syntastic_style_error_symbol'*
                 *'syntastic_warning_symbol'* *'syntastic_style_warning_symbol'*
Use this option to control what the syntastic |:sign| text contains. Several
error symbols can be customized:
    syntastic_error_symbol - For syntax errors, defaults to '>>'
    syntastic_style_error_symbol - For style errors, defaults to 'S>'
    syntastic_warning_symbol - For syntax warnings, defaults to '>>'
    syntastic_style_warning_symbol - For style warnings, defaults to 'S>'

Example: >
    let g:syntastic_error_symbol = "✗"
    let g:syntastic_warning_symbol = "⚠"
<
                                                 *'syntastic_enable_balloons'*
Default: 1
Use this option to tell syntastic whether to display error messages in balloons
when the mouse is hovered over erroneous lines: >
    let g:syntastic_enable_balloons = 1
<
Note that Vim must be compiled with |+balloon_eval|.

                                             *'syntastic_enable_highlighting'*
Default: 1
Use this option to tell syntastic whether to use syntax highlighting to mark
errors (where possible). Highlighting can be turned off with the following >
    let g:syntastic_enable_highlighting = 0
<
                                        *'syntastic_always_populate_loc_list'*
Default: 0
By default syntastic doesn't fill the |location-list| with the errors found
by the checkers, in order to reduce clashes with other plugins. Enable this
option to tell syntastic to always stick any detected errors into the
|location-list|: >
    let g:syntastic_always_populate_loc_list = 1
<
Please note that if |'syntastic_auto_jump'| is set to a non-zero value the
location list is overwritten with Syntastic's own list when taking a jump,
regardless of the value of |'syntastic_always_populate_loc_list'|.  The
location list is also overwritten when running the |:Errors| command.

                                                       *'syntastic_auto_jump'*
Default: 0
Enable this option if you want the cursor to jump to the first detected issue
when saving or opening a file.
```

```

When set to 0 the cursor won't jump automatically. >
    let g:syntastic_auto_jump = 0
<
When set to 1 the cursor will always jump to the first issue detected,
regardless of type. >
    let g:syntastic_auto_jump = 1
<
When set to 2 the cursor will jump to the first issue detected, but only if
this issue is an error. >
    let g:syntastic_auto_jump = 2
<
When set to 3 the cursor will jump to the first error detected, if any. If
all issues detected are warnings, the cursor won't jump. >
    let g:syntastic_auto_jump = 3
<
Please note that in either situation taking the jump also has the side effect
of the location list being overwritten with Syntastic's own location list,
regardless of the value of |'syntastic_always_populate_loc_list'|.

                                                   *'syntastic_auto_loc_list'*
Default: 2
Use this option to tell syntastic to automatically open and/or close the
|location-list| (see |syntastic-error-window|).

When set to 0 the error window will be neither opened nor closed
automatically. >
    let g:syntastic_auto_loc_list = 0
<
When set to 1 the error window will be automatically opened when errors are
detected, and closed when none are detected. >
    let g:syntastic_auto_loc_list = 1
<
When set to 2 the error window will be automatically closed when no errors are
detected, but not opened automatically. >
    let g:syntastic_auto_loc_list = 2
<
When set to 3 the error window will be automatically opened when errors are
detected, but not closed automatically. >
    let g:syntastic_auto_loc_list = 3
<
                                                 *'syntastic_loc_list_height'*
Default: 10
Use this option to specify the height of the location lists that syntastic
opens. >
    let g:syntastic_loc_list_height = 5
<
                                                    *'syntastic_ignore_files'*
Default: []
Use this option to specify files that syntastic should never check.  It's a
list of |regular-expression| patterns.  The full paths of files (see |::p|) are
matched against these patterns, and the matches are case-sensitive. Use |\c|
to specify case-insensitive patterns.  Example: >
    let g:syntastic_ignore_files = ['\m^/usr/include/', '\m\c\.h$']
<
                                                    *'syntastic_filetype_map'*
Default: {}
Use this option to map non-standard filetypes to standard ones.  Corresponding
checkers are mapped accordingly, which allows syntastic to check files with
non-standard filetypes: >
    let g:syntastic_filetype_map = {
        \ "plaintex": "tex",
        \ "gentoo-metadata": "xml" }
<
Composite filetypes can also be mapped to simple types, which disables the
default behaviour of running both checkers against the input file: >
    let g:syntastic_filetype_map = { "handlebars.html": "handlebars" }
<
                                                        *'syntastic_mode_map'*
Default: { "mode": "active",
           "active_filetypes": [],
           "passive_filetypes": [] }
Use this option to fine tune when automatic syntax checking is done (or not
done).

The option should be set to something like: >

    let g:syntastic_mode_map = {
        \ "mode": "active",
        \ "active_filetypes": ["ruby", "php"],
        \ "passive_filetypes": ["puppet"] }
<
"mode" can be mapped to one of two values - "active" or "passive". When set
to "active", syntastic does automatic checking whenever a buffer is saved or
initially opened.  When set to "passive" syntastic only checks when the user
calls |:SyntasticCheck|.

The exceptions to these rules are defined with "active_filetypes" and
"passive_filetypes". In passive mode, automatic checks are still done for
filetypes in the "active_filetypes" array (and "passive_filetypes" is
ignored). In active mode, automatic checks are not done for any filetypes in
the "passive_filetypes" array ("active_filetypes" is ignored).

If any of "mode", "active_filetypes", or "passive_filetypes" are left
unspecified, they default to values above.

If local variable |'b:syntastic_mode'| is defined its value takes precedence
over all calculations involving |'syntastic_mode_map'| for the corresponding
buffer.

At runtime, the |:SyntasticToggleMode| command can be used to switch between
active and passive modes.

                                                          *'b:syntastic_mode'*
Default: unset
Only the local form |'b:syntastic_mode'| is used. When set to either "active"
or "passive", it takes precedence over |'syntastic_mode_map'| when deciding
whether the corresponding buffer should be checked automatically.

                                                  *'syntastic_quiet_messages'*
Default: {}
Use this option to filter out some of the messages produced by checkers.  The
option should be set to something like: >
    let g:syntastic_quiet_messages = {
        \ "!level":  "errors",
        \ "type":    "style",
        \ "regex":   '\m\[C03\d\d\]',
        \ "file:p":  ['\m^/usr/include/', '\m\c\.h$'] }
<
Each element turns off messages matching the patterns specified by the
corresponding value. Values are lists, but if a list consist of a single
element you may omit the brackets (e.g. you may write "style" instead of
["style"]). Elements with values [] or '' are ignored (this is useful for
overriding filters, cf. |filter-overrides|).

    "level" - takes one of two values, "warnings" or "errors"
    "type"  - can be either "syntax" or "style"
    "regex" - each item in list is matched against the messages' text as a
              case-insensitive |regular-expression|
    "file"  - each item in list is matched against the filenames the messages
              refer to, as a case-sensitive |regular-expression|.

If a key is prefixed by an exclamation mark "!", the corresponding filter is
negated (i.e. the above example silences all messages that are NOT errors).

The "file" key may be followed by one or more filename modifiers (see
|filename-modifiers|). The modifiers are applied to the filenames the messages
refer to before matching against the value (i.e. in the above example the full
path of the issues are matched against '\m^/usr/include/' and '\m\c\.h$').

If |'syntastic_id_checkers'| is set, filters are applied before error messages
are labeled with the names of the checkers that created them.

There are also checker-specific variants of this option, providing finer
control. They are named |'syntastic_<filetype>_<checker>_quiet_messages'|.

```

```
For a particular checker, if both a |'syntastic_quiet_messages'| filter and
a checker-specific filter are present, they are both applied (to the list of
errors produced by the said checker). In case of conflicting values for the
same keys, the values of the checker-specific filters take precedence.

                                                            *filter-overrides*
Since filter elements with values [] or '' are ignored, you can disable global
filters for particular checkers, by setting the values of the corresponding
elements in |'syntastic_<filetype>_<checker>_quiet_messages'| to [] or ''. For
example, the following setting will silence all warnings, except for the
ones produced by "pylint": >
    let g:syntastic_quiet_messages = { "level": "warnings" }
    let g:syntastic_python_pylint_quiet_messages = { "level" : [] }
<
                                                      *'syntastic_stl_format'*
Default: [Syntax: line:%F (%t)]
Use this option to control what the syntastic statusline text contains. Several
magic flags are available to insert information:
    %e  - number of errors
    %w  - number of warnings
    %t  - total number of warnings and errors
    %ne - filename of file containing first error
    %nw - filename of file containing first warning
    %N  - filename of file containing first warning or error
    %pe - filename with path of file containing first error
    %pw - filename with path of file containing first warning
    %P  - filename with path of file containing first warning or error
    %fe - line number of first error
    %fw - line number of first warning
    %F  - line number of first warning or error

These flags accept width and alignment controls similar to the ones used by
|'statusline'| flags:
    %-0{minwid}.{maxwid}{flag}

All fields except {flag} are optional. A single percent sign can be given as
"%%".

Several additional flags are available to hide text under certain conditions:
    %E{...} - hide the text in the brackets unless there are errors
    %W{...} - hide the text in the brackets unless there are warnings
    %B{...} - hide the text in the brackets unless there are both warnings AND
              errors
These flags can't be nested.

Example: >
    let g:syntastic_stl_format = '[%E{Err: %fe #%e}%B{, }%W{Warn: %fw #%w}]'
<
If this format is used and the current buffer has 5 errors and 1 warning
starting on lines 20 and 10 respectively then this would appear on the
statusline: >
    [Err: 20 #5, Warn: 10 #1]
<
If the buffer had 2 warnings, starting on line 5 then this would appear: >
    [Warn: 5 #2]
<
                                                   *'b:syntastic_skip_checks'*
Default: unset
Only the local form |'b:syntastic_skip_checks'| is used. When set to a true
value, no checks are run against the corresponding buffer. Example: >
    let b:syntastic_skip_checks = 1
<
                                                    *'syntastic_full_redraws'*
Default: 0 in GUI Vim and MacVim, 1 otherwise
Controls whether syntastic calls |:redraw| or |:redraw!| for screen redraws.
Changing it can in principle make screen redraws smoother, but it can also
cause screen to flicker, or cause ghost characters. Leaving it to the default
should be safe.

                                                     *'syntastic_exit_checks'*
Default: 0 when running under "cmd.exe" on Windows, 1 otherwise

Syntastic attempts to catch abnormal termination conditions from checkers by
```

```
looking at their exit codes. The "cmd.exe" shell on Windows make these checks
meaningless, by returning 1 to Vim when the checkers exit with non-zero codes.
The above variable can be used to disable exit code checks in syntastic.

                                                           *'syntastic_shell'*
Default: Vim's 'shell'
This is the (full path to) the shell syntastic will use to run the checkers.
On UNIX and Mac OS-X this shell must accept Bourne-compatible syntax for
file "stdout" and "stderr" redirections ">file" and "2>file".  Examples of
compatible shells are "zsh", "bash", "ksh", and of course the original Bourne
"sh".

This shell is independent of Vim's 'shell', and it isn't used for interactive
operations.  It must take care to initialize all environment variables needed
by the checkers you're using.  Example: >
    let g:syntastic_shell = "/bin/sh"
<
                                              *'syntastic_nested_autocommands'*
Default: 0
Controls whether syntastic's autocommands |BufReadPost| and |BufWritePost|
are called from other |BufReadPost| and |BufWritePost| autocommands (see
|autocmd-nested|).  This is known to trigger interoperability problems with
other plugins, so only enable it if you actually need that functionality.

                                                           *'syntastic_debug'*
Default: 0
Set this to the sum of one or more of the following flags to enable
debugging:

     1 - trace general workflow
     2 - dump location lists
     4 - trace notifiers
     8 - trace autocommands
    16 - dump options
    32 - trace running of specific checkers

Example: >
    let g:syntastic_debug = 1
<
Syntastic will then add debugging messages to Vim's |message-history|. You can
examine these messages with |:mes|.

                                                      *'syntastic_debug_file'*
Default: unset
When set, debugging messages are written to the file named by its value, in
addition to being added to Vim's |message-history|: >
    let g:syntastic_debug_file = '~/syntastic.log'
<
                                                 *'syntastic_extra_filetypes'*
Default: []
List of filetypes handled by checkers external to syntastic.  If you have a Vim
plugin that adds a checker for syntastic, and if the said checker deals with a
filetype that is unknown to syntastic, you might consider adding that filetype
to this list: >
    let g:syntastic_extra_filetypes = [ "make", "gitcommit" ]
<
This will allow |:SyntasticInfo| to do proper tab completion for the new
filetypes.

==============================================================================
5. Checker Options                                   *syntastic-checker-options*

------------------------------------------------------------------------------
5.1 Choosing which checkers to use                 *syntastic-filetype-checkers*

                                           *'g:syntastic_<filetype>_checkers'*
You can tell syntastic which checkers to run for a given filetype by setting a
variable 'g:syntastic_<filetype>_checkers' to a list of checkers, e.g. >
    let g:syntastic_php_checkers = ["php", "phpcs", "phpmd"]
<

                                                      *'b:syntastic_checkers'*
There is also a per-buffer version of this setting, 'b:syntastic_checkers'.
```

```
When set, it takes precedence over |'g:syntastic_<filetype>_checkers'|.  You can
use this in an autocmd to configure specific checkers for particular paths: >
    autocmd FileType python if stridx(expand("%:p"), "/some/path/") == 0 |
        \ let b:syntastic_checkers = ["pylint"] | endif
<
If neither |'g:syntastic_<filetype>_checkers'| nor |'b:syntastic_checkers'|
is set, a default list of checker is used. Beware however that this list
deliberately kept minimal, for performance reasons.

Take a look at the wiki to find out what checkers and filetypes are supported
by syntastic:

    https://github.com/scrooloose/syntastic/wiki/Syntax-Checkers

Use |:SyntasticInfo| to see which checkers are available for a given filetype.

------------------------------------------------------------------------------
5.2 Choosing the executable                            *syntastic-config-exec*

                                       *'syntastic_<filetype>_<checker>_exec'*
The executable run by a checker is normally defined automatically, when the
checker is registered. You can however override it, by setting the variable
'g:syntastic_<filetype>_<checker>_exec': >
    let g:syntastic_ruby_mri_exec = '~/bin/ruby2'
<
This variable has a local version, 'b:syntastic_<filetype>_<checker>_exec',
which takes precedence over the global one in the corresponding buffer.

                                                *'b:syntastic_<checker>_exec'*
And there is also a local variable named 'b:syntastic_<checker>_exec', which
takes precedence over both 'b:syntastic_<filetype>_<checker>_exec' and
'g:syntastic_<filetype>_<checker>_exec' in the buffers where it is defined.

------------------------------------------------------------------------------
5.3 Configuring specific checkers                   *syntastic-config-makeprg*

Most checkers use the 'makeprgBuild()' function and provide many options by
default - in fact you can customise every part of the command that gets called.

                                   *'syntastic_<filetype>_<checker>_<option>'*
Checkers that use 'makeprgBuild()' construct a 'makeprg' like this: >
    let makeprg = self.makeprgBuild({
                \ "exe": self.getExec(),
                \ "args": "-a -b -c",
                \ "post_args": "--more --args",
                \ "tail": "2>/dev/null" })
<
The result is a 'makeprg' of the form: >
    <exe> <args> <fname> <post_args> <tail>
<
All arguments above are optional, and can be overridden by setting global
variables 'g:syntastic_<filetype>_<checker-name>_<option-name>' - even
parameters not specified in the call to makeprgBuild(). These variables also
have local versions 'b:syntastic_<filetype>_<checker-name>_<option-name>',
which take precedence over the global ones in the corresponding buffers.

If one of these variables has a non-empty default and you want it to be empty,
you can set it to an empty string, e.g.: >
    let g:syntastic_javascript_jslint_args = ""
<
                                        *'syntastic_<filetype>_<checker>_exe'*
The 'exe' is normally the same as the 'exec' attribute described above, in
which case it may be omitted. However, you can use it to add environment
variables, or to change the way the checker is run. For example this setup
allows you to run PC-Lint under Wine emulation on Linux: >
    let  g:syntastic_c_pc_lint_exec = "wine"
    let  g:syntastic_c_pc_lint_exe = "wine c:/path/to/lint-nt.exe"
<
To override the args and the tail: >
    let g:syntastic_c_pc_lint_args = "-w5 -Iz:/usr/include/linux"
    let g:syntastic_c_pc_lint_tail = "2>/dev/null"
<
The general form of the override options is: >
```

```
    syntastic_<filetype>_<checker>_<option-name>
<
For checkers that do not use the 'makeprgBuild()' function you will have to
look at the source code of the checker in question. If there are specific
options that can be set, these are usually documented in the wiki:

    https://github.com/scrooloose/syntastic/wiki/Syntax-Checkers

                             *'syntastic_<filetype>_<checker>_quiet_messages'*
In the same vein, 'g:syntastic_<filetype>_<checker-name>_quiet_messages' can
be used to restrict message filters to messages produced by specific checkers.
Example: >
    let g:syntastic_python_pylama_quiet_messages = {
        \ "type":  "style",
        \ "regex": '\m\[C03\d\d\]' }
<
See |syntastic_quiet_messages| for the syntax.

------------------------------------------------------------------------------
5.4 Sorting errors                                      *syntastic-config-sort*

                                       *'syntastic_<filetype>_<checker>_sort'*
Syntastic may decide to group the errors produced by some checkers by file,
then sort them by line number, then by type, then by column number. If you'd
prefer to see the errors in the order in which they are output by the external
checker you can set the variable |'g:syntastic_<filetype>_<checker>_sort'| to 0.

Alternatively, if syntastic doesn't reorder the errors produced by a checker
but you'd like it to sort them, you can set the same variable to 1.

There is also a local version |'b:syntastic_<filetype>_<checker>_sort'| of
this variable, that takes precedence over it in the buffers where it is
defined.

For aggregated lists (see |syntastic-aggregating-errors|) these variables are
ignored if |'syntastic_sort_aggregated_errors'| is set (which is the default).

==============================================================================
6. Notes                                                      *syntastic-notes*

------------------------------------------------------------------------------
6.1. Handling of composite filetypes                      *syntastic-composite*

Some Vim plugins use composite filetypes, such as "django.python" or
"handlebars.html". Normally, syntastic deals with this situation by splitting
the filetype in its simple components, and calling all checkers that apply.
If this behaviour is not desirable, you can disable it by mapping the
composite filetypes to simple ones using |'syntastic_filetype_map'|, e.g.: >
    let g:syntastic_filetype_map = { "handlebars.html": "handlebars" }
<
------------------------------------------------------------------------------
6.2 Editing files over network                                *syntastic-netrw*

The standard plugin |netrw| allows Vim to transparently edit files over
network and inside archives.  Currently syntastic doesn't support this mode
of operation.  It can only check files that can be accessed directly by local
checkers, without any translation or conversion.

------------------------------------------------------------------------------
6.3 The 'shellslash' option                              *syntastic-shellslash*

The 'shellslash' option is relevant only on Windows systems.  This option
determines (among other things) the rules for quoting command lines, and there
is no easy way for syntastic to make sure its state is appropriate for your
shell.  It should be turned off if your 'shell' (or |'syntastic_shell'|) is
"cmd.exe", and on for shells that expect an UNIX-like syntax, such as Cygwin's
"sh".  Most checkers will stop working if 'shellslash' is set to the wrong
value.

------------------------------------------------------------------------------
6.4 Saving Vim sessions                                    *syntastic-sessions*

If you use |:mksession| to save Vim sessions you should probably make sure to
```

```
remove option "blank" from 'sessionoptions': >
    set sessionoptions-=blank
<
This will prevent |:mksession| from saving |syntastic-error-window| as empty
quickfix windows.

==============================================================================
7. Compatibility with other software                 *syntastic-compatibility*

------------------------------------------------------------------------------
7.1 The csh and tcsh shells                                    *syntastic-csh*

The "csh" and "tcsh" shells are mostly compatible with syntastic.  However,
some checkers assume Bourne shell syntax for redirecting "stderr".  For this
reason, you should point |'syntastic_shell'| to a Bourne-compatible shell,
such as "zsh", "bash", "ksh", or even the original Bourne "sh": >
    let g:syntastic_shell = "/bin/sh"
<
------------------------------------------------------------------------------
7.2. Eclim                                                   *syntastic-eclim*

Syntastic can be used together with "Eclim" (see http://eclim.org/). However,
by default Eclim disables syntastic's checks for the filetypes it supports, in
order to run its own validation. If you'd prefer to use Eclim but still run
syntastic's checks, set |g:EclimFileTypeValidate| to 0: >
    let g:EclimFileTypeValidate = 0
<
It is also possible to re-enable syntastic checks only for some filetypes, and
run Eclim's validation for others. Please consult Eclim's documentation for
details.

------------------------------------------------------------------------------
7.3 The fish shell                                            *syntastic-fish*

At the time of this writing the "fish" shell (see http://fishshell.com/)
doesn't support the standard UNIX syntax for file redirections, and thus it
can't be used together with syntastic. You can however set |'syntastic_shell'|
to a more traditional shell, such as "zsh", "bash", "ksh", or even the
original Bourne "sh": >
    let g:syntastic_shell = "/bin/sh"
<
------------------------------------------------------------------------------
7.4. The fizsh shell                                         *syntastic-fizsh*

Using syntastic with the "fizsh" shell (see https://github.com/zsh-users/fizsh)
is possible, but potentially problematic. In order to do it you'll need to set
'shellredir' like this: >
    set shellredir=>%s\ 2>&1
<
Please keep in mind however that Vim can't take advantage of any of the
interactive features of "fizsh". Using a more traditional shell such as "zsh",
"bash", "ksh", or the original Bourne "sh" might be a better choice: >
    let g:syntastic_shell = "/bin/sh"
<
------------------------------------------------------------------------------
7.5 flagship                                              *syntastic-flagship*

The "flagship" Vim plugin (https://github.com/tpope/vim-flagship) has its
own mechanism of showing flags on the |'statusline'|. To allow "flagship"
to manage syntastic's statusline flag add the following |autocommand| to
your vimrc, rather than explicitly adding the flag to your |'statusline'| as
described in the |syntastic-statusline-flag| section above: >
    autocmd User Flags call Hoist("window", "SyntasticStatuslineFlag")
<
------------------------------------------------------------------------------
7.6. powerline                                           *syntastic-powerline*

The "powerline" Vim plugin (https://github.com/powerline/powerline) comes
packaged with a syntastic segment. To customize this segment create a file
~/.config/powerline/themes/vim/default.json, with a content like this: >
    {
        "segment_data" : {
            "powerline.segments.vim.plugin.syntastic.syntastic" : {
                "args" : {
                    "err_format" : "Err: {first_line} #{num} ",
                    "warn_format" : "Warn: {first_line} #{num} "
                }
            }
        }
    }
<
------------------------------------------------------------------------------
7.7. The PowerShell shell                               *syntastic-powershell*

At the time of this writing, syntastic is not compatible with using "Windows
PowerShell" (http://technet.microsoft.com/en-us/library/bb978526.aspx) as Vim's
'shell'.  You may still run Vim from 'PowerShell', but you do have to point
Vim's 'shell' to a more traditional program, such as "cmd.exe": >
    set shell=cmd.exe
<
------------------------------------------------------------------------------
7.8 python-mode                                             *syntastic-pymode*

Syntastic can be used along with the "python-mode" Vim plugin (see
https://github.com/klen/python-mode). However, they both run syntax checks by
default when you save buffers to disk, and this is probably not what you want.
To avoid both plugins opening error windows, you can either set passive mode
for python in syntastic (see |'syntastic_mode_map'|), or disable lint checks in
"python-mode", by setting |pymode_lint_on_write| to 0. E.g.: >
    let g:pymode_lint_on_write = 0
<
------------------------------------------------------------------------------
7.9. vim-auto-save                                     *syntastic-vim-auto-save*

Syntastic can be used together with the "vim-auto-save" Vim plugin (see
https://github.com/907th/vim-auto-save).  However, syntastic checks in active
mode only work with "vim-auto-save" version 0.1.7 or later.

------------------------------------------------------------------------------
7.10. vim-go                                                *syntastic-vim-go*

Syntastic can be used along with the "vim-go" Vim plugin (see
https://github.com/fatih/vim-go).  However, both "vim-go" and syntastic run
syntax checks by default when you save buffers to disk.  To avoid conflicts,
you have to either set passive mode in syntastic for the go filetype (see
|syntastic_mode_map|), or prevent "vim-go" from showing a quickfix window when
|g:go_fmt_command| fails, by setting |g:go_fmt_fail_silently| to 1.  E.g.: >
    let g:go_fmt_fail_silently = 1
<
------------------------------------------------------------------------------
7.11. vim-virtualenv                                  *syntastic-vim-virtualenv*

At the time of this writing, syntastic can't run checkers installed
in Python virtual environments activated by "vim-virtualenv" (see
https://github.com/jmcantrell/vim-virtualenv).  This is a limitation of
"vim-virtualenv".

------------------------------------------------------------------------------
7.12 YouCompleteMe                                             *syntastic-ycm*

Syntastic can be used together with the "YouCompleteMe" Vim plugin (see
http://valloric.github.io/YouCompleteMe/).  However, by default "YouCompleteMe"
disables syntastic's checkers for the "c", "cpp", "objc", and "objcpp"
filetypes, in order to allow its own checkers to run.  If you want to use YCM's
identifier completer but still run syntastic's checkers for those filetypes you
have to set |g:ycm_show_diagnostics_ui| to 0. E.g.: >
    let g:ycm_show_diagnostics_ui = 0
<
------------------------------------------------------------------------------
7.13 The zsh shell and MacVim                                  *syntastic-zsh*

If you're running MacVim together with the "zsh" shell (http://www.zsh.org/)
you need to be aware that MacVim does not source your .zshrc file, but will
source a .zshenv file.  Consequently you have to move any setup steps relevant
to the checkers you're using from .zshrc to .zshenv, otherwise your checkers
```

```
will misbehave when run by syntastic.  This is particularly important for
programs such as "rvm" (https://rvm.io/) or "rbenv" (http://rbenv.org/), that
rely on setting environment variables.

==============================================================================
8. About                                                     *syntastic-about*

The core maintainers of syntastic are:
    Martin Grenfell (GitHub: scrooloose)
    Gregor Uhlenheuer (GitHub: kongo2002)
    LCD 047 (GitHub: lcd047)

Find the latest version of syntastic at:

    http://github.com/scrooloose/syntastic

==============================================================================
9. License                                                 *syntastic-license*

Syntastic is released under the WTFPL.
See http://sam.zoy.org/wtfpl/COPYING.

 vim:tw=78:sw=4:ft=help:norl:
```