SublimeLinter Documentation

Release 3.4.24

The SublimeLinter Community

Apr 11, 2017

Contents

SublimeLinter is a plugin for Sublime Text 3 that provides a framework for linting code. Whatever language you code in, SublimeLinter can help you write cleaner, better, more bug-free code. SublimeLinter has been designed to provide maximum flexibility and usability for users and maximum simplicity for linter authors.

The documentation for SublimeLinter is divided into two sections: one for users, and one for developers who would like to create their own linter plugins.

User Documentation | *Developer Documentation*

The SublimeLinter source is available on github.

CHAPTER 1

Support

Please use the SublimeLinter issue tracker for support and bug reporting but before opening a new ticket, verify there isn't already a ticket in the SublimeLinter issue tracker or the now deprecated SublimeLinter google group.

CHAPTER 2

Be Part of the Team

Hundreds of hours have been spent writing and documenting SublimeLinter to make it the best it can be — easy to use, easy to configure, easy to update, easy to extend. If you depend on SublimeLinter to make your coding life better and easier, please consider making a donation to help fund development and support. Thank you!

CHAPTER 3

User Documentation

About SublimeLinter

SublimeLinter is a linting framework. The actual linting is done by separate Sublime Text 3 plugins, which can be installed via Package Control.

What is a linter?

A linter is a small program that checks code for stylistic or programming errors. Linters are available for most syntaxes, from Python to HTML. Here is a sample list of syntaxes and their linters:

Syntax	Linter
Python	flake8
JavaScript	jshint
CSS	csslint
Ruby	`ruby -wc`

SublimeLinter does not do the linting itself; it acts as a host for linting plugins. The linting plugins themselves usually do not perform linting either; they just act as a bridge between the code you type in Sublime Text and the actual linter.

Note that SublimeLinter is not limited to a single linter plugin per syntax — you are free to install multiple linter plugins for a syntax, and all of them will run when you edit a file in that syntax.

In addition, SublimeLinter supports multiple syntaxes in a single file, which is common when editing HTML. For example, a single HTML file may contain embedded CSS, JavaScript, and PHP. SublimeLinter will lint all of the embedded code using the appropriate linter plugin.

Why do I need a linter?

Programming is hard. We are bound to make mistakes. The big advantage of using SublimeLinter is that your code can be linted **as you type** (before saving your changes) and any errors are highlighted **immediately**, which is considerably

easier than saving the file, switching to a terminal, running a linter, reading through a list of errors, then switching back to Sublime Text to locate the errors!

In addition, linters can help to enforce coding standards, find unused variables, and even make coffee for you — okay, so maybe they can't make coffee. But they are an invaluable part of your programming toolkit.

Ready to get started? The next step is to *install SublimeLinter* and the linter plugins you need.

Installation

SublimeLinter itself is only a **framework** for linters. The linters are distributed as independent Sublime Text 3 plugins.

SublimeLinter (and the linter plugins) can be installed via a plugin called Package Control or from source. I **strongly** recommend that you use Package Control! Not only does it ease installation, but more importantly it automatically updates the plugins it installs, which ensures you will get the latest features and bug fixes.

Upgrading from previous versions

If you are upgrading to SublimeLinter 3 from a previous version (including an ST3 branch), please be aware that SublimeLinter 3 is a complete rewrite and is **not** a drop-in replacement. The basic functionality is the same, but there are key differences:

- Linters are not included, you must install them — and the linter binaries they depend on — separately. Linters can be found in Package Control with the name "SublimeLinter-<linter>", for example "SublimeLinter-jshint".
- *Settings* do not work in the same way.
- You no longer need to use path settings voodoo to find linter executables. Anything in your system `PATH` is *found automatically*.
- Most settings can be configured *via menus and the Command Palette*, which you are encouraged to do.
- There are dozens of new features.

Warning: SublimeLinter 3 is **not** a drop-in replacement for earlier versions. If you are coming from an earlier version of SublimeLinter and don't read the documentation, you will get confused and frustrated. **Read the docs.**

Installing via Package Control

To install SublimeLinter via Package Control, follow these steps:

1. Open the Command Palette (`cmd+shift+p` on Mac OS X, `ctrl+shift+p` on Linux/Windows).
2. Type `install` and select `Package Control: Install Package` from the Command Palette. There will be a pause of a few seconds while Package Control finds the available packages.
3. When the list of available packages appears, type `linter` and select `SublimeLinter`. **Note:** The github repository name is "SublimeLinter3", but the plugin name remains "SublimeLinter".
4. After a few seconds SublimeLinter will be installed and loaded. Depending on your setup, you may see some prompts from SublimeLinter. For more information on SublimeLinter's startup actions, see *Startup actions*.
5. You will see an install message. After reading the message, restart Sublime Text 3.

If you have a previous installation of SublimeLinter via Package Control, including "SublimeLinter Beta", it should be updated correctly from the new version. If something goes wrong, use Package Control to remove SublimeLinter and then follow the steps above to install again.

Note: SublimeLinter 3 does **not** include linters, unlike earlier versions. You **must** install linter plugins separately. They can be found in Package Control with the name "SublimeLinter-<linter>", for example "SublimeLinter-jshint".

Installing from source

I **very strongly** discourage you from installing from source. There is **no** advantage to installing from source vs. using Package Control. In fact, there are several disadvantages, including no automatic updates, no update messages, etc.

If you insist on installing from source, please do not do so unless you are comfortable with the command line and know what you are doing. To install SublimeLinter from source, do the following:

1. Quit Sublime Text.
2. If you have a previous source installation at `Packages/SublimeLinter`, delete it.
3. Type in a terminal:

```
cd '/path/to/Sublime Text 3/Packages'
git clone https://github.com/SublimeLinter/SublimeLinter3.git SublimeLinter
```

4. Restart Sublime Text 3.

Please consider using Package Control instead!

Linter plugins

Regardless of how you install SublimeLinter, once it is installed you will want to install linters appropriate to the languages in which you will be coding.

Warning: Linter plugins are **not** part of SublimeLinter 3.

Linter plugins are separate Sublime Text 3 plugins that are hosted in separate repositories. There are a number of officially supported linter plugins in the SublimeLinter organization. There are third party linters available as well.

Again, I **strongly** recommend that you use Package Control to locate and install linter plugins. To install linter plugins in Package Control, do the following:

1. Open the Command Palette (`cmd+shift+p` on Mac OS X, `ctrl+shift+p` on Linux/Windows).
2. Type `install` and select `Package Control: Install Package` from the Command Palette. There will be a pause of a few seconds while Package Control finds the available packages.
3. When the list of available packages appears, type `sublimelinter-`. You will see a list of plugins whose names begin with "SublimeLinter-". Click on the plugin you wish to install.
4. After a few seconds the plugin will be installed and loaded. You will then see an install message with instructions on what you should do to complete the installation.
5. After reading the instructions, restart Sublime Text 3.

Warning: Most linter plugins require you to install a linter binary or library and *configure your PATH* so that SublimeLinter can find it. You **must** follow the linter plugin's installation instructions to successfully use it.

If you have problems installing or configuring SublimeLinter. First read the *Troubleshooting guide*. Then if necessary, report your problem on the SublimeLinter issue tracker.

Read the docs!

An enormous amount of time and effort went into creating SublimeLinter and this documentation. **Before** you launch Sublime Text 3 with SublimeLinter installed, please take the time to read the *Usage* documentation to understand what happens when SublimeLinter loads and how it works. Otherwise you won't get the most out of it!

Usage

SublimeLinter is designed to work well out of the box, but there are many ways to customize it to your taste. Before we get to that, though, let's take a look at how SublimeLinter works.

Startup actions

When SublimeLinter is loaded by Sublime Text 3, it performs a number of actions to initialize its environment:

Settings

The default settings are loaded from the plugin and merged with the settings in `Packages/User/SublimeLinter.sublime-settings`. For more information on SublimeLinter settings, see *Settings*.

Color scheme

SublimeLinter has to convert color schemes for its use. For more information, see *Choosing color schemes*.

Customized syntax definitions

SublimeLinter supports linting of embedded syntaxes, such as JavaScript and CSS within an HTML file, by specifying a scope to which the linter is limited. Unfortunately the stock syntax definitions that ship with Sublime Text 3 incorrectly classify the scope of embedded languages, which leads to false errors during linting. To solve this problem, at load time SublimeLinter installs fixed versions of the `HTML` and `HTML (Rails)` syntax packages in the `Packages` directory.

Note: The first time the fixed syntaxes are installed, you may need to restart Sublime Text 3 for them to be applied to source files in those syntaxes.

Assigning linters

When a file is opened in Sublime Text 3, SublimeLinter checks the syntax assigned to the file (Python, JavaScript, etc.), and then uses that name (lowercased) to locate any linters (there may be several) that have advertised they can lint that syntax. Any found linters are assigned to that *view* of the file. SublimeLinter assigns separate linter instances to each view, even if there are multiple views of the same file.

Linting

Here's where the magic happens.

When you activate or make any modifications to a file, the following sequence of events occurs:

- SublimeLinter checks to see if the syntax of the file has changed; and if so, reassigns linters to the view.
- If the **lint mode** is `background`, a lint request is added to a threaded queue with a delay. The delay is there to prevent lints from occurring instantly on every keystroke — you don't want the linter complaining too much while you are typing, it quickly becomes annoying. The delay is there to allow a little idle time before a lint occurs.

  For more information on lint modes, see *Lint Modes*. The delay can be configured, for more information see *Global Settings*.
- The lint request is eventually pulled off the queue after the given delay. If the view it belongs to has been modified since the lint request was made, the request is discarded, since another lint request was generated when the view was modified.
- Each of the linters assigned to the base syntax of the view is run with the current text of the view. The linter calls an external linter binary (such as jshint), or if the linter is python-based (such as flake8), it may directly call a python linting library.
- If any linters assigned to the view support embedded code and that embedded code is found, the linters are run with the appropriate embedded code.
- Each linter adds a set of regions indicating the portions of the source code that generated errors or warnings.
- When all of the linters have finished, if the view has still not been modified since the initial lint request, all of the error and warning regions are aggregated and drawn according to the currently configured *mark style* and *gutter theme*. Errors and warnings are marked with separate colors and gutter icons to make it easy to see which is which.

How linter executables are located

When calling a system linter binary, the user's `PATH` environment variable is used to locate the binary. On Windows, the `PATH` environment variable is used as is. On Mac OS X and Linux, if the user's shell is `bash`, `zsh`, or `fish`, a login shell is used to get the `PATH` value. If you are using a shell other than the ones just mentioned, `PATH` effectively becomes:

```
/bin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/php/bin:/usr/local/php5/bin
```

Warning: On Mac OS X and Linux, special care must be taken to ensure your `PATH` is set up in such a way that SublimeLinter can read it. For more information, see *Debugging PATH problems*.

In addition to the `PATH` SublimeLinter reads from the system, any directories in the global `"paths"` setting for the current platform are searched when attempting to locate a binary. For more information, see the *Global Settings* documentation.

Python paths

When locating python and python scripts such as flake8, SublimeLinter goes through a special process. For more information, see *the @python meta setting*.

Disabling all linters

There may be times when you want to turn off all linting. To do so, bring up the Command Palette and type `disable`. Among the commands you should see `SublimeLinter: Disable Linting`. If that command is not highlighted, use the keyboard or mouse to select it.

Once you do this, all linters are disabled and all error marks are cleared from all views. To re-enable linting, follow the same steps as above, but select `SublimeLinter: Don't Disable Linting`. Note that this does not enable all linters; if you have *disabled individual linters* in the settings, they will remain disabled.

Toggling linters

You can quickly toggle a linter on or off. To do so:

1. Bring up the Command Palette (`cmd+shift+p` on Mac OS X, `ctrl+shift+p` on Linux/Windows) and type `toggle`, `disable`, or `enable` according to what you want to view all linters, only enabled linters, or only disabled linters.
2. Among the commands you should see `SublimeLinter: Toggle Linter`, `SublimeLinter: Disable Linter` or `SublimeLinter: Enable Linter`, depending on what you typed. If the command is not highlighted, use the keyboard or mouse to select it.
3. Once you select the command, a list of the relevant linters appears. If you chose `SublimeLinter: Disable Linter`, only the enabled linters appear in the list. If you chose `SublimeLinter: Enable Linter`, only the disabled linters appear.
4. Select a linter from the list. It will be toggled, disabled or enabled, depending on the command you chose.

Choosing color schemes

In order to color errors, warnings and gutter icons correctly, SublimeLinter relies on specific named colors being available in the current color scheme. Whenever a color scheme is loaded — either implicitly at startup or by selecting a color scheme — SublimeLinter checks to see if the color scheme contains its named colors. If not, it adds those colors to a copy of the color scheme, writes it to the `Packages/User/SublimeLinter` directory with a " (SL)" suffix added to the filename, and switches to the modified color scheme.

For example, if you select `Preferences > Color Scheme > Color Scheme - Default > Monokai`, SublimeLinter will convert it, write the converted color scheme to `Packages/User/SublimeLinter/Monokai (SL).tmTheme`, and switch to that color scheme. If you then open the `Preferences > Color Scheme` menu, `User > SublimeLinter > Monokai (SL)` is checked.

> **Warning:** If you choose an unconverted color scheme and an existing converted color scheme exists in `Packages/User/SublimeLinter`, it will be overwritten.

Note: If you ever want to clean up, and delete all the SublimeLinter made color schemes not being used in the settings, simply use the `SublimeLinter: Clear Color Scheme Folder` command from the Command Pallete, Tools menus, or Context menu.

For more information on customizing the colors used by SublimeLinter, see *Global Settings*.

User interface

There are four main aspects to the SublimeLinter user interface:

- *Lint mode* — The lint mode determines when linting occurs.
- *Mark style* — The mark style determines how errors are marked in the text.
- *Gutter theme* — The gutter theme determines how lines with errors are marked in the gutter.
- *Navigating errors* — Once linters find errors in your code, you can quickly and easily navigate through them.

Lint Modes

Which events trigger linting depends on the **lint mode**. The lint mode is a *global setting* that applies to all views in all windows. Which mode you choose is a matter of preference.

The modes

There are four lint modes in SublimeLinter: *background*, *load/save*, *save only*, and *manual*.

Background

In **background** mode, lint requests are generated for every modification of a view, as well as on file loading and saving. This is the default mode. Remember that background lint requests only trigger a lint if the associated view has not been modified when the request is pulled off the queue (see *Linting*).

Pros

- Immediate feedback on errors.

Cons

- If the delay is too short, you will end up with a lot of false positives.
- Some linters are unavoidably slow and can affect the performance of editing. In such cases you may want to use a different lint mode.

Load/Save

In **load/save** mode, a file is linted and errors are marked whenever it is loaded and saved. After loading or saving, any modifications to the file clear all marks.

Pros

- There are no distractions from error marks while typing, since errors are only displayed when you are ready to save your work.
- This mode avoids performance issues resulting from slow linters.

Cons

- You have to manually go through the errors, unless the *"show_errors_on_save"* setting is on. For more information on that setting, see *Showing errors on save* below.

Save only

save only mode is the same as **load/save** mode, but linting only occurs when a file is saved, not when it is loaded.

Pros

- If you have very large files that are relatively slow to lint, and you tend to leave many files open when quitting Sublime Text 3, in **background** or **load/save** mode, those files will be linted when Sublime Text 3 starts up, which potentially could take several seconds. **save only** mode avoids this problem by linting only when saving a file.

Cons

- Your files will not be linted when loaded.

Manual

In **manual** mode, linting only occurs when you manually initiate a lint. After linting, any modifications to the file clear all marks.

Pros

- If you are fairly confident in your coding and only want to lint occasionally, this is the mode for you.

Cons

- You may forget to lint!

Choosing a lint mode

There are three ways to select a lint mode:

Command Palette

Bring up the Command Palette and type `mode`. Among the commands you should see `SublimeLinter: Choose Lint Mode`. If that command is not highlighted, use the keyboard or mouse to select it. A list of the available lint modes appears with the current mode highlighted. Type or click to select the lint mode you would like to use.

Tools menu

At the bottom of the Sublime Text 3 `Tools` menu, you will see a `SublimeLinter` submenu. Select `SublimeLinter > Lint Modes` and then select a mode from the submenu.

Context menu

If you right-click (or Control-click on OS X) within a file view, you will see a `SublimeLinter` submenu at the bottom of the context menu. Select `SublimeLinter > Lint Modes` and then select a mode from the submenu.

Once you have selected a new lint mode, all of the open views are redrawn: if the mode is **background**, all views are linted, otherwise all errors marks are cleared. The lint mode you select is saved in your user settings, so it will still be active after restarting Sublime Text.

Manually linting

If you select **manual** lint mode, you must manually lint your files. To do so, do one of the following:

Command Palette

Bring up the Command Palette and type `lint`. Among the commands you should see `SublimeLinter: Lint This View`. If that command is not highlighted, use the keyboard or mouse to select it.

Tools menu

At the bottom of the Sublime Text `Tools` menu, you will see a `SublimeLinter` submenu. Select `SublimeLinter > Lint This View`.

Context menu

If you right-click (or Control-click on OS X) within a file view, you will see a `SublimeLinter` submenu at the bottom of the context menu. Select `SublimeLinter > Lint This View`.

Keyboard

On Mac OS X, press `Command+Control+L`. On Linux/Windows, press `Control+K, Control+L`.

Showing errors on save

When the lint mode is not **background**, you may wish to automatically lint a file and display any errors whenever it is saved. SublimeLinter makes it easy to do this with the *"show_errors_on_save"* setting. By default, this setting is off. To turn this setting on, do one of the following:

Command Palette

Bring up the Command Palette and type `show`. Among the commands you should see `SublimeLinter: Show Errors on Save`. If that command is not highlighted, use the keyboard or mouse to select it.

Tools menu

At the bottom of the Sublime Text `Tools` menu, you will see a `SublimeLinter` submenu. If that item is not checked, select `SublimeLinter > Show Errors on Save`.

Context menu

If you right-click (or Control-click on OS X) within a file view, you will see a `SublimeLinter` submenu at the bottom of the context menu. If that item is not checked, select `SublimeLinter > Show Errors on Save`.

Note: As of this writing, the Linux version of Sublime Text 3 does not check menu items, so you cannot tell by examining the menu item whether this option is on or off.

Once you have turned `"show_errors_on_save"` on, every time a file is saved, it is linted and any errors are displayed in the *Show All Errors* Quick Panel.

To turn `"show_errors_on_save"` off, follow the instructions above for turning it on, but you will see "Don't Show Errors on Save" instead of "Show Errors on Save".

Mark Styles

When linting is done, SublimeLinter marks errors in three ways: the suspect code itself is marked, the line on which the code occurs is marked *in the gutter*, and the status bar (at the bottom of the window) gives information on the errors based on the current selection. Code marks and gutter marks can be configured separately.

Status bar info

If there are linting errors in the current view, the status bar is updated as you change the selection.

- If the first character in the first selection **is not** on a line with an error, the status bar will indicate the total number of errors, for example "7 errors".
- If the first character in the first selection **is** on a line with an error, the status bar will indicate the range of errors on that line, along with all of the error messages for those errors, separated by semicolons, for example "2-3 of 7 errors: Multiple spaces after keyword; Undefined name 'bar'".

Code mark styles

There are five different code mark styles available: **fill**, **outline**, **solid underline**, **squiggly underline**, and **stippled underline**. In addition, you can choose to turn code marks off completely if you just want to see gutter marks.

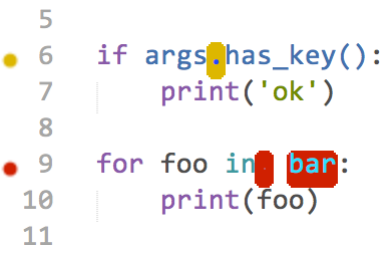
There are actually two types of marks: errors and warnings. Most linters classify the issues they find as errors or warnings, and the linter plugins in turn decide whether to report them to SublimeLinter as errors or warnings. Errors and warnings are drawn in separate, *configurable colors*. This helps you to visually identify which marks are errors and which are warnings.

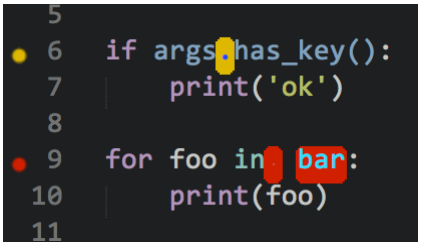
Which mark style you use is a matter of taste. Below are samples of each mark style using a light and dark color scheme (Tomorrow and Tomorrow-Night). The colored dots on the left are the default gutter marks.

Note: As you can see below, there is currently a limitation in Sublime Text 3 that prevents underlines from drawing under non-word characters (such as whitespace). Take this into account when choosing a mark style.

---

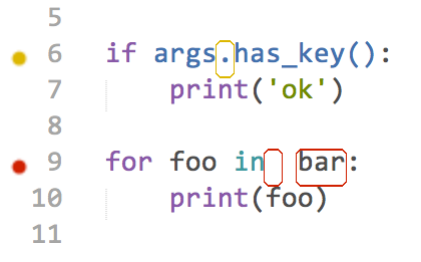
fill

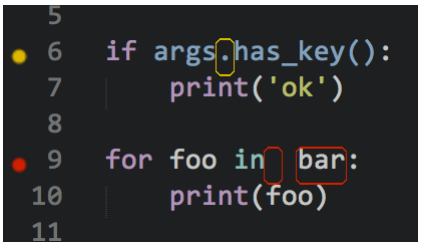




---

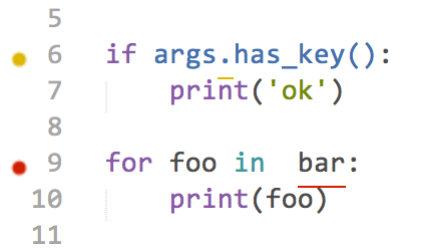
outline

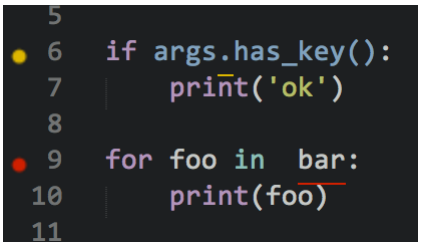




---

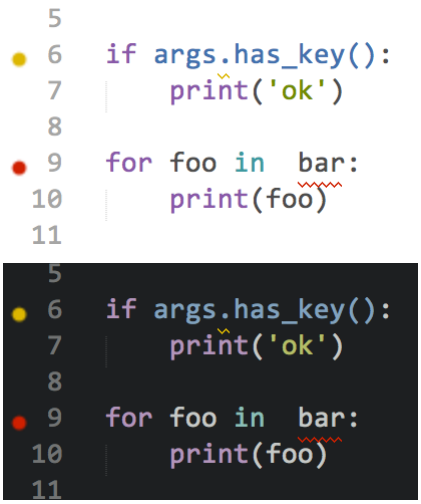
solid underline





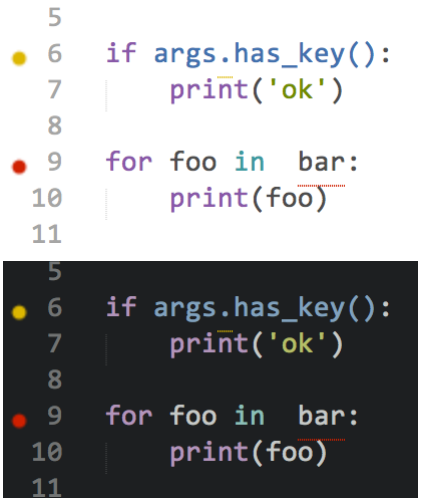
---

squiggly underline



---

stippled underline



Choosing a mark style

There are three ways to select a mark style:

Command Palette

Bring up the Command Palette and type `mark`. Among the commands you should see `SublimeLinter: Choose Mark Style`. If that command is not highlighted, use the keyboard or mouse to select it. A list of the available mark styles appears with the current mark style highlighted. Type or click to select the mark style you would like to use.

Tools menu

At the bottom of the Sublime Text 3 `Tools` menu, you will see a `SublimeLinter` submenu. Select `SublimeLinter > Mark Styles` and then select a mark style from the submenu.

Context menu

If you right-click (or Control-click on OS X) within a file view, you will see a `SublimeLinter` submenu at the bottom of the context menu. Select `SublimeLinter > Mark Styles` and then select a mark style from the submenu.

Once you have selected a new mark style, all of the open views are redrawn with the new style. The mark style you select is saved in your *user settings*, so it will still be active after restarting Sublime Text 3.

No-column mode

When a linter reports an error with no column information, by default a mark is put in the gutter but no text is highlighted. You may also choose to highlight the entire line when it there is no column information. To change the no-column highlighting mode, do one of the following:

Command Palette

Bring up the Command Palette and type `column`. Among the commands you should see either `SublimeLinter: No Column Highlights Entire Line` or `SublimeLinter: No Column Only Marks Gutter`. If the command is not highlighted, use the keyboard or mouse to select it. Choosing the command toggle the setting.

Tools menu

At the bottom of the Sublime Text 3 `Tools` menu, you will see a `SublimeLinter` submenu. Select `SublimeLinter > Mark Styles` and then select `No column Highlights Entire Line` from the submenu.

Context menu

If you right-click (or Control-click on OS X) within a file view, you will see a `SublimeLinter` submenu at the bottom of the context menu. Select `SublimeLinter > Mark Styles` and then select `No Column Highlights Entire Line` from the submenu.

Once you have selected a highlight mode, all of the open views are relinted with the new mode. The no-column mode you select is saved in your *user settings*, so it will still be active after restarting Sublime Text 3.

Gutter Themes

When linting is done, SublimeLinter marks errors in two ways: the suspect code itself is *marked*, and the line on which the code occurs is marked in the gutter. Code marks and gutter marks are configured separately.

Gutter theme structure

There are actually two types of gutter marks: errors and warnings. This helps you to visually identify which marks are errors and which are warnings.

Gutter marks are drawn using PNG images. If a theme is *colorized*, the images are tinted with the current *"error_color"* or *"warning_color"* colors in your settings. Otherwise the images are drawn as is.

Standard gutter themes

Which gutter theme you use is a matter of taste. Below is a list of the built in gutter themes that come with SublimeLinter. (The names come from the third party icon set from which the icons were chosen.) If none of these suit you, you can easily *create your own*.

Note: Colorized icons are in fact mostly white; they are displayed here as they appear when drawn by SublimeLinter, tinted with the error or warning color. Creating the icons white allows the tint color to come through unchanged.

Name	Error/Warning
Blueberry - cross	
Blueberry - round	
Circle	 [colorized]
Danish Royalty	
Default	[colorized]
Hands	
Knob - simple	
Knob - symbol	
Koloria	
ProjectIcons	

Choosing a gutter theme

There are three ways to choose a gutter theme:

Command Palette

Bring up the Command Palette and type `gutter`. Among the commands you should see `SublimeLinter: Choose Gutter Theme`. If that command is not highlighted, use the keyboard or mouse to select it.

A list of the available gutter themes appears with the current gutter theme highlighted. Below each gutter theme name is an indication of whether the theme is a standard SublimeLinter theme or a user theme, as well as whether the theme is colorized.

If you type or use the arrow keys to move through the list, the current gutter theme will change dynamically to the currently selected theme. If you have a view open with gutter marks, this allows you to preview other themes. Pressing `Return/Enter` or clicking on a theme will commit that change. Pressing `Escape` will revert to the theme in use before the Command Palette opened.

Tools menu

At the bottom of the Sublime Text 3 `Tools` menu, you will see a `SublimeLinter` submenu. Select `SublimeLinter > Choose Gutter Theme...` and then follow the instructions for selecting from the Command Palette.

Context menu

If you right-click (or Control-click on OS X) within a file view, you will see a `SublimeLinter` submenu at the bottom of the context menu. Select `SublimeLinter > Choose Gutter Theme...` and then follow the instructions for selecting from the Command Palette.

Once you have selected a new gutter theme, all of the open views are redrawn with the new theme. The gutter theme you select is saved in your user settings, so it will still be active after restarting Sublime Text 3.

Creating a gutter theme

With SublimeLinter, you are free to create or install new gutter themes. You can mix and match the existing images, or use entirely new images. SublimeLinter's built in gutter themes can be found in `Packages/SublimeLinter/gutter-themes`.

A gutter theme is simply a directory that contains the following three files:

<name>.gutter-theme

This file is what SublimeLinter uses to locate gutter themes, and **<name>** (without the < >) is used for the gutter theme name (the parent directory name can be anything). If a gutter theme is colorized, this file should contain the following JSON:

```
{
    "colorize": true
}
```

If the gutter theme is not colorized, the file may be empty, or it may include the same JSON but set `"colorized"` to `false`.

error.png

This image is displayed in the gutter on any line that has errors.

warning.png

This image is displayed in the gutter on any line that has warnings but no errors; errors always have precedence over warnings.

When you choose a gutter theme, SublimeLinter looks for any directory with these three files within `Packages`, `Packages/User`, or `Installed Packages`. Within `Installed Packages`, the gutter theme must be somewhere within a compressed `.sublime-package` file.

Gutter images

Sublime Text 3 scales gutter images to 16 x 16. For best results with Retina displays, gutter images should be 32 x 32 at 72dpi.

If your gutter icons will be colorized, they should be mostly white, with shades of gray used to create shadow areas. The entire image should be grayscale, so that the error and warning colors do not change when they are applied to the icons.

Installing gutter themes

Third party gutter themes may be searched for and installed via Package Control, or if you have created your own gutter theme, by placing the gutter theme directory in the Sublime Text 3 `Packages` or `Packages/User` directory. Once you have installed the new gutter theme, follow the instructions above to *choose the theme*. That's all there is to it!

Navigating Errors

If errors occur during linting, there are several ways available to quickly navigate through them. There are three commands for navigating errors: `Next Error`, `Previous Error`, and `Show All Errors`.

Note: These commands do **not** lint the current view, they only navigate. If your *lint mode* is not **background**, you will have *manually lint* if you want to see the lint errors for the current state of the file.

Accessing navigation commands

You can access these commands in several ways.

Command Palette

Bring up the Command Palette and type `next`, `previous`, or `show`. Among the commands you should see `SublimeLinter: Next Error`, `SublimeLinter: Previous Error`, or `SublimeLinter: Show All Errors`. If the command is not highlighted, use the keyboard or mouse to select it.

Tools menu

At the bottom of the Sublime Text `Tools` menu, you will see a `SublimeLinter` submenu. Select `SublimeLinter > Next Error`, `SublimeLinter > Previous Error`, or `SublimeLinter > Show All Errors`.

Context menu

If you right-click (or Control-click on OS X) within a file view, you will see a `SublimeLinter` submenu at the bottom of the context menu. Select `SublimeLinter > Next Error`, `SublimeLinter > Previous Error`, or `SublimeLinter > Show All Errors`.

Keyboard

You can also use the keyboard to access the navigation commands.

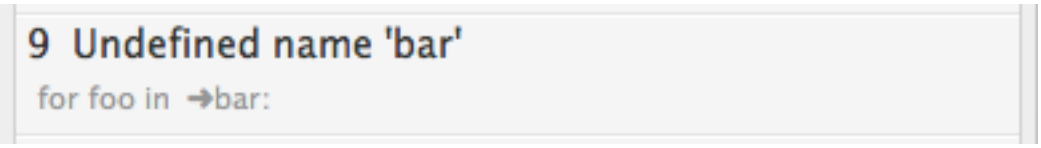
Command	Mac OS X	Linux/Windows
Next Error	Command+Control+E	Control+K, n
Previous Error	Command+Control+Shift+E	Control+K, p
Show All Errors	Command+Control+A	Control+K, a

If there are any errors in the current view, the next/previous error will be selected, or all errors will be displayed in a Quick Panel. If there are no errors, an alert will appear which indicates there are no errors.

Note: If the *"wrap_find"* setting is `false`, the search for the next/previous error will stop at the bottom/top of the view.

Showing all errors

When you select the `Show All Errors` command, all errors in the view are displayed in a Quick Panel. Each item displays the line number and error message on the first line, and the source code on the second line:



Note that a is inserted in the source code at the position where the error occurs.

Selecting an error and typing `Return/Enter` or clicking on an error navigates directly to that error.

Making warnings passive

If you don't want warnings to be displayed in the `Show All Errors` Quick Panel, you can make use of the `Make Warnings Passive` command to not have them included in the window. This command determines if the *"passive_warnings"* setting is `true` or `false`. You can access this command in several ways.

Command Palette

Bring up the Command Palette and type `warnings`. Among the commands you should see `SublimeLinter: Make Warnings Passive`, or if it is already turned on you should see `SublimeLinter: Don't Make Warnings Passive`. If the command is not highlighted, use the keyboard or mouse to select it.

Tools menu

At the bottom of the Sublime Text `Tools` menu, you will see a `SublimeLinter` submenu. Select `SublimeLinter > Make Warnings Passive`.

Context menu

If you right-click (or Control-click on OS X) within a file view, you will see a `SublimeLinter` submenu at the bottom of the context menu. Select `SublimeLinter > Make Warnings Passive`..

Note: If the *"passive_warnings"* setting is `true`, warnings will still be visible in the gutter and view.

Settings

Most of the settings that affect SublimeLinter's user interface are available through commands and menu items. But there are some SublimeLinter settings that can only be changed manually in settings files. In addition to SublimeLinter's settings, each linter may define its own settings, which usually must be changed manually as well.

Settings stack

When SublimeLinter (or a linter plugin) asks for a setting value, SublimeLinter merges settings from several sources to calculate the value. The settings sources can be visualized as a stack, with settings at the top taking precedence over settings lower down:

```
Inline overrides
Inline settings
.sublimelinterrc settings
Project settings
User settings
Default settings
```

After the default, user, and project settings are merged, *tokens* are replaced within the settings. Each of the settings sources is covered in detail *below*.

Setting types

There are three distinct types of settings:

Global

Global settings control SublimeLinter's behavior and apply to all views. For example, the `"error_color"` setting determines the color of error marks and applies to all views. Defaults for all global settings are defined in the SublimeLinter default settings and may only be modified within the user settings.

Linter

Linter settings apply only to a specific named linter. Linter settings are always defined within a `"linters"` object whose subobjects are named according to the lowercase class name of the linter. For an example, see the *user settings* sample below.

Meta

Meta settings are special settings whose names begin with `"@"`. When defined at the global level, their value is applied to the settings of every linter. For example, when you select the *Disable Linting* command, SublimeLinter sets the meta setting `"@disable"` to `true` at the global level, which is applied to all linters.

Meta settings may also be set within a single linter's settings, and in that case they apply only to that linter.

Note: A meta setting at the global level overrides the same linter meta setting. For example, even if `"@disable"` is `true` within a linter's settings, setting `"@disable"` to `false` at the global level will override the linter setting and enable that linter.

Settings sources

Let's take a look at each of the settings sources in the stack, starting from the base level and working our way up.

Default settings

Default settings are defined by SublimeLinter and by each linter. You should **never** edit the default settings, as your changes may be overwritten the next time SublimeLinter is updated. You should **always** edit the user settings (or other settings higher in the stack).

User settings

User settings are located in `Packages/User/SublimeLinter.sublime-settings`. You should consider this to be the global settings for SublimeLinter and its linters. To make it easier to remember what settings are available, whenever you open the user settings, they are filled in with any missing default settings from SublimeLinter and from all installed linters.

Here is an example user settings file:

```
{
    "user": {
        "debug": false,
        "delay": 0.25,
        "error_color": "D02000",
        "gutter_theme": "Packages/SublimeLinter/gutter-themes/Knob/simple/Knob -␣
↪simple.gutter-theme",
```

```
        "gutter_theme_excludes": [],
        "lint_mode": "background",
        "linters": {
            "csslint": {
                "@disable": false,
                "args": [],
                "excludes": []
            },
            "flake8": {
                "@disable": false,
                "args": [],
                "excludes": [],
                "ignore": "",
                "max-complexity": -1,
                "max-line-length": null,
                "select": ""
            }
        },
        "mark_style": "outline",
        "paths": {
            "*": [],
            "linux": [],
            "osx": [],
            "windows": []
        },
        "python_paths": {
            "linux": [],
            "osx": [],
            "windows": []
        },
        "rc_search_limit": 3,
        "show_errors_on_save": false,
        "show_marks_in_minimap": true,
        "syntax_map": {
            "php": "html"
        },
        "warning_color": "DDB700",
        "wrap_find": true
    }
}
```

All of these values were initially filled in by SublimeLinter when the file was first opened. After that, it's just a matter of changing the settings. There are three easy ways to open the user settings:

Command Palette

Bring up the Command Palette and type `prefs`. Among the commands you should see `Preferences: SublimeLinter Settings - User`. If that command is not highlighted, use the keyboard or mouse to select it.

Tools menu

At the bottom of the Sublime Text `Tools` menu, you will see a `SublimeLinter` submenu. Select `SublimeLinter > Open User Settings`.

Context menu

If you right-click (or Control-click on OS X) within a file view, you will see a `SublimeLinter` submenu at the bottom of the context menu. Select `SublimeLinter > Open User Settings`.

Project settings

SublimeLinter project settings are defined by a `"SublimeLinter"` object within Sublime Text's project settings. These settings apply to all files within the project.

Note: Only meta-settings and linter settings are recognized in project settings.

Project settings are opened from the `Project > Edit Project` menu. Here is an example project settings file with some SublimeLinter settings:

```
{
    "folders":
    [
        {
            "follow_symlinks": true,
            "path": "/Users/aparajita/Projects/SublimeLinter"
        }
    ],
    "SublimeLinter":
    {
        "linters":
        {
            "flake8": {
                "excludes": [
                    "*/test/**"
                ],
                "ignore": "W"
            }
        }
    }
}
```

Note: Be sure you are **not** putting the `"SublimeLinter"` object inside the `settings` object. They should be sibling objects in the root document.

Unlike user settings, project settings are not filled in by SublimeLinter; you are responsible for adding any settings you wish to apply to files in the project.

.sublimelinterrc settings

Sometimes it is useful to apply settings to files in a particular directory (or subdirectory thereof). For example, you may want to apply specific settings to a directory that is not part of a Sublime Text project. Or you may wish to apply specific settings to a directory within a Sublime Text project.

SublimeLinter allows per-directory settings through `.sublimelinterrc` files ("rc" stands for "runtime configuration").

Note: Only meta-settings and linter settings are recognized in `.sublimelinterrc` files.

When reading the settings for a given file, SublimeLinter does the following:

- Searches in the file's directory for a `.sublimelinterrc` file.

- If it is not found, the parent directories are searched until the root directory is reached or until the maximum number of search directories (including the file's directory) are searched.

The maximum number of search directories is determined by the *"rc_search_limit" setting*. By default, the limit is 3. Setting `"rc_search_limit"` to `null` means the search will stop only at the root directory. Setting it to `0` disables the search for `.sublimelinterrc` entirely for the scope of the settings file in which `"rc_search_limit"` is found. This can be useful for projects that are hosted on slow remote filesystems.

The first `.sublimelinterrc` file found is used; SublimeLinter does **not** merge multiple `.sublimelinterrc` files in the search path together.

So, for example, let's assume we have the following file structure:

```
Projects/
    Foobar/
        build/
            out.py
        src/
            foo/
                foo.py
                foobar.py
                baz/
                    baz.py
            bar/
                bar.py
        test/
            footest.py
            foobartest.py
```

Given an `"rc_search_limit"` of 3, placing a `.sublimelinterrc` file within the following directories would have the following effects:

- **foo** – This would apply to `foo.py`, `foobar.py` and `baz/baz.py`.
- **src** – This would apply to all of the files within `foo`, `foo/baz`, and `bar`.
- **Foobar** – This would apply to all files within `build`, `src/foo`, `src/bar`, and `test` directories, but **not** to files within `src/foo/baz`, because `Foobar` is more than 3 directories from `baz.py`. In this case you would have to increase `"rc_search_limit"` to at least 4.

.sublimelinterrc structure

The contents of a `.sublimelinterrc` file should be JSON settings in the same format as the `"user"` object in user settings. For example, here is a `.sublimelinterrc` that sets the *"@python" meta setting* for all linters and configures flake8 to ignore all warnings:

```
{
    "@python": 3,
    "linters": {
        "flake8": {
            "ignore": "W"
        }
    }
}
```

Inline settings

Sometimes you need to change the settings for a single file. Some linters may define one or more **inline settings**, which are settings that can specified directly in a file.

Note: Inline settings must appear within a comment on the first two lines of a file to recognized.

The format for inline settings is as follows:

```
<comment> [SublimeLinter <linter>-<setting>:<value> ...]
```

Let's break this down a bit:

`<comment>`

This represents the comment start characters for the linter's language. This may be followed by any number of characters before the actual inline settings.

`[SublimeLinter`

This marks the beginning of the inline settings. "SublimeLinter" is not case-sensitive, so "Sublimelinter" and "sublimelinter" are also valid.

`<linter>`

The lowercase name of the linter to which the setting belongs, followed by "-".

`<setting>:<value>`

The setting name and value. Any amount whitespace may be placed before or after the ":". The value may not have any whitespace, as whitespace is used to delimit multiple settings.

`...]`

Any number of `<linter>-<setting>:<value>` settings may included before the terminating "]".

Here is an example of an inline setting that sets two values for the flake8 linter:

```
# [SublimeLinter flake8-max-line-length:100 flake8-max-complexity:10]
```

Those inline settings are the equivalent of the following in a settings file:

```
{
    "linters": {
        "flake8": {
            "max-line-length": 100,
            "max-complexity": 10
        }
    }
}
```

But in the case of the inline settings, it applies only to the file in which they appear.

Note: Please see the documentation for each linter to find out what inline settings it supports.

shebangs

Each linter has the option to turn a file's `shebang` into an inline setting. For example, python-based linters turn this:

```
#!/usr/bin/env python3
```

into the inline setting `@python: 3`.

Note: Please see the documentation for each linter to find out if it supports a shebang inline setting.

Inline overrides

Often linters accept options with multiple values. For example, the flake8 python linter has a `select` and `ignore` option that takes one or more values. Let's assume you aren't interested in warnings about trailing whitespace, since you have configured Sublime Text 3 to trim trailing whitespace when saving. In addition, you would like the default maximum line length to be 100 characters, and you don't care about how many blank lines are before a method or class definition. So you have added the following to the flake8 settings in the user or project settings:

```
{
    "linters": {
        "flake8": {
            "@disable": false,
            "args": [],
            "excludes": [],
            "ignore": "E302,W291,W293",
            "max-complexity": -1,
            "max-line-length": 100,
            "select": ""
        }
    }
}
```

E302 will ignore PEP8 errors for the number of blank lines before a method or class definition. W291 and W293 will ignore trailing whitespace on a non-empty and empty line respectively.

This works great so far. But there is one file where you actually need to conform to PEP8 spacing rules for methods and classes, and you would like to ignore W601 warnings about `has_key` being deprecated. It would be nice if you could specify only the additions and subtractions to the ignore setting, without affecting the base setting you made lower in the settings stack.

Inline overrides provide this mechanism. Inline overrides are specified inline in exactly the same way as inline settings, but instead of replacing settings of the same name lower in the settings stack, they add or remove options within a setting.

So, for example, given the example above where we want to remove the E302 ignore, add a W601 ignore, and set the maximum line length to 120, you would do this:

```
# [SublimeLinter flake8-ignore:-E302,+W601 flake8-max-line-length:120]
```

A couple things to note:

- A prefix of − removes that option.
- A prefix of + adds that option.
- No prefix adds that option, so `-E302,+W601` and `-E302,W601` are equivalent.

- In the above example, `flake8-ignore` is an inline override, and `flake8-max-line-length` is an inline setting.
- Each linter defines what settings are inline settings and which are inline overrides.
- Each linter defines the separator you must use between multiple values in inline overrides.

In the example above, without the inline overrides, the ignore option passed to flake8 would be `E302,W291,W293`, which is taken from our base settings. With the inline overrides, the ignore option is `W201,W293,W601`.

Note: Please see the documentation for each linter to find out what inline overrides it supports.

Setting tokens

After the default, user and project settings are merged, SublimeLinter iterates over all settings values and replaces the following tokens with their current values:

Token	Value
${sublime}	The full path to the Sublime Text packages directory
${project}	The full path to the project's parent directory, if available.
${directory}	The full path to the parent directory of the current view's file.
${home}	The full path to the current user's home directory.
${env:x}	The environment variable 'x'.

Please note:

- Directory paths do **not** include a trailing directory separator.
- `${project}` and `${directory}` expansion are dependent on a file being open in a window, and thus may not work when running lint reports.
- The environment variables available to the `${env:x}` token are those available within the Sublime Text python context, which is a very limited subset of those available within a command line shell.

Project and parent directory paths are especially useful if you want to load specific configuration files for a linter. For example, you could use the `${project}` and `${home}` tokens in your project settings:

```
{
    "folders":
    [
        {
            "follow_symlinks": true,
            "path": "/Users/tinytim/Projects/Tulips"
        }
    ],
    "SublimeLinter":
    {
        "linters":
        {
            "phpcs": {
                "standard": "${project}/build/phpcs/MyPHPCS"
            },
            "phpmd": {
                "args": ["${home}/phpmd-ruleset.xml"]
            }
        }
    }
}
```

After token replacement, SublimeLinter sees the linter settings as:

```
{
    "linters":
    {
        "phpcs": {
            "standard": "/Users/tinytim/Projects/Tulips/build/phpcs/MyPHPCS"
        },
        "phpmd": {
            "args": ["/Users/tinytim/phpmd-ruleset.xml"]
        }
    }
}
```

Global Settings

SublimeLinter supports the following global settings:

debug

If `true`, SublimeLinter prints information to the Sublime Text 3 console that may help in debugging problems with a linter. For example, the command passed to a linter and the output from the linter are logged to the console in debug mode. All of SublimeLinter's log messages are prefixed by "SublimeLinter: ". The default value is `false`.

Rather than change this setting manually, you are better off using the user interface to *set the debug mode*.

delay

When the *lint mode* is `"background"`, this setting determines how long SublimeLinter will wait until acting on a lint request. The default value is 0.25 (seconds). For a discussion of this setting, see *Usage*.

error_color

This setting determines the color used to mark errors in the text. It should be a six-digit hex RGB color (like those used in CSS), with or without a leading "#". If this setting is changed, SublimeLinter will offer to update all user color schemes with the new error color. The default value is `"D02000"`.

Warnings are marked with *warning_color*.

gutter_theme

This setting should be the full `Packages`-relative path to the `.gutter-theme` file for the current gutter theme. Rather than changing this setting manually, you are better off using the user interface to *choose a gutter theme*. The default value is `Packages/SublimeLinter/gutter-themes/Default.gutter-theme`.

gutter_theme_excludes

If you wish to exclude one or more gutter themes from the list of available gutter themes, you can add one or more patterns to the array in this setting. The patterns are matched against the gutter theme name using python's fnmatch method. The default value is an empty array.

For example, if you wanted to exclude the "Blueberry" gutter themes that come with SublimeLinter, you would use this setting:

```
{
    "gutter_theme_excludes": [
        "Blueberry*"
    ]
}
```

lint_mode

This setting determines the current *lint mode*. Possible values are `"background"`, `"load/save"`, `"save only"`, and `"manual"`. The default value is `"background"`.

Rather than change this setting manually, you are better off using the user interface to *choose a lint mode*.

mark_style

This setting determines the current *mark style*. Possible values are `"fill"`, `"outline"`, `"solid underline"`, `"squiggly underline"`, `"stippled underline"`, and `"none"`. The default value is `"outline"`.

Rather than change this setting manually, you are better off using the user interface to *choose a mark style*.

no_column_highlights_line

This setting determines what happens when a linter reports an error with no column information. By default, a mark is put in the gutter but no text is highlighted. If this setting is `true`, in addition to the gutter mark, the entire line is highlighted.

Rather than change this setting manually, you are better off using the user interface to *set the no-column highlight mode*.

passive_warnings

This setting allows you the ability to hide warnings in the *"Show All Errors"* Quick Panel. See *Navigating Errors* for more information on this setting. The default value is `false`.

paths

This setting provides extra paths to be searched when *locating system executables*.

Note: Instead of using this setting, consider *setting up your PATH correctly* in your shell.

This setting works like the `PATH` environment variable; you provide **directories**, relative or absolute, that will be searched for executables (e.g. `"/opt/bin"`), **not** paths to specific executables.

If you give a relative path to a directory, it is considered relative to the Sublime Text 3 executable.

You may provide separate paths for each platform on which Sublime Text 3 runs. The default value is empty path lists.

```
{
    "paths": {
        "linux": [],
        "osx": [],
        "windows": []
    }
}
```

python_paths

When SublimeLinter starts up, it reads `sys.path` from the system python 3 (if it is available), and adds those paths to the SublimeLinter `sys.path`. So you should never need to do anything special to access a python module within a linter. However, if for some reason `sys.path` needs to be augmented, you may do so with this setting. Like the `"paths"` setting, you may provide separate paths for each platform on which Sublime Text 3 runs. The default value is empty path lists.

rc_search_limit

This setting determines how many directories will be searched when looking for a `.sublimelinterrc` settings file. The default value is 3. See *.sublimelinterrc settings* for more information.

shell_timeout

This setting determines the number of seconds that SublimeLinter will wait when executing a shell command, for example when getting the value of PATH. The default value is 10. If the SublimeLinter debug log says that shell commands are timing out, you may need to increase the value of this setting.

show_errors_on_save

This setting determines if a Quick Panel with all errors is displayed when a file is saved. The default value is `false`.

Rather than change this setting manually, you are better off using the user interface to *change this setting*.

show_marks_in_minimap

This setting determines whether error marks are shown in the minimap. The default value is `true`.

syntax_map

This setting allows you to map one syntax **name** to another. Because linters are tied to a syntax name, this is useful when there are variations on a syntax that should use the same linter.

Note: Syntax names are the name of the .tmLanguage file that defines the syntax. This is **not** the file extension of the files to be linted, and may not necessarily be what is in the `View > Syntax` menu and in the lower right of the status bar.

The syntax names in the map should be lowercase. The default value is:

```
{
    "python django": "python",
    "html 5": "html",
    "html (django)": "html",
    "html (rails)": "html",
    "php": "html"
}
```

This means that any file that has the named syntax which is one of the keys will be linted by any linter than supports the named syntax corresponding to that key. For example, any file with the "python django" syntax will be linted by any linter that supports the "python" syntax.

Let's say you install some fancy new syntax package for python named "Totally Awesome Python". To ensure SublimeLinter will lint files that use that syntax, you would modify the `"syntax_map"` setting as follows:

```
{
    "totally awesome python": "python",
    "python django": "python",
    "html 5": "html",
    "html (django)": "html",
    "html (rails)": "html",
    "php": "html"
}
```

warning_color

This setting determines the color used to mark warnings in the text. It should be a six-digit hex RGB color (like those used in CSS), with or without a leading "#". If this setting is changed, SublimeLinter will offer to update all user color schemes with the new warning color. The default value is `"DDB700"`.

Errors are marked with *error_color*.

wrap_find

This setting determines if the `Next Error` and `Previous Error` commands wrap around when reaching the end or beginning of the file. See *Navigating Errors* for more information on those commands. The default value is `true`.

Meta Settings

Meta settings are special global settings that can be used both at the global and linter level. When used globally, they are applied to every linter and override a linter meta setting. Meta setting names always begin with "@".

The following meta settings are supported:

@disable

This boolean setting disables linters, preventing them from running. If this setting is present globally, it forces all linters to be disabled if `true` or forces all linters to be enabled if `false`.

If you want to disable all linters, rather than change this setting manually, you are better off using the user interface to *disable all linters*.

Note: Setting `@disable` to `false` enables **all** linters, regardless of whether you have set `@disable` to `true` for an individual linter. If you want to disable linters individually, you must remove the global `@disable` setting. Setting the global `@disable` setting through the user interface does this for you.

@python

Because new versions of python are potentially backwards-incompatible with earlier versions, dealing with python-based linters can be tricky:

- The linter itsef may only run on a specific major.minor version of python (or later).
- It may be necessary to run it on a version of python compatible with the version in which the code to be linted was written.
- When a linter provides a python 3-compatible API, a linter plugin will usually want to use the API directly instead of calling an external binary.

The `@python` meta setting is a floating point number that specifies the python version that should be used when running python-based linters. This is especially useful when used in project settings or *.sublimelinterrc settings* to specify that the files in a particular project or directory should be treated as a particular version of python.

For example, let's say you are working on a project called "Widget" that is written in python 3, and you want to make sure it is treated as such by linters such as flake8. In the project settings, you would do this:

```
{
    "folders":
    [
        {
            "follow_symlinks": true,
            "path": "/Users/aparajita/Projects/Widget"
        }
    ],
    "SublimeLinter":
    {
        "@python": 3
    }
}
```

That's all there is to it. Of course, beneath the hood a lot of magic is happening.

Resolving python versions

What happens when SublimeLinter is asked to resolve a `@python` version depends on the linter plugin and the platform.

- If the linter plugin indicates that a specific version of python must be used, that version will always be used, regardless of your `@python` setting.

- If the linter plugin indicates that any version of python may be used, the default python is used, unless the linter plugin specifies that the linter executable is sensitive to python versions, in which case the version you specify with the `@python` setting will be used.

The following algorithm is used both when resolving a python version for a linter plugin and when resolving a python version you specify with the `@python` meta setting.

Mac OS X/Linux

On Posix systems, python is installed with binaries (or symlinks to binaries) for both the major.minor version and the major version. When a specific version is requested, the following happens:

- First the exact version is located. For example, if the requested version is `2.7`, SublimeLinter attempts to locate an executable named `python2.7`.
- If the exact version cannot be located, the major version of the requested version is located. For example, if the requested version is `2.7`, SublimeLinter attempts to locate an executable named `python2`.
- If the major version is not available, SublimeLinter attempts to locate `python`.

Windows

On Windows, python is usually installed in the root volume in a directory called `Python<major><minor>`, where `<major>` and `<minor>` are the major and minor python version, e.g. `C:\Python27`. When a specific version is requested, the following happens:

- Directories whose names begin with "Python" in the root volume are iterated. The remainder of the directory name is used as the version.
- If the exact requested version does not match any of the directory versions, SublimeLinter attempts to match the major requested version.
- If the major requested version does not match any of the directory versions, SublimeLinter attempts to `python`.

Version matching

Once an available version of python is located, its full version is matched against the requested version. An available version satisfies a version request if one of the following is true:

- The requested version has no minor version and the available major version matches.
- The requested major version matches the available major version and the requested minor version is <= the available minor version.

If the available version satisfies the requested version, its path (or the built-in python) is used. Otherwise the request fails and the linter will not run.

Linter Settings

Each linter plugin is responsible for providing its own settings. In addition to the linter-provided settings, SublimeLinter adds the following settings to every linter:

@disable

This is actually a meta setting that is added to every linter's settings. For a discussion of the `@disable` setting, see *Meta Settings*.

Rather than change this setting manually, you can use the user interface to *disable or enable a linter*.

args

This setting specifies extra arguments to pass to an external binary. This is useful when a linter binary supports an option that is not part of the linter's settings.

The value may be a string or an array. If it is a string, it will be parsed as if it were passed on a command line. For example, these values are equivalent:

```
{
    "args": "--foo=bar --bar=7 --no-baz"
}

{
    "args": [
        "--foo=bar",
        "--bar=7",
        "--no-baz"
    ]
}
```

The default value is an empty array.

Note: If a linter runs python code directly, without calling an external binary, it is up to the linter to decide what to do with this setting.

---

chdir

This setting specifies the linter working directory.

The value must be a string, corresponding to a valid directory path.

```
{
    "chdir": "${project}",
}
```

With the above example, the linter will get invoked from the `${project}` directory (see *Setting Tokens* for more info on using tokens).

Note: If the value of `chdir` is unspecified (or inaccessible), then:

- If linting an unsaved file, the directory is unchanged
- If linting a saved file, the directory is set to that of the linted file

---

excludes

This setting specifies a list of path patterns to exclude from linting. If there is only a single pattern, the value may be a string. Otherwise it must be an array of patterns.

Patterns are matched against a file's **absolute path** with all symlinks/shortcuts resolved, using python's fnmatch method. This means to match a filename, you must match everything in the path before the filename. For example, to exclude any python files whose name begins with "foo", you would use this pattern:

```
{
    "excludes": "*/foo*.py"
}
```

The default value is an empty array.

ignore_match

This setting specifies a python regular expression that is matched against error messages reported by the linter. If the regular expression matches a message, the error is ignored.

The value of this setting may be:

- A single regular expression pattern string.
- An array of pattern strings.
- A map, where the keys are **lowercase** filename extensions to match (with or without a leading dot), and the values are either single pattern strings or arrays of pattern strings.

Note: The pattern strings are regular JSON strings, not raw strings as you would usually use in python. If you need to escape regular expression pattern characters, be sure to use double backslashes (`\\`). For example, to match `Undeclared (variable)`, you would have to use the string `"Undeclared \\(variable\\)"`.

For example, the `html-tidy` linter complains if you are editing a portion of a page, as is often the case with `php`. The errors are:

```
missing <!DOCTYPE> declaration
inserting implicit <body>
inserting missing 'title' element
```

Obviously we don't want to see those errors, because we know they don't apply in this case. By using the `ignore_match` setting, we can ignore them like this:

```
{
    "ignore_match": [
        "missing <!DOCTYPE> declaration",
        "inserting implicit <body>",
        "inserting missing 'title' element"
    ]
}
```

Of course, since these are regular expressions, we could also do it like this:

```
{
    "ignore_match": [
        "missing <!DOCTYPE> declaration",
        "inserting (?:implicit <body>|missing 'title' element)"
    ]
}
```

Now let's suppose you only want the `ignore_match` to apply to `.inc` files, which we use for partials. We can do that by using a map like this:

```
{
    "ignore_match": {
        "inc": [
            "missing <!DOCTYPE> declaration",
            "inserting (?:implicit <body>|missing 'title' element)"
        ]
}
```

In *debug mode*, SublimeLinter logs each occurrence of an ignore match.

Note: If you need help constructing, testing or debugging a regular expression, regular expressions 101 provides an easy way to do so.

Troubleshooting

SublimeLinter is a complex beast, and because it tries to tie together Sublime Text 3, a linter plugin, and a linter binary, there are plenty of places for things to go wrong.

Console output

If SublimeLinter detects an error during its operation, it prints a (hopefully) useful message to the Sublime Text 3 console. To open the console, press `Control+`. Error messages will be prefixed with `SublimeLinter: WARNING:` or `SublimeLinter: ERROR:`.

This will cover most errors, but if the linter executable itself generates an internal error, SublimeLinter cannot detect that. To see error messages generated by an executable, you will have to turn debug mode on.

Debug mode

If you are having trouble installing or configuring SublimeLinter, or it doesn't seem to be working, and none of the error messages in the console are indicating why, you can turn on **debug mode**. In debug mode, SublimeLinter prints copious amounts of information about what is happening internally to the Sublime Text 3 console.

To turn on debug mode, follow these steps:

1. Click on `Tools > SublimeLinter` in the main menu bar.
2. At the bottom of the `SublimeLinter` menu, you will see a `Debug Mode` item. If it is unchecked, select the item.
3. Restart Sublime Text 3.
4. Press `Control+`` to open the Sublime Text 3 console.
5. Look for messages that begin with `SublimeLinter:`, especially ones that begin with `SublimeLinter: WARNING:` or `SublimeLinter: ERROR:`.
6. Go to the SublimeLinter issue tracker and report the problem along with any `SublimeLinter:` console output.

Among the information dumped to the console in debug mode:

- The `PATH` detected from your system plus whatever you put in the *"paths" global setting*, used to locate executables.

- If python 3 is on your system, `sys.path` for that python.
- Success or failure of module importing by python-based linter plugins.
- The command line used to execute a linter.
- The output from the linter, including both linting reports and internal linter errors.

The linter doesn't work!

The first thing you should do when a linter does not work within SublimeLinter is to test it from the command line (Terminal in Mac OS X/Linux, Command Prompt in Windows). If it does not work from the command line, it definitely won't work in SublimeLinter.

Here are the most common reasons why a linter does not work:

- The linter binary is not installed. In that case the linter plugin will print a warning to the Sublime Text 3 console like this:

  ```
  SublimeLinter: WARNING: jshint deactivated, cannot locate 'jshint'
  ```

  Be sure to install the linter as documented in the linter plugin's README.

- The linter binary is installed (as verified from the command line), but its path is not available to SublimeLinter. In that case you will see a warning message like the one above for a missing linter binary. Follow the steps in *Debugging PATH problems* below to ensure the binary's path is available to SublimeLinter.

- The linter binary is installed, but it does not fulfill the plugin's version requirements. In that case SublimeLinter will print a warning to the Sublime Text 3 console like this:

  ```
  SublimeLinter: WARNING: jshint deactivated, version requirement (>= 2.5.0) not␣
  ↪fulfilled by 2.4.3
  ```

  In most cases this means you have an old version of the linter binary installed. Upgrading to the latest version should fix the problem.

- A python-based linter binary is installed, but it does not work with python 2 or python 3 code. In that case, follow the steps in *Debugging python-based linters* below.

Use the group, Luke

Please do **not** open a ticket on Github issues until you have verified there isn't already a ticket in the SublimeLinter issue tracker or the now deprecated SublimeLinter google group.

Once you are confident the problem is solved, you can turn debug mode off using the steps above to uncheck the `Debug Mode` menu item.

Debugging PATH problems

In order for SublimeLinter to use linter executables, it must be able to find them on your system. There are two possible sources for executable path information:

1. The `PATH` environment variable.
2. The *"paths"* global setting.

At startup SublimeLinter queries the system to get your `PATH` and merges that with paths in the *"paths"* setting. In *debug mode* SublimeLinter prints the computed path to the console under the heading `SublimeLinter: computed PATH <source>:`. You can use that information to help you determine why a linter executable cannot be found.

If a linter's executable cannot be found when the linter plugin is loaded, the plugin is disabled and you will see a message like this in the console:

```
SublimeLinter: WARNING: jshint deactivated, cannot locate 'jshint'
```

On the other hand, if the linter plugin's executable can be found at load time, but later on it becomes unavailable, when you try to use that linter you will see a message like this in the console:

```
SublimeLinter: ERROR: could not launch ['/usr/local/bin/jshint', '--verbose', '-']
SublimeLinter: reason: [Errno 2] No such file or directory: '/usr/local/bin/jshint'
SublimeLinter: PATH: <your PATH here>
```

Another possibility is that an executable called by the linter executable is missing. For example, if jshint is available but node is not, you would see something like this in the console:

```
SublimeLinter: jshint output:
env: node: No such file or directory
```

Note: On Windows, linter errors messages will not always appear. It appears to be a bug in python.

Unlike the other error messages mentioned earlier, you would not see this message unless debug mode was turned on, because it isn't an error message detected by SublimeLinter, it is the output captured from the jshint executable. So if you aren't seeing any errors or warnings in the console and the linter isn't working, turn on debug mode to see if you can find the source of the problem.

Finding a linter executable

If SublimeLinter says it cannot find a linter executable, there are several steps you should take to locate the source of the problem.

First you need to see if the linter executable is in your `PATH`. Enter the following at a command prompt, replacing `linter` with the linter executable name:

```
# Mac OS X, Linux
hash -r
which linter

# Windows
where linter
```

If the result says that the linter could not be found, that means the linter executable is in a directory which is not in your `PATH`, and SublimeLinter will not be able to find it. At this point you will have to find out what directory the executable was installed in from the linter's documentation. Once you find that, you will need to augment your `PATH` by following the steps in *Augmenting PATH* below.

If the result of `which` displays a path, this means the executable is in your `PATH`, but you are on Mac OS X or Linux and the path to the executable is exported in a shell startup file that SublimeLinter does not read. This means you must add the parent directory of the executable to your `PATH` by following the steps in *Augmenting PATH* below.

Augmenting PATH

If the path to an executable's parent directory is not available to SublimeLinter, you have two choices:

1. Add the path to the *"paths"* global setting.
2. On Mac OS X or Linux, adjust your shell startup files. On Windows, add the directory to your `PATH` environment variable.

Note: Paths in the *"paths"* setting will be searched before system paths.

Adding to the "paths" setting

This approach is the quickest and usually the easiest, but means that you will have to maintain paths both in your system and in SublimeLinter. In addition, it isn't always obvious what path to add without consulting the documentation for software you install.

Once you determine a path that needs to be added, *open your user settings* and add the path to the `"paths"` array for your platform. For example, let's say you are using rbenv on Mac OS X, which adds the path `~/.rbenv/shims` to your `PATH`. You would change the `"paths"` setting like this:

```
{
    "paths": {
        "linux": [],
        "osx": [
            "~/.rbenv/shims"
        ],
        "windows": []
    }
}
```

Adjusting shell startup files

This approach is a little more complicated, but once you get it right then your `PATH` will be correct for command line usage and for SublimeLinter.

All shells read various files when they are run. Depending on the command line arguments, shells read a "profile/env" file of some sort and an "rc" (runtime configuration) file. For example, `bash` reads `.bash_profile` and `.bashrc` (among others) and `zsh` reads `.zshenv` and/or `.zprofile` (depending on the platform) and `.zshrc` (among others).

If you aren't sure what shell you are using, type this in a terminal:

```
echo $SHELL
```

When SublimeLinter starts up, it runs your shell as a **login shell** to get the `PATH`. This forces the shell to read the "profile/env" file, but for most shells the "rc" file is not read. There is a very good reason for this: performing initialization that only relates to interactive shells is not only wasteful, it will in many cases fail if there is no terminal attached to the process. By the same token, you should avoid putting code in the "profile/env" file that has any output (such as `motd` or `fortune`), since that only works with interactive shells attached to a terminal.

The list of shells supported by SublimeLinter and the startup file that must contain `PATH` augmentations is shown in this table:

Shell	File
bash	~/.bash_profile (or ~/.profile on Ubuntu)
zsh (Mac OS X)	~/.zprofile
zsh (Linux)	~/.zshenv or ~/.zprofile
fish	~/.config/fish/config.fish

If you are using `zsh` on Linux, you need to determine which file is used in your flavor of Linux. To do so, follow these steps:

1. Open `.zshenv` in an editor and insert `echo env` on the first line. If the file does not exist, create it.
2. Do the same for `.zprofile`, but insert `echo profile`.
3. In a terminal, enter `$SHELL -l -c 'echo hello'`. If you see both "env" and "profile", use `.zshenv` for `PATH` augmentations. If you see only one of the two, use that file for `PATH` augmentations.
4. Remove or comment out the `echo` lines you added.

When you installed a linter executable, it may have augmented your `PATH` in the "rc" file. But for these path augmentations to be visible to SublimeLinter, you must move such augmentations to the "profile/env" file. For example, if you are using `bash` as your shell and you installed rbenv, you would probably find this in your `.bashrc` file:

```
eval "$(rbenv init -)"
```

For SublimeLinter to "see" this, however, you have to move that line from `.bashrc` to the file that SublimeLinter will see, which is `.bash_profile` from the table above.

If `which` or `where` cannot find a linter executable from the command line, you need to add the executable's parent directory to your `PATH`. Assuming a directory of `/opt/bin`, on Mac OS X or Linux the changes you would make are summarized in the following table:

Shell	File	Code
bash	~/.bash_profile	export PATH=/opt/bin:$PATH
zsh (Mac OS X)	~/.zprofile	export PATH=/opt/bin:$PATH
zsh (Linux)	~/.zshenv or ~/.zprofile	export PATH=/opt/bin:$PATH
fish	~/.config/fish/config.fish	set PATH /opt/bin $PATH

Special considerations for `bash`

If you are using `bash` as your shell, there is one more step you must take after augmenting your `PATH` in `.bash_profile`.

- On Mac OS X, add this code to the **bottom** of `.bash_profile`:

  ```
  case $- in
     *i*) source ~/.bashrc
  esac
  ```

  On Mac OS X, `bash` does **not** load `.bashrc` unless explicitly run with the `-i` command line argument. On the other hand, `.bash_profile` is loaded in each new interactive Terminal session and if `bash` is run as a login shell. So you must load `.bashrc` in `.bash_profile`, but should only do so if the shell is interactive, which is what the code above does.

- On Linux, add this code to the **top** of `.bashrc`:

  ```
  source ~/.bash_profile
  ```

On Linux, by default `bash` does **not** load `.bash_profile` for an interactive session, but it does for a login shell. So if you move your `PATH` augmentations to `.bash_profile` and source that in `.bashrc`, your `PATH` augmentations will always be loaded.

Editing `PATH` on Windows

On Windows you need to edit your `PATH` environment variable directly. The easiest way to do this is with the Path Editor, a free application. Once you install and launch Path Editor, follow these steps:

1. Click the Add button.
2. Select the parent directory of the linter executable and click OK.
3. Click OK at the bottom of the Path Editor window.

On any platform, after you have changed your `PATH`, you will need to restart Sublime Text 3.

Validating your PATH

To verify that SublimeLinter will be able to see the changes you made above, enter the following at a command prompt, replacing "linter" with the name of the linter executable which could not be found:

```
# Mac OS X, Linux
> $SHELL -l -c '/usr/bin/which linter'

# Windows
> where linter
```

If your changes were correct, it will print the path to the linter executable. If the executable path is not printed, then do the following to see what `PATH` SublimeLinter will see:

```
# bash, zsh
> $SHELL -l -c 'echo $PATH | tr : "\n"'

# fish
> fish -l -c 'for p in $PATH; echo $p; end'

# Windows
> path
```

At this point you should double-check that you followed the instructions above correctly, and if you still cannot figure out what is going wrong, post a message on the SublimeLinter issue tracker. Be sure to outline the steps you took and include the contents of your shell startup file.

Debugging python-based linters

When using python-based linters, there are more possibilities for configuration problems:

- The version of python or the python script specified in the linter plugin may not be available.
- The version of python you specify for your source code with the *@python meta setting* or a *shebang* may not be available.
- The specified version of python may be available, but the linter module for that version may not be installed.

To understand how these might occur, it's important to understand *how SublimeLinter resolves python versions*. Let's look at the console output for each case to see how to spot these problems.

Linter's python is not available

When a python-based linter plugin is loaded that does not support direct execution (the `module` attribute is `None`), if the `cmd` attribute specifies `script@python<version>`, where `script` is a python script such as `flake8`, and `<version>` is a major[.minor] version, SublimeLinter attempts to *locate a version of python* that satisfies `<version>`.

If no version of python can be found that satisfies the requested version, the linter plugin is disabled, and you will see the following message in the console (where "foo" is the linter name):

```
SublimeLinter: WARNING: foo deactivated, no available version of python or foo␣
↪satisfies foo@python2
```

Setting python is not available

If the linter plugin does **not** specify a python version in the `cmd` attribute (e.g. `flake8@python`), then SublimeLinter will tentatively enable the linter when it is loaded, even if no default python can be found, because the requested python version may change based on your settings.

For example, if there were no default version of python available for flake8, you would see this in the console at startup:

```
SublimeLinter: flake8 activated: (None, None)
```

Now if you tried to use the flake8 linter with code that did not have a specific python version set with the *@python meta setting*, *inline setting* or *shebang*, you would see this error in the console:

```
SublimeLinter: ERROR: flake8 cannot locate 'flake8@python'
```

Module not installed

On the other hand, if `python2` is available and you have a `@python: 2` meta or inline setting, **but** you do not have flake8 installed for python 2, you would see something like this in the console:

```
SublimeLinter: flake8: test.py ['/usr/bin/python2', '/usr/local/bin/flake8', '--max-
↪complexity=-1', '-']
SublimeLinter: flake8 output:
Traceback (most recent call last):
  File "/usr/local/bin/flake8", line 5, in <module>
    from pkg_resources import load_entry_point
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python/
↪pkg_resources.py", line 2556, in <module>
    working_set.require(__requires__)
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python/
↪pkg_resources.py", line 620, in require
    needed = self.resolve(parse_requirements(requirements))
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/Extras/lib/python/
↪pkg_resources.py", line 518, in resolve
    raise DistributionNotFound(req)  # XXX put more info here
pkg_resources.DistributionNotFound: flake8==2.1.0
```

Some good advice

To ensure your python linters work well, always ensure:

- The versions of python you code in are available in your `PATH`.
- You install the linter module (using easy_install or pip) for all versions of python you plan to use it with.

If you do that, you shouldn't have any problems. But if you do, hopefully the troubleshooting guide above will help you understand what is wrong with your system configuration.

CHAPTER 4

Developer Documentation

Creating a Linter Plugin

SublimeLinter makes it easy to create a linter plugin in a few steps:

1. Using the `Create Linter Plugin` command, create a template plugin complete with user documentation.
2. Change a few *attributes* in the linter class.
3. Update the user documentation.
4. Submit it for review.

On this page we'll cover Step 1.

Creating a template plugin

Before creating your plugin, it's important to understand the naming convention.

Note: Linter plugins must be named after the linter **binary** they use, **not** the language they lint (unless the language binary itself is used to lint). For example, to lint `python` with `pylint`, the linter name you will enter must be `pylint`, **not** `python`.

Got it? Okay, here we go:

1. Bring up the Command Palette and type `plugin`. Among the commands you should see `SublimeLinter: Create Linter Plugin`. If that command is not highlighted, use the keyboard or mouse to select it.
2. A dialog will appear explaining the naming convention for linter plugins. If you understand, click "I understand". If you click "I understand" and still name the linter plugin after the language instead of the linter binary, you will have to pay me $1,000,000.
3. An input field will appear at the bottom of the window. Enter the name of the linter binary — **not** the language — that the plugin will interface with and press `Return`.

4. You will be asked what language the linter is based on. The linter **plugin** is always python-based, the question here is what language the linter executable (such as jshint) itself is based on. If you select a language, SublimeLinter will fill out the template plugin, copy it to the Sublime Text 3 `Packages` directory with the name `SublimeLinter-contrib-<linter>`, initialize it as a git repository if `git` is available, and then open it in a new window.

   Note: Do **not** rename the plugin directory unless absolutely necessary. The directory name **must** come after "SublimeLinter" alphabetically to ensure SublimeLinter loads before the linter plugins. Also, user-created linter plugins use the "-contrib" prefix to distinguish them from "official" plugins that have been vetted and moved into the SublimeLinter org on github.

5. The plugin directory will be opened in Sublime Text 3. You can then start modifying the linter plugin (`linter.py`) according to your needs.

Coding guidelines

For the benefit of all users, I try to maintain a consistently high standard in all third party SublimeLinter plugins. This is enforced by maintaining control over the channel Package Control uses for all SublimeLinter-related repos. If you would like your linter plugin to be published to Package Control, you must follow these guidelines:

- Indent is 4 spaces.
- Install the flake8 and pep257 linters to check your code and fix all errors.
- Vertical whitespace helps readability, don't be afraid to use it.
- Please use descriptive variable names, no abbrevations unless they are very well known.

Updating class attributes

Template linter plugins are created with almost all of the Linter class attributes filled in with the default values. To make your new linter plugin functional, at the very least you need to do the following:

- Change the *syntax* attribute to indicate the syntax (or syntaxes) that the linter lints.
- Change the *cmd* attribute to include the executable and arguments you want to include on every run. Or if you are going to implement a *cmd* method, set the attribute to `None` and set the *executable* attribute to the name of the linter executable.
- Change the *regex* attribute to correctly capture the error output from the linter.
- Change the *multiline* attribute to `True` if the regex parses multiline error messages.
- Determine the minimum/maximum versions of the linter executable that will work with your plugin and change the *version_args*, *version_re* and *version_requirement* attributes accordingly.
- If the linter executable does not accept input via `stdin`, set the *tempfile_suffix* attribute to the filename suffix of the temp files that will be created.

These are the minimum requirements to make a linter plugin functional. However, depending on the features of the linter executable, you may need to configure other class attributes.

- If the linter outputs errors only on `stderr` or `stdout`, set *error_stream* to `util.STREAM_STDERR` or `util.STREAM_STDOUT` respectively.
- If you wish to support *inline settings* and/or *inline overrides*, add them to the *inline_settings* and *inline_overrides* attributes and be sure to set the *comment_re* attribute, unless you are subclassing from *PythonLinter* or *RubyLinter*, which do that for you.

- If you wish to support embedded syntaxes, set the *selectors* attribute accordingly.
- If the linter subclasses from *PythonLinter*, remove the *module* attribute if you do not plan to use the linter's python API. If you do, you will need to implement the *check* method.

You should remove attributes that you do not change, as their values will be provided by the superclass.

Note: Please read the *linter attributes* documentation to learn more about these attributes before changing them blindly! You should also look at the existing collection of linter plugins in the SublimeLinter organization for reference.

Updating documentation

SublimeLinter creates a fairly complete set of template documentation for you, but you will need to fill in a few things.

1. Open `README.md` and do the following:
   - Change `__linter_homepage__` to the URL where users can find info about the linter.
   - Change `__syntax__` to the syntax name or names that the plugin will lint. Syntax names are the **internal** syntax names used by Sublime Text 3. See *Syntax names* for more information.
   - If necessary, complete the linter installation instructions. Try to be as complete as possible, listing all necessary prerequisites (with links) and instructions for all platforms if they differ.
   - If your linter plugin does not define the *defaults* attribute, remove the two paragraphs beginning with "In addition to the standard SublimeLinter settings". If your linter plugin does define the *defaults* attribute, document their values.
   - If any of the values in the *defaults* attribute are also used in *inline_settings* or *inline_overrides*, add a checkmark to the appropriate column in the template linter settings table. A checkmark is the html entity `&#10003;`. If you are not using *inline_settings* or *inline_overrides*, remove those columns in the linter settings table.
2. Open `messages/install.txt` and change the repo URL to be the correct URL for your plugin's repository.
3. Follow your own instructions! Try following the installation instructions in the README — if possible on Mac OS X, Linux and Windows — to see if you missed any important information or possibilities for confusion.

Preparing for publication

When you have followed all of the steps above and you think your plugin is ready for release, post a message on the SublimeLinter issue tracker with a link to your repo and it will be reviewed for correctness and completeness.

> **Warning:** Do **NOT** make a pull request on wbond/package_control_channel. All SublimeLinter plugins must go through SublimeLinter/package_control_channel.

Once your plugin has been reviewed and all issues have been fixed, you need to tag the final commit with a version number before publishing to Package Control:

```
git tag 1.0.0
git push origin 1.0.0
```

After the plugin is published to Package Control, every time you make a change, you must increment the version and tag the commit you want to publish. If it is a bug fix, increment the last number in the version. If you add functionality, increment the middle number. Then do the steps above with the new version.

Linter Attributes

All linter plugins must be subclasses (direct or indirect) of `SublimeLinter.lint.Linter`. The Linter class provides the attributes and methods necessary to make linters work within the SublimeLinter framework.

The Linter class is designed to allow interfacing with most linter executables/libraries through the configuration of class attributes, with no coding necessary. Some linters, however, will need to do more work to set up the environment for the linter executable, or may do the linting directly in the linter plugin itself. In that case, you will need to read the *linter method documentation*.

cmd

Mandatory. A string, list, tuple or *callable* that returns a string, list or tuple, containing the command line (with arguments) used to lint. If a string, it should be as if it were entered on a command line, and is parsed by shlex.split.

If `cmd` is `None`, it is assumed the plugin overrides the *run* method.

A `@` argument will be replaced with the filename, which allows you to guarantee that certain arguments will be passed after the filename. When *tempfile_suffix* is set, the filename will be the temp filename.

A `*` argument will be replaced with the arguments built from the linter settings, which allows you to guarantee that certain arguments will be passed at the end of the argument list.

Note: If the linter executable is python-based, there is a special form you should use for the `cmd` attribute. See *Python-based linters* for more information.

Examples

Here is the `cmd` attribute used for the csslint linter plugin:

```
cmd = 'csslint --format=compact'
```

For the jshint linter plugin:

```
cmd = 'jshint --verbose -'
```

For the flake8 linter plugin:

```
cmd = ('flake8@python', '*', '-')
```

Note how `'*'` is used as a placeholder for arguments built from settings, to ensure that `'-'` is passed as the last argument, which tells flake8 to use `stdin`.

comment_re

If the *inline_settings* or *inline_overrides* attribute is set, this attribute must be set to a regex pattern that matches the beginning of a comment. If you are subclassing from *PythonLinter* or *RubyLinter*, this attribute is set for you.

For example, to specify a match for JavaScript comments, you would use the pattern `r'\s*/[/*]'`.

config_file

Many linters look for a config file in the linted file's directory and in all parent directories up to the root directory. However, some of them will not do this if receiving input from `stdin`, and others use temp files, so looking in the temp file directory doesn't work.

If this attribute is set to a tuple of a config file argument and the name of the config file, the linter will automatically try to find the config file, and if it is found, add the config file argument to the executed command.

For example, if `config_file` is set to:

```
config_file = ('--config', '.jshintrc')
```

when SublimeLinter builds the argument list for the command line, if the file being linted has been saved to disk, SublimeLinter will look in the file's directory for `.jshintrc` and in all parent directories up to the root. If `.jshintrc` is found, SublimeLinter adds:

```
--config /path/to/.jshintrc
```

to the command line that runs the linter executable. Note that this facility works correctly when `'*'` is used as an argument placeholder in *cmd*.

You may also pass an arbitrary number of auxiliary directories to search after the second element, and ~ is expanded in those paths. If the hierarchy search fails, the auxiliary directories are checked in the order they are declared.

Note: When checking auxiliary directories, the hierarchy is **not** traversed. Only those directories are checked for the given filename.

Going back to the `.jshintrc` example above, to search in the file hierarchy and then in the user's home directory for a `.jshintrc` file, we would use this:

```
config_file = ('--config', '.jshintrc', '~')
```

The default value for `config_file` is `None`.

default_type

As noted in the *regex* documentation, you use the `error` and `warning` named capture groups to classify linter errors. If the linter output does not provide information which can be captured with those groups, this attribute is used to determine how to classify the linter error. The value should be `highlight.ERROR` or `highlight.WARNING`. The default value is `highlight.ERROR`.

defaults

If you want to provide default settings for the linter, set this attribute to a dict of setting names and values.

If a setting will be passed as an argument to the linter executable, you may specify the format of the argument here and the setting will automatically be passed as an argument to the executable. The format specification is as follows:

```
<prefix><name><joiner>[<sep>[+]]
```

- **prefix** – Either '@', '-' or '–'.

- **name** – The name of the setting.
- **joiner** – Either '=' or ':'. If `prefix` is '@', this attribute is ignored (but may not be omitted). Otherwise, if this is '=', the setting value is joined with `name` by '=' and passed as a single argument. If ':', `name` and the value are passed as separate arguments.
- **sep** – If the argument accepts a list of values, `sep` specifies the character used to delimit the list (usually ',').
- **+** – If the setting can be a list of values, but each value must be passed as a separate argument, terminate the setting with '+'.

After the format is parsed, the prefix and suffix are removed and the setting key is replaced with `name`.

Note: When building the list of arguments to pass to the linter, if the setting value evaluates to `False` (`None`, zero, `False`, or an empty sequence), the argument is not passed to the linter.

Examples

The flake8 linter accepts many command line options, but the ones we want to configure and pass to the linter are `--select`, `--ignore`, `--max-line-length`, and `--max-complexity`. So we define the `defaults` attribute as follows:

```
defaults = {
    '--select=,': '',
    '--ignore=,': '',
    '--max-line-length=': None,
    '--max-complexity=': -1
}
```

Here's how the `--select=,` default is parsed by SublimeLinter:

```
prefix: --
name: select
joiner: =
sep: ,
```

So `--select=,` accepts a list of values separated by ',', the default value list is `""`, and when passed to the linter it will be passed as one argument, with the values joined to `--select` by '='. After parsing, `--select=,` is changed to `select`.

The user sees these settings in their user settings:

```
{
    "flake8": {
        "@disable": false,
        "args": [],
        "excludes": [],
        "ignore": "",
        "max-complexity": -1,
        "max-line-length": null,
        "select": ""
    }
}
```

Given those values, the arguments passed to `flake8` are:

```
--max-complexity=-1
```

If the user changes the settings to this:

```
{
    "flake8": {
        "@disable": false,
        "args": [],
        "excludes": [],
        "ignore": "W291,W293",
        "max-complexity": -1,
        "max-line-length": 120,
        "select": ""
    }
}
```

the arguments passed to `flake8` are:

```
--ignore=W291,W293 --max-complexity=-1 --max-line-length=120
```

Here are the `defaults` for the gjslint linter plugin:

```
defaults = {
    '--jslint_error:,+': '',
    '--disable:,': '',
    '--max_line_length:': None
}
```

Here's how the `--jslint_error:,+` default is parsed by SublimeLinter:

```
prefix: --
name: jslint_error
joiner: :
sep: ,
+: present
```

So `--jslint_error:,+` accepts a list of values separated by ',', the default value list is `""`, and when passed to the linter each value will be passed as a separate argument. After parsing, `--jslint_error:,+` is changed to `jslint_error`.

The user sees these settings in their user settings:

```
{
    "gjslint": {
        "@disable": false,
        "args": [],
        "disable": "",
        "excludes": [],
        "jslint_error": "",
        "max_line_length": null
    }
}
```

Given those values, no arguments are passed to gjslint.

If the user changes the settings to this:

```
{
    "gjslint": {
        "@disable": false,
        "args": [],
        "disable": "0131,02",
        "excludes": [],
        "jslint_error": "indentation,unused_private_members",
        "max_line_length": 120
    }
}
```

the arguments passed to `gjslint` are:

```
--disable 0131,02 --jslint_error indentation --jslint_error unused_private_members --
↪max_line_length 120
```

Here is an example of using the '@' prefix. The phpmd linter does not use named arguments, and it takes as the last argument a comma-delimited list of rulesets to use. We want these rulesets to be a setting and an inline override. To accomplish this, we define the linter attributes like this:

```
cmd = ('phpmd', '@', 'text')
defaults = {
    '@rulesets:,': 'cleancode,codesize,controversial,design,naming,unusedcode'
}
inline_overrides = 'rulesets'
comment_re = r'\s*<!--'
```

By default, the following arguments are passed to `phpmd`:

```
/path/to/temp/file text cleancode,codesize,controversial,design,naming,unusedcode
```

The user can turn off individual rulesets inline, like this:

```
<!-- [SublimeLinter phpmd-rulesets:-controversial,-codesize] -->
```

which results in these arguments being passed to `phpmd`:

```
/path/to/temp/file text cleancode,design,naming,unusedcode
```

error_stream

Some linters report errors on `stdout`, some on `stderr`. For efficiency reasons there is no point in parsing non-error output, so by default SublimeLinter ignores `stderr` since most linters report errors on `stdout`.

However, it's very important that you capture errors generated by the linter itself, for example a bad command line argument or some internal error. Usually linters will report their own errors on `stderr`. To ensure you capture both regular linter output and internal linter errors, you need to determine on which stream the linter writes reports and errors.

For example, with jshint we would do the following, using a file `test.js` that we know has errors:

```
>> jshint test.js
test.js: line 4, col 13, Unnecessary semicolon.

1 error
```

So far so good. Now we turn off `stdout` to see where this is coming from.

```
>> jshint test.js > /dev/null
```

Note: On Windows, replace `/dev/null` with `nul`.

So now we see that linter report is on `stdout`. Now let's force an error and see what happens.

```
>> jshint foo.js > /dev/null
ERROR: Can't open foo.js
```

Since `stdout` is off, that means internal jshint errors are on `stderr`. So to capture both reports and errors we need to set `error_stream` to `util.STREAM_BOTH`.

Let's look at another example:

```
>> csslint test.css
csslint: There are 2 problems in /Users/aparajita/test.css.

test.css
1: warning at line 1, col 1
Don't use IDs in selectors.
#foo {

test.css
2: warning at line 2, col 10
Values of 0 shouldn't have units specified.
  width: 0px;

>> csslint test.css > /dev/null

# It was empty, meaning it uses stdout for reporting
>> csslint foo.css > /dev/null

# Still empty, errors might be on stdout
>>> csslint foo.css
csslint: Could not read file data in /Users/aparajita/foo.css. Is the file empty?

# Errors are also on stdout
```

So in this case, we set `error_stream` to `util.STREAM_STDOUT`. In this case, since there is no output on `stderr`, we could also leave it at the default of `util.STREAM_BOTH`, but for other linters there is unwanted output on a stream, so it's better to get in the habit of setting this to the exact value necessary.

With your linter, you will need to go through this process and set `error_stream` accordingly. Of course, you can be lazy and just set it to `util.STREAM_BOTH`, but I recommend against it, because it might not always work the way you expect.

Here are the possibilities:

Output	Errors	error_stream
stdout	stderr	util.STREAM_BOTH
stdout	stdout	util.STREAM_STDOUT
stderr	stdout	util.STREAM_BOTH
stderr	stderr	util.STREAM_STDERR

executable

If the name of the executable cannot be determined by the first element of `cmd` (for example when `cmd` is a method that dynamically generates the command line arguments), this can be set to the name of the executable used to do linting. Once the executable's name is determined, its existence is checked in the user's path. If it is not available, the linter is deactivated.

Note: If the `cmd` attribute is a string, list or tuple whose first element is the linter executable name, you do **not** need to define this attribute.

---

inline_overrides

This attribute is exactly like *inline_settings*, but defines a tuple/list of settings that can be used as *inline overrides*.

inline_settings

This attribute defines a tuple/list of settings that can be specified *inline*. If an inline setting is used as an argument to the linter executable, be sure to define the setting as an argument in *defaults*. If this attribute is defined, you must define *comment_re* as well, unless you are subclassing from *PythonLinter* or *RubyLinter*, which does that for you.

Within a file, the actual inline setting name is `<linter>-setting`, where `<linter>` is the lowercase name of the linter class. For example, the `Flake8` linter class defines the following:

```
inline_settings = ('max-line-length', 'max-complexity')
```

This means that `flake8-max-line-length` and `flake8-max-complexity` are recognized as inline settings.

line_col_base

This attribute is a tuple that defines the number base used by linters in reporting line and column numbers. Linters usually report errors with a line number, and some report a column number as well. In general, most linters use one-based line numbers and column numbers, so the default value is `(1, 1)`. If a linter uses zero-based line numbers or column numbers, the linter class should define this attribute accordingly.

For example, if the linter reports one-based line numbers but zero-based column numbers, the value of this attribute should be `(1, 0)`.

multiline

This attribute determines whether the *regex* attribute parses multiple lines. The linter may output multiline error messages, but if *regex* only parses single lines, this attribute should be `False` (the default). It is important that you set this attribute correctly; it does more than just add the `re.MULTILINE` flag when it compiles the *regex* pattern.

If `multiline` is `False`, the linter output is split into lines (using str.splitlines and each line is matched against *regex* pattern.

If `multiline` is `True`, the linter output is iterated over using re.finditer until no more matches are found.

re_flags

If you wish to add custom re flags that are used when compiling the `regex` pattern, you may specify them here.

For example, if you want the pattern to be case-insensitive, you could do this:

```
re_flags = re.IGNORECASE
```

As noted in the *examples*, these flags can also be included within the *regex* pattern itself. It's up to you which technique you prefer.

regex

Mandatory. A python regular expression pattern used to extract information from the linter's output. The pattern must contain at least the following named capture groups:

Name	Description
line	The line number on which the error occurred
message	The error message

Actually the pattern doesn't *have* to have these named capture groups, but if it doesn't you must override the *split_match* method and provide those values yourself.

In addition to the above capture groups, the pattern should contain the following named capture groups when possible:

Name	Description
col	The column number where the error occurred, or a string whose length provides the column number
error	If this is not empty, the error will be marked as an error by SublimeLinter
warning	If this is not empty, the error will be marked as a warning by SublimeLinter
near	If the linter does not provide a column number but mentions a name, match the name with this capture group and SublimeLinter will attempt to highlight that name. Enclosing single or double quotes will be stripped, you may include them in the capture group. If the linter provides a column number, you may still use this capture group and SublimeLinter will highlight that text (stripped of quotes) exactly.

Examples

The output from the flake8 linter looks like this:

```
test.py:12:8: W601 .has_key() is deprecated, use 'in'
test.py:15:11: E271 multiple spaces after keyword
test.py:22:11: E225 missing whitespace around operator
test.py:25:16: F821 undefined name 'barrrrr'
```

The structure of the output is:

```
<file>:<line>:<col> <error code> <message>
```

We translate that into this `regex`:

```
regex = (
    r'^.+?:(?P<line>\d+):(?P<col>\d+): '
    r'(?:(?P<error>[EF])|(?P<warning>[WCN]))\d+ '
    r'(?P<message>.+)'
)
```

A few things to note about this pattern:

- We are using the trick of enclosing multiple strings in parentheses to split the string up visually. Python concatenates them into one string.
- We don't bother capturing the filename (`^.+?:`), it isn't used by SublimeLinter.
- To capture **either** an error or a warning, those capture groups are wrapped in a non-capturing group with alternation.
- Based on the letter prefix of the error code, the linter plugin decides whether to report it to SublimeLinter as an error or a warning.

Here is the output from jsl:

```
(2): lint warning: empty statement or extra semicolon
 b =0;;
......^

(6): warning: variable bar hides argument
    var bar = 'this is a really long string that should be too long';
........^

(10): lint warning: unreachable code
    var i = 0;
....^
```

The structure of the output is:

```
(<line>): <type>: <message>
<code>
<position marker>^
```

We translate that structure into this `regex`:

```
regex = r'''(?xi)
    # First line is (lineno): type: error message
    ^\((?P<line>\d+)\):.*?(?:(?P<warning>warning)|(?P<error>error)):\s*(?P<message>.+)
↪$\r?\n

    # Second line is the line of code
    ^.*$\r?\n

    # Third line is a caret pointing to the position of the error
    ^(?P<col>[^\^]*)\^$
'''
multiline = True
```

A few things to note:

- The *multiline* attribute is set to `True`, because each error message occupies more than one line.
- We set the pattern to be case-insensitive and verbose with `(?xi)`. This could have been done with the *re_flags* attribute, but doing it within the regex pattern is easier.
- We use `\r?\n` at the end of a line to ensure Windows CRLF is matched.
- By capturing the dots before the caret with the `(?P<col>[^\^]*)` pattern, we get the column position of the error on the line.

Note: If you need help constructing, testing or debugging a regular expression, regular expressions 101 provides an easy way to do so.

selectors

If a linter can be used with embedded code, you need to tell SublimeLinter which portions of the source code contain the embedded code by specifying the embedded scope selectors. This attribute maps syntax names to embedded scope selectors.

For example, the HTML syntax uses the scope `source.js.embedded.html` for embedded JavaScript. To allow a JavaScript linter to lint that embedded JavaScript, you would set this attribute to:

```
selectors = {
    'html': 'source.js.embedded.html'
}
```

shebang_match

Some linters may want to turn a shebang into an inline setting. To do so, set this attribute to a callback which receives the first line of code and returns a tuple/list which contains the name and value for the inline setting, or `None` if there is no match.

For example, the `SublimeLinter.lint.PythonLinter` class defines the following:

```
@staticmethod
def match_shebang(code):
    """Convert and return a python shebang as a @python:<version> setting."""

    match = PythonLinter.SHEBANG_RE.match(code)

    if match:
        return '@python', match.group('version')
    else:
        return None

shebang_match = match_shebang
```

syntax

Mandatory. This attribute is the primary way that SublimeLinter associates a linter plugin with files of a given syntax. See *Syntax names* below for info on how to determine the correct syntax names to use.

This may be a single string, or a list/tuple of strings. If the linter supports multiple syntaxes, you may either use a list/tuple of strings, or a single string which begins with `^`, in which case it is compiled as a regular expression pattern which is matched against a syntax name.

If the linter supports embedded syntaxes, be sure to make this attribute a list/tuple or regex pattern which includes the embedding syntax, one of whose values should match one of the keys in the *selectors* dict. For example, `CSSLint` defines the `syntax` and `selectors` attributes as:

```
syntax = ('css', 'html')
selectors = {
    'html': 'source.css.embedded.html'
}
```

Syntax names

The syntax names SublimeLinter uses are based on the **internal** syntax name used by Sublime Text 3, which does not always match the display name. The internal syntax name can be found by doing the following:

1. Open a file which has the relevant syntax, or alternately create a new file and set the syntax in the `View > Syntax` menu.
2. Open the Sublime Text 3 console and enter `view.settings().get('syntax')`. The result will be a path to a `.tmLanguage` file, for example `'Packages/JavaScript/JavaScript.tmLanguage'`.
3. The lowercase filename without the extension (.e.g. `javascript`) is the syntax name SublimeLinter uses.

tempfile_suffix

This attribute configures the behavior of linter executables that cannot receive input from `stdin`.

If the linter executable require input from a file, SublimeLinter can automatically create a temp file from the current code and pass that file to the linter executable. To enable automatic temp file creation, set this attribute to the suffix of the temp file name (with or without a leading '.').

For example, csslint cannot use `stdin`, so the linter plugin does this:

```
tempfile_suffix = 'css'
```

If the suffix needs to be mapped to the syntax of a file, you may make this attribute a dict that maps syntax names (all lowercase, as used in the *syntax* attribute), to temp file suffixes. The name used to lookup the suffix is the mapped syntax, after using *"syntax_map"* in settings. If the view's syntax is not in this map, the class' syntax will be used.

For example, here is a `tempfile_suffix` map for a linter that supports three different syntaxes:

```
tempfile_suffix = {
    'haskell': 'hs',
    'haskell-sublimehaskell': 'hs',
    'literate haskell': 'lhs'
}
```

File-only linters

Some linters can only work from an actual disk file, because they rely on an entire directory structure that cannot be realistically be copied to a temp directory (e.g. `javac`). In such cases, you can mark a linter as "file-only" by setting `tempfile_suffix` to `'-'`.

File-only linters will only run on files that have not been modified since their last save, ensuring that what the user sees and what the linter executable sees is in sync.

version_args

This attribute defines the arguments that should be passed to the linter executable to get its version. It may be a string, in which case it may contains multiple arguments separated by spaces, or it may be a list or tuple containing one argument per element.

For example, most linter executables return the current version when passed `--version` as an argument:

```
version_args = '--version'
```

Note: This attribute should **not** include the linter executable name or path.

---

version_re

This attribute should be a regex pattern or compiled regex used to match the numeric portion of the version returned by executing the linter binary with *version_args*. It must contain a named capture group called "version" that captures only the version, including dots but excluding a prefix such as "v".

For example, `jshint --version` returns `jshint v2.4.1`, so the `version_re` is:

```
version_re = r'\bv(?P<version>\d+\.\d+\.\d+)'
```

Note that we did not try to match "jshint " at the beginning, just in case that text changes in the future.

Note: In general, it is best to make the regex as lenient as possible to allow for changes in the way linter executables format version output.

---

version_requirement

This attribute should be a string which describes the version requirements, suitable for passing to the distutils.versionpredicate.VersionPredicate constructor.

Note: Only the version requirements (what is inside the parens) should be specified here, do not include the package name or parens.

For example, the SublimeLinter-jsl plugin requires version 0.3.x of `jsl`, and will not work with a minor version higher than 3. So the version requirement is:

```
version_requirement = '>= 0.3.0, < 0.4.0'
```

Note that if you were actually constructing a `VersionPredicate`, you would have to pass a string like this:

```
predicate = VersionPredicate('SublimeLinter.jsl (>= 0.3.0, < 0.4.0)')
```

In the case of `version_requirement` however, you only need to specify what is inside the parentheses. SublimeLinter fills in the rest.

word_re

If a linter reports a column position, SublimeLinter highlights the nearest word at that point. By default, SublimeLinter uses the regex pattern `r'^([-\w]+)'` to determine what is a word. You can customize the regex used to highlight words by setting this attribute to a pattern string or a compiled regex.

For example, the csslint linter plugin defines:

```
word_re = r'^(#?[-\w]+)'
```

This allows an id selector such as "#foo" to be highlighted as one word. Without the custom pattern, only the "#" would be highlighted.

Linter Methods

The `SublimeLinter.lint.Linter` class is designed to allow interfacing with most linter executables/libraries through the configuration of class attributes, with no coding necessary. Some linters, however, will need to do more work to set up the environment for the linter executable, or may do the linting directly in the linter plugin itself.

In those cases, you will need to override one or more methods. SublimeLinter provides a well-defined set of methods that are designed to be overridden.

build_options

```
build_options(self, options, type_map, transform=None)
```

This method builds a list of options to be passed directly to a linting method. It is designed for use with linters that do linting directly in python code and need to pass a dict of options. Usually you will call this within the *PythonLinter.check* method.

options is the starting dict of options. *type_map* is a dict that maps an option name to a value of the desired type. If not `None`, *transform* must be a method that takes an option name and returns a transformed name.

For each of the *default settings* marked as an argument, this method does the following:

- Checks if the setting name is in the view settings.
- If so, and the setting's current value is non-empty, checks if the setting name is in *type_map*. If so, the value is converted to the type of the value in *type_map*.
- If *transform* is not `None`, pass the setting name to it to get a transformed name.
- Adds the name/value pair to *options*.

For an example of how `build_options` is used, see the *check* method documentation.

can_lint_syntax

```
can_lint_syntax(cls, syntax)
```

This method returns `True` if a linter can lint a given syntax.

Subclasses may override this if the built in mechanism in the `can_lint` method is not sufficient. When this method is called, `cls.executable_path` has been set to the path of the linter executable. If it is `''`, that means the executable was not specified or could not be found.

cmd

```
cmd(self)
```

If you need to dynamically generate the command line that is executed in order to lint, implement this method in your `Linter` subclass. Return either a command line string or a tuple/list with separate arguments. The first argument in the result should be the full path to the linter executable. If the executable is the same as what you specified in the `executable` class attribute, you can use `self.executable_path`. Otherwise, if you need to find some other executable, you should use the *which* method.

For example, the coffeelint linter plugin does the following:

```
def cmd(self):
    """Return a list with the command line to execute."""

    result = [self.executable_path, '--jslint', '--stdin']

    if persist.get_syntax(self.view) == 'coffeescript_literate':
        result.append('--literate')

    return result
```

communicate

```
communicate(self, cmd, code)
```

This method runs an external executable using the command line specified by the tuple/list *cmd*, passing *code* using `stdin`. The output of the command is returned.

Normally there is no need to call this method, as it called automatically if the linter plugin does not define a value for *tempfile_suffix*. If you override the *run* method you can use this method to execute an external linter that accepts input via `stdin`.

get_view_settings

```
get_view_settings(self, inline=True)
```

If you need to get the *merged settings* for a view, use this method. If *inline* is `False`, inline settings will not be included with the merged settings.

run

```
run(self, cmd, code)
```

This method does the actual linting. *cmd* is a tuple/list of the command to be executed (with arguments), *code* is the text to be linted.

If a linter plugin always uses built-in code (as opposed to a subclass of *PythonLinter* that may use a *module*), it should override this method and return a string as the output. Subclasses of `PythonLinter` that specify a `module` attribute should **not** override this method, but the *check* method instead.

If a linter plugin needs to do complicated setup or will use the *tmpdir* method, it will need to override this method.

split_match

```
split_match(self, match)
```

This method extracts the named capture groups from the *regex* and return a tuple of *match*, *line*, *col*, *error*, *warning*, *message*, *near*.

If subclasses need to modify the values returned by the regex, they should override this method, call `super().split_match(match)`, then modify the values and return them.

For example, the csslint linter plugin overrides `split_match` because it sometimes returns errors without a line number.

```
def split_match(self, match):
    """
    Extract and return values from match.

    We override this method so that general errors that do not have
    a line number can be placed at the beginning of the code.

    """

    match, line, col, error, warning, message, near = super().split_match(match)

    if line is None and message:
        line = 0
        col = 0

    return match, line, col, error, warning, message, near
```

tmpdir

```
tmpdir(self, cmd, files, code)
```

This method creates a temp directory, copies the files in the sequence *files* to the directory, appends the temp directory name to the sequence *cmd*, runs the external executable (with arguments) specified by *cmd*, and returns its output.

Normally there is no need to call this method, but if you override the *run* method you can use this method to execute an external linter that requires a group of files in a specific directory structure.

tmpfile

```
tmpfile(self, cmd, code, suffix='')
```

This method creates a temp file with the filename extension *suffix*, writes *code* to the temp file, appends the temp file name to the sequence *cmd*, runs the external executable (with arguments) specified by *cmd*, and returns its output.

Normally there is no need to call this method, as it is called automatically if the linter plugin defines a value for *tempfile_suffix*. If you override the *run* method you can use this method to execute an external linter that does not accept input via `stdin`.

which

```
which(cls, cmd)
```

This method returns the full path to the executable named in *cmd*. If the executable cannot be found, `None` is returned.

If *cmd* is in the form `script@python[version]`, this method gets the `module` class attribute (which is `None` for non-*PythonLinter* subclasses) and does the following:

- If not `None`, *version* should be a string/numeric version of python to locate, e.g. "3" or "3.3". Only major/minor versions are examined. This method then does its best to locate a version of python that satisfies the requested version. If *module* is not `None`, Sublime Text 3's python version is tested against the requested version.
- If *version* is `None`, the path to the default system python is used, unless *module* is not `None`, in which case "" is returned for the python path.
- If not `None`, *script* should be the name of a python script that is typically installed with easy_install or pip, e.g. `pep8` or `pyflakes`.
- A tuple of the python path and script path is returned.

PythonLinter class

If your linter plugin interfaces with a linter that is written in python, you should subclass from `SublimeLinter.lint.PythonLinter`.

Note: This is done for you if you use the *Create Linter Plugin* command and select `Python` as the linter language.

By doing so, you get the following features:

- *comment_re* is defined correctly for python.
- *@python* is added to *inline_settings*.
- *shebang_match* is set to a method that returns a python shebang as the `@python:<version>` meta setting.
- Execution directly via a python module method or via an external executable.

SublimeLinter-flake8, SublimeLinter-pep8, and SublimeLinter-pyflakes are good examples of python-based linters.

check (method)

```
check(self, code, filename)
```

If your `PythonLinter` subclass sets the *module* attribute, you must implement this method.

This method is called if:

- You set the *module* attribute.
- The named module is successfully imported.
- *Version matching* allows the use of Sublime Text 3's built-in python.

This method should perform linting and return a string with one more lines per error, an array of strings, or an array of objects that can be converted to strings. Here is the `check` method from the `Flake8` class, which is a good template for your own `check` implementations:

```
def check(self, code, filename):
    """Run flake8 on code and return the output."""

    options = {
        'reporter': Report
    }

    type_map = {
        'select': [],
        'ignore': [],
        'max-line-length': 0,
        'max-complexity': 0
    }

    self.build_options(options, type_map, transform=lambda s: s.replace('-', '_'))

    if persist.settings.get('debug'):
        persist.printf('{} options: {}'.format(self.name, options))

    checker = self.module.get_style_guide(**options)

    return checker.input_file(
        filename=os.path.basename(filename),
        lines=code.splitlines(keepends=True)
    )
```

A few things to note:

- We use the *build_options* method to build the options expected by the `get_style_guide` method.
- We print the options to the console if we are in debug mode.

check_version (class attribute)

Some python-based linters are version-sensitive; the python version they are run with has to match the version of the code they lint. If you define the *module* attribute, this attribute should be set to `True` if the linter is version-sensitive.

cmd (class attribute)

When using a python-based linter, there is a special form that should be used for the `cmd` attribute:

```
script@python[version]
```

script is the name of the linter script, and *version* is the optional version of python required by the script.

For example, the SublimeLinter-pyflakes linter plugin defines `cmd` as:

```
cmd = 'pyflakes@python'
```

This tells SublimeLinter to locate the `pyflakes` script and run it on the system python or the version of python configured in settings.

When using the `script@python` form, SublimeLinter does the following:

- Locates *script* in a cross-platform way. Python scripts are installed differently on Windows than they are on Mac OS X and Linux.

- Does version matching between the version specified in the `cmd` attribute and the version specified by settings.
- Defers to using the built-in python if possible.

module (class attribute)

If you want to import a python module and run a method directly in order to lint, this attribute should be set to the module name, suitable for passing to importlib.import_module. During class construction, the named module will be imported, and if successful, the attribute will be replaced with the imported module.

Note: Because the module is going to run within Sublime Text 3, it must be compatible with python 3.3 or later. If not, do not define this attribute.

For example, the `Flake8` linter class defines:

```
module = 'flake8.engine'
```

Later, when it wants to use the method `flake8.engine.get_style_guide`, it does so like this:

```
checker = self.module.get_style_guide(**options)
```

If the module attribute is defined and is successfully imported, whether it is used depends on the following algorithm:

- If the *check_version* attribute is `False`, the module will be used because the module is not version-sensitive.
- If the *"@python"* setting is set and Sublime Text 3's built-in python satisfies that version, the module will be used.
- If the *cmd* attribute specifies `@python` and Sublime Text 3's built-in python satisfies that version, the module will be used. Note that this check is done during class construction.
- Otherwise the external linter executable will be used with the python specified in the *"@python"* setting, the *cmd* attribute, or the default system python.

If you set the `module` attribute, you must implement the *check* in your `PythonLinter` subclass in order to use the module to do the linting.

RubyLinter class

If your linter plugin interfaces with a linter that is written in ruby, you should subclass from `SublimeLinter.lint.RubyLinter`.

Note: This is done for you if you use the *Create Linter Plugin* command and select `Ruby` as the linter language.

By doing so, you get the following features:

- *comment_re* is defined correctly for ruby.
- Support for rbenv and rvm (via rvm-auto-ruby).

rbenv and rvm support

During class construction, SublimeLinter attempts to locate the gem and ruby specified in *cmd*.

The following forms are valid for the first argument of `cmd`:

```
gem@ruby
gem
ruby
```

If `rbenv` is installed and the gem is also under `rbenv` control, the gem will be executed directly. Otherwise `(ruby [, gem])` will be executed.

If `rvm-auto-ruby` is installed, `(rvm-auto-ruby [, gem])` will be executed.

Otherwise `ruby` or `gem` will be executed.

Contributing

If you would like to submit a fix or enhancement to SublimeLinter, thank you!

BEFORE you submit a pull request, please be sure you have followed these steps:

1. Fork the SublimeLinter repo if you haven't already.
2. Create an upstream remote if you haven't already:

   ```
   git remote add -t master -m master -f upstream git@github.com:SublimeLinter/
   ↪SublimeLinter3.git
   ```

3. Create a new branch from the upstream master:

   ```
   git checkout --no-track -b fix upstream/master
   ```

   Feel free to change "fix" to something more descriptive, like "fix-no-args".

4. Make your changes. Please follow the *coding guidelines* below.
5. Commit your changes.
6. When you are ready to push, merge upstream again to make sure your changes will merge cleanly:

   ```
   git pull --rebase upstream/master
   ```

   If there are merge conflicts, fix them, commit the changes, and do this step again until it merges cleanly.

7. Push your branch to your fork:

   ```
   git push -u origin fix
   ```

   Substitute your branch name for "fix".

8. Go to your fork on github and make a pull request. Please give as much information in the description as possible, including the conditions under which the bug occurs, what OS you are using, which linters are affected, sample code which caused the error, etc.

Coding guidelines

I'm a total fanatic about clean code that reads like a story, so please follow these guidelines when writing code for inclusion in SublimeLinter:

- Indent is 4 spaces.
- Code should pass flake8 and pep257 linters.
- Vertical whitespace helps readability, don't be afraid to use it. I especially like to separate any control structures (if/elif, loops, try/except, etc.) from surrounding code by a blank line above and below.
- Please use descriptive variable names, no abbreviations unless it's well known.

Acknowledgements

SublimeLinter has come full circle — twice. Here's a basic timeline of how it evolved:

- Ryan Hileman released the sublimelint plugin for Sublime Text 2.
- Germán Bravo forked sublimelint, added features, and released it as SublimeLinter.
- Aparajita Fishman rewrote the SublimeLinter linter architecture and added more features.
- Jake Swartwood picked up the torch and did a lot of great work maintaining and extending SublimeLinter.
- Sublime Text 3 (ST3) arrived. SublimeLinter needed some work; it didn't run on ST3, and had some serious architectural problems that were making maintenance difficult.
- Jake and Aparajita talked about what to do, and approached Ryan about merging sublimelint and SublimeLinter. Ryan pointed out the fantastic ST3 version of sublimelint he had done.
- Aparajita took one look at that code and realized it was a **way** better foundation for a new ST3 version of SublimeLinter than the existing SublimeLinter codebase.
- Realizing how big the task was, Aparajita solicited donations to fund the development of SublimeLinter 3. The SublimeLinter community responded!
- Aparajita forked the sublimelint ST3 branch and spent two months rewriting SublimeLinter to be better, faster, much easier to use, much easier to configure, and much easier to extend than the previous version.
- Blake Grotewold took over lead development of SublimeLinter 3 after Aparajita's decided to resign from the project.

Special thanks to the main developers and all those in the community who contributed code or money to make SublimeLinter 3 possible!