Sublime Text Help

Release X

guillermooo

December 24, 2016

Contents

1 About This Documentation **1**
1.1 Conventions in This Guide . . . 1

2 Basic Concepts **3**
2.1 The `Data` Directory . . . 3
2.2 The `Packages` Directory . . . 3
2.3 The Python Console . . . 3
2.4 Textmate Compatibility . . . 4
2.5 Be Sublime, My Friend . . . 4

3 Core Features **5**
3.1 Commands . . . 5
3.2 Build Systems . . . 6
3.3 Command Line Usage . . . 7
3.4 Projects . . . 7

4 Customizing Sublime Text **9**
4.1 Settings . . . 9
4.2 Indentation . . . 11
4.3 Key Bindings . . . 11
4.4 Menus . . . 12

5 Extending Sublime Text **13**
5.1 Macros . . . 13
5.2 Snippets . . . 14
5.3 Completions . . . 17
5.4 Command Palette . . . 19
5.5 Syntax Definitions . . . 20
5.6 Plugins . . . 28
5.7 Packages . . . 31

6 FAQ **35**

7 Reference **37**
7.1 Snippets . . . 37
7.2 Syntax Definitions . . . 39
7.3 Build Systems . . . 42
7.4 Key Bindings . . . 45
7.5 Settings (Reference) . . . 49

7.6 Command Palette 51
7.7 Completions 52
7.8 Plugins 54
7.9 Python API 56

CHAPTER 1

About This Documentation

This is the unofficial documentation for the Sublime Text editor, maintained by volunteers. We hope it's useful!

1.1 Conventions in This Guide

- This guide is written from the perspective of a Windows user, but most instructions should only require trivial changes to work on other platforms.
- Relative paths (e. g. `Packages/User`) are rooted at the *data directory* unless otherwise noted. The *data directory* is explained in the Basic Concepts section.
- We assume default key bindings when indicating keyboard shortcuts unless otherwise noted. Due to the way Sublime Text maps keys to commands, **some key bindings won't match your locale's keyboard layout**.

Before you continue reading the material contained in this guide, we encourage you to take a look at the Basic Concepts section.

Happy learning!

CHAPTER 2

Basic Concepts

In these pages we use several forms of *shorthand* and make a few assumptions. Below we explain them in full, in addition to describing fundamental aspects of the Sublime Text 2 editor. All this information will enable you to understand better the content of this guide.

2.1 The `Data` Directory

Sublime Text 2 stores nearly all of the interesting files for users under the data directory. This is a platform-dependent location:

- **Windows**: `%APPDATA%\Sublime Text 2`
- **OS X**: `~/Library/Application Support/Sublime Text 2`
- **Linux**: `~/.Sublime Text 2`
- **Portable Installation**: `Sublime Text 2/Data`

2.2 The `Packages` Directory

This is a location within the data directory that we'll keep referring to again and again. You can obtain it by means of an API call: `sublime.packages_path()`. In this guide, we refer to this location as `Packages`, *packages path* or *packages folder*.

2.2.1 The `User` Package

`Packages/User` is a catch-all directory for custom plugins, snippets, macros, etc. Consider it your personal area in the packages folder. Sublime Text 2 will never overwrite the contents of `Packages/User` during upgrades.

2.3 The Python Console

Sublime Text 2 has an embedded Python interpreter. You will find yourself turning to it often in order to inspect Sublime Text 2 settings and to quickly test API calls while you're writing plugins.

To open the Python console, press `Ctrl+` ` or select **View | Show Console** in the menu.

2.3.1 Your System's Python vs Sublime Text 2 Embedded Python

Sublime Text 2 comes with its own Python interpreter and it's separate from your system's Python installation. This embedded interpreter is intended only to interact with the plugin API, not for general development.

2.4 Textmate Compatibility

Sublime Text 2 is compatible with Textmate snippets, color schemes, `.tmLanguage` files and `.tmPreferences` files.

2.5 Be Sublime, My Friend

Borrowing from Bruce Lee's wisdom, Sublime Text 2 can become almost anything you need it to be. In skilled hands, it can defeat an army of ninjas without your breaking a sweat. Empty your mind. Be sublime, my friend.

CHAPTER 3

Core Features

Built-in features requiring little fiddling on the part of the user to make them meet their needs.

3.1 Commands

Warning: This topic is a draft and could contain wrong information!

Commands are the basic building blocks underlying Sublime Text's automation facilities. Key bindings, menu items, toolbar buttons and macros all work through the command system.

Commands can take parameters and can be bound to a view, a window or the Sublime Text application.

There are built-in and python commands. Built-in commands are included in the editor's core and python commands are defined as plugins (python scripts). Within python commands, there are shipped commands, included by default, and user commands, added by the user. Don't read too much into these categorization, it's merely there for the sake of clarity in these help files–Sublime Text doesn't care about the differences.

3.1.1 Built-in commands

The official documentation only covers Sublime Text v1 at the moment. The main difference between Sublime Text 1 and Sublime Text 2 is naming conventions. Sublime Text 1 used to use ''camelCase'', but Sublime Text 2 uses underscores to separate words: ''fire_gun''. Most of the time, if a command is implemented and documented for Sublime Text 1, it may work by changing its name accordingly.

See official documentation for commands.

3.1.2 Custom commands

Custom commands are created with python plugins.

Naming conventions for custom commands

Command names are written in *CamelCase* and are always suffixed with *Command* (e. g. `MyNewCommand`, `NukeCommand`, `DuplicateLineCommand`).

Sublime Text will unify all command names by removing the *Command* suffix and separating words with underscores. Following with the previous examples, you would call them like this (with `view.run_command` or a similar API call):

- `my_new`
- `nuke`
- `duplicate_line`

Otherwise, Sublime Text wouldn't find them and would fail silently.

3.1.3 Using Commands

There are many ways to use commands, but if you just want to try out one of them, you can use the python console (`CTRL + ~`).

```
# UNTESTED view command
view.run_command("goto_line", {"line": 7})

# UNTESTED window command
view.window().run_command("show_minimap", {"key": True})
```

Note that commands take arguments passed as a dictionary; not `**kwargs`.

3.1.4 Exploring Python Commands

Shipped commands can be found in many packages under the `Packages` folder. In `Packages/Default` you'll find many python commands that are used frequently.

3.1.5 UNSORTED COMMANDS (WORK IN PROGRESS)

auto_complete build clear_fields close_file copy cut decrease_font_size delete_word duplicate_line exec expand_selection find_all_under find_next find_prev find_under find_under_expand find_under_prev focus_group hide_auto_complete hide_overlay hide_panel increase_font_size indent insert insert_snippet join_lines left_delete move move_to move_to_group new_file new_window next_field next_result next_view next_view_in_stack paste paste_and_indent prev_field prev_result prev_view prev_view_in_stack prompt_open_file prompt_save_as prompt_select_project redo redo_or_repeat reindent right_delete run_macro run_macro_file save scroll_lines select_all select_lines set_layout show_overlay show_panel show_scope_name single_selection slurp_find_string slurp_replace_string soft_redo soft_undo sort_lines split_selection_into_lines swap_line_down swap_line_up switch_file toggle_comment toggle_full_screen toggle_overwrite toggle_record_macro toggle_side_bar transpose undo unindent

3.2 Build Systems

See also:

Reference for build systems Complete documentation on all available options, variables, etc.

Build systems provide a convenient way to supply arguments and environmental information to external programs and run them. Use build systems if you need to run your files through build programs like **`make`**, command line utilities like **`tidy`**, interpreters, etc.

Note: Executables called from build systems must be in your `PATH`.

3.2.1 File Format

Like many other configuration files in Sublime Text, build systems are JSON files. Fittingly, they have the extension `.sublime-build`.

Example

Here's an example of a build system:

```
{
    "cmd": ["python", "-u", "$file"],
    "file_regex": "^[ ]*File \"(...*?)\", line ([0-9]*)",
    "selector": "source.python"
}
```

`cmd` Required. This option contains the actual command line to be executed:

```
python -u /path/to/current/file.ext
```

`file_regex` A Perl-style regular expression to capture error information out of the external program's output. This information is then used to help you navigate through error instances with `F4`.

`selector` If the **Tools | Build System | Automatic** option is set, Sublime Text will use this information to determine which build system is to be used for the active file.

In addition to options, you can use variables in build systems too, like we've done above with `$file`, which expands to the full path of the file underlying the active buffer.

3.2.2 Where to Store Build Systems

Build systems must be located somewhere under the `Packages` folder (e. g. `Packages/User`).

3.2.3 Running Build Systems

Build systems can be run by pressing `F7` or from **Tools | Build**.

3.3 Command Line Usage

See also:

OS X Command Line Official Sublime Text Documentation

3.4 Projects

See also:

Projects Official Sublime Text Documentation

CHAPTER 4

Customizing Sublime Text

Sublime Text is highly customizable. In the topics below, we'll explain you how you can adapt it to your needs and preferences.

4.1 Settings

Sublime Text uses `.sublime-settings` files to store configuration data. Settings control many aspects of the editor, from visual layout to file type settings.

4.1.1 Format

Settings files use JSON and have the `.sublime-settings` extension. The purpose of a `.sublime-settings` file is determined by its name. For example, `Python.sublime-settings` controls settings for Python files, whereas `Minimap.sublime-settings` controls the minimap settings, etc.

4.1.2 Types of Settings

As mentioned above, there are several types of `.sublime-settings` files controlling several aspects of the editor. In this section only file type settings are explained.

4.1.3 File Settings

A hierarchy of `.sublime-settings` files controls settings specific to a file type. Therefore, the location of settings matters. As it's always the case when merging files of any kind, Sublime Text gives the top priority to files in the `User` package. See the section *Merging and Order of Precedence* for more information.

In addition, there's yet another layer of settings that overrides the others: the *session*. Session data is updated as you work on a file, so if you adjust settings for a file in any way (mainly through API calls), they will be recorded in the session and will take precedence over any `.sublime-settings` files. Calls to `obj.settings().get()` always return the value in effect for `obj`.

When untangling the applicable settings for a file at any time, one must also keep in mind that Sublime Text adjusts settings automatically in some situations. For example, if `auto_detect_indentation` is on, the value a call to `view.settings().get('tab_size')` returns might appear unexpected, especially if you've explicitly set `tab_size` moments earlier.

Below, you can see the order in which Sublime Text would process a hypothetical hierarchy of settings for Python on Windows:

- `Packages/Default/Base File.sublime-settings`
- `Packages/Default/Base File (Windows).sublime-settings`
- `Packages/User/Base File.sublime-settings`
- `Packages/Python/Python.sublime-settings`
- `Packages/User/Python.sublime-settings`
- *Session data for the current file*
- *Auto adjusted settings*

4.1.4 Global File Type Settings

There are two types of global settings files affecting file types:

- `Base File.sublime-settings` and
- `Base File (<platform>).sublime-settings`.

`Base File` is always in effect for all platforms, whereas `Base File (<platform>)` only applies to the named platform. Multiple `Base File` and `Base File (<platform>)` files can coexist with the exception of `Packages/User`. From `Packages/User`, only `Base File` will be read. This is so there is only one global file that overrides all the other ones.

Legal values for `<platform>` are: `Linux`, `OSX` and `Windows`.

4.1.5 Settings Specific to a File Type

If you want to target a specific file type in a `.sublime-settings` file, give it the name of the applicable syntax definition for said file type. Note you have to use the syntax definition's *file name*, not a *scope name*. For example, if our syntax definition was called `Python.tmLanguage`, we'd need to call our settings file `Python.sublime-settings`.

Settings files for specific file types usually live in packages, like `Packages/Python`, but there can be multiple settings files for the same file type in separate locations. Similarly to global settings, one can establish platform-specific settings for file types. For example, `Python (Linux).sublime-settings` would only be consulted under Linux. Also, under `Pakages/User` only `Python.sublime-settings` would be read, but not `Python (<platform>).sublime-settings`.

Regardless of its location, any file-type-specific settings file has precedence over every global settings file affecting file types.

4.1.6 Where to Store User Settings

Whenever you want to persist settings, especially if they should be preserved between upgrades, place the relevant `.sublime-settings` file in `Packages/User`. This is the recommended place to store user settings.

You can nevertheless save settings files under other subdirectories of `Packages`. For example, `Packages/ZZZ/Python.sublime-settings` would override `Packages/Python/Python.sublime-settings` by virtue of alphabetical order. However, `Packages/User/Python.sublime-settings` would continue to have the highest precedence for the Python file type settings.

4.2 Indentation

See also:

Indentation Official Sublime Text Documentation.

4.3 Key Bindings

See also:

Reference for key bindings Complete documentation on key bindings.

Key bindings let you map sequences of key presses to actions.

4.3.1 File Format

Key bindings are defined in JSON and stored in `.sublime-keymap` files. In order to integrate better with each platform, there are separate key map files for Linux, OSX and Windows. Only key maps for the corresponding platform will be loaded.

Example

Here's an excerpt from the default key map for Windows:

```
[
        { "keys": ["ctrl+shift+n"], "command": "new_window" },
        { "keys": ["ctrl+o"], "command": "prompt_open_file" }
]
```

4.3.2 Defining and Overriding Key Bindings

Sublime Text ships with a default key map (e. g. `Packages/Default/Default (Windows).sublime-keymap`). In order to override key bindings defined there or add new ones, you can store them in aseparate key map with a higher precedence, for example `Packages/User/Default (Windows).sublime-keymap`.

See *Merging and Order of Precedence* for more information about how Sublime Text sorts files for merging.

4.3.3 Advanced Key Bindings

Simple key bindings consist of a key combination and a command to be executed. However, there are more complex syntaxes to pass arguments and provide contextual awareness.

Passing Arguments

Arguments are specified in the `args` key:

```
{ "keys": ["shift+enter"], "command": "insert", "args": {"characters": "\n"} }
```

Here, `\n` is passed to the `insert` command when you press `Shift+Enter`.

Contexts

Contexts determine when a given key binding will be enabled based on the caret's position or some other state.

```
{ "keys": ["escape"], "command": "clear_fields", "context":
        [
                { "key": "has_next_field", "operator": "equal", "operand": true }
        ]
}
```

This key binding translates to *clear snippet fields and resume normal editing if there is a next field available*. Thus, pressing `ESC` when you are not cycling through snippet fields will **not** trigger this key binding (however, something else might occur instead if `ESC` happens to be bound to a different context too —and that's likely to be the case for `ESC`).

4.4 Menus

No documenation available about this topic.

But here's Bruce Lee screaming.

CHAPTER 5

Extending Sublime Text

As it can be seen from the long list of topics below, Sublime Text is a very extensible editor.

5.1 Macros

Macros are a basic automation facility consisting in sequences of commands. Use them whenever you need to repeat the exact same steps to perform an operation.

Macro files are JSON files with the extension `.sublime-macro`. Sublime Text ships with a few macros providing core functionality, such as line and word deletion. You can find these under **Tools | Macros**.

5.1.1 How to Record Macros

To start recording a macro, press `Ctrl+q` and subsequently execute the desired steps one by one. When you're done, press `Ctrl+q` again to stop the macro recorder. Your new macro won't be saved to a file, but kept in the macro buffer instead. You will now be able to run the recorded macro by pressing `Ctrl+Shift+q` or save it to a file by selecting **Tools | Save macro...**.

Note that the macro buffer will only remember the macro recorded latest. Also, recorded macros only capture commands sent to the buffer: window level commands, such as creating a new file, will be ignored.

5.1.2 How to Edit Macros

Alternatively to recording a macro, you can edit it by hand. Save a new file with the extension `.sublime-macro` under `PackagesUser` and add commands to it. This is how a macro file looks like:

```
[
    {"command": "move_to", "args": {"to": "hardeol"}},
    {"command": "insert", "args": {"characters": "\n"}}
]
```

See the Commands section for more information on commands.

If you're editing a macro by hand, you need to escape quotation marks, blank spaces and backslashes by preceding them with `\`.

5.1.3 Where to Store Macros

Macro files can be stored in any package folder, and they will show up under **Tools | Macros | <PackageName>**.

5.2 Snippets

Whether you are coding or writing the next vampire best-seller, you're likely to need certain short fragments of text again and again. Use snippets to save yourself tedious typing. Snippets are smart templates that will insert text for you and adapt it to their context.

To create a new snippet, select **Tools | New Snippet...**. Sublime Text will present you with an skeleton for a new snippet.

Snippets can be stored under any package's folder, but to keep it simple while you're learning, you can save them to your `Packages/User` folder.

5.2.1 Snippets File Format

Snippets typically live in a Sublime Text package. They are simplified XML files with the extension `sublime-snippet`. For instance, you could have a `greeting.sublime-snippet` inside an `Email` package.

The structure of a typical snippet is as follows (including the default hints Sublime Text inserts for your convenience):

```
<snippet>
    <content><![CDATA[Type your snippet here]]></content>
    <!-- Optional: Tab trigger to activate the snippet -->
    <tabTrigger>xyzzy</tabTrigger>
    <!-- Optional: Scope the tab trigger will be active in -->
    <scope>source.python</scope>
    <!-- Optional: Description to show in the menu -->
    <description>My Fancy Snippet</description>
</snippet>
```

The `snippet` element contains all the information Sublime Text needs in order to know *what* to insert, *whether* to insert it and *when*. Let's see all of these parts in turn.

`content` The actual snippet. Snippets can range from simple to fairly complex templates. We'll look at examples of both later.

Keep the following in mind when writing your own snippets:

- If you want the get a literal `$`, you have to escape it like this: `\$`.
- When writing a snippet that contains indentation, always use tabs. The tabs will be transformed into spaces when the snippet is inserted if the option `translateTabsToSpaces` is set to `true`.

The `content` must be included in a `<![CDATA[...]]>` section. Snippets won't work if you don't do this!

`tabTrigger` Defines the sequence of keys you will press to insert this snippet. The snippet will kick in as soon as you hit the `Tab` key after typing this sequence.

A tab trigger is an implicit key binding.

`scope` Scope selector determining the context where the snippet will be active. See *Scopes* for more information.

`description` Used when showing the snippet in the Snippets menu. If not present, Sublime Text defaults to the name of the snippet.

With this information, you can start writing your own snippets as described in the next sections.

Note: In the interest of brevity, we're only including the `content` element's text in examples unless otherwise noted.

5.2.2 Snippet Features

Environment Variables

Snippets have access to contextual information in the form of environment variables. Sublime Text sets the values of the variables listed below automatically.

You can also add your own variables to provide extra information. These custom variables are defined in `.sublime-options` files.

$PARAM1, $PARAM2...	Arguments passed to the `insert_snippet` command. (Not covered here.)
$SELECTION	The text that was selected when the snippet was triggered.
$TM_CURRENT_LINE	Content of the line the cursor was in when the snippet was triggered.
$TM_CURRENT_WORD	Current word under the cursor when the snippet was triggered.
$TM_FILENAME	File name of the file being edited including extension.
$TM_FILEPATH	File path to the file being edited.
$TM_FULLNAME	User's user name.
$TM_LINE_INDEX	Column the snippet is being inserted at, 0 based.
$TM_LINE_NUMBER	Row the snippet is being inserted at, 1 based.
$TM_SELECTED_TEXT	An alias for **$SELECTION**.
$TM_SOFT_TABS	`YES` if `translate_tabs_to_spaces` is true, otherwise `NO`.
$TM_TAB_SIZE	Spaces per-tab (controlled by the `tab_size` option).

Let's see a simple example of a snippet using variables:

```
====================================
USER NAME:          $TM_FULLNAME
FILE NAME:          $TM_FILENAME
 TAB SIZE:          $TM_TAB_SIZE
SOFT TABS:          $TM_SOFT_TABS
====================================

# Output:
====================================
USER NAME:          guillermo
FILE NAME:          test.txt
 TAB SIZE:          4
SOFT TABS:          YES
====================================
```

Fields

With the help of field markers, you can cycle through positions within the snippet by pressing the `Tab` key. Fields are used to walk you through the customization of a snippet once it's been inserted.

```
First Name: $1
Second Name: $2
Address: $3
```

In the example above, the cursor will jump to `$1` if you press `Tab` once. If you press `Tab` a second time, it will advance to `$2`, etc. You can also move backwards in the series with `Shift+Tab`. If you press `Tab` after the highest tab stop, Sublime Text will place the cursor at the end of the snippet's content so that you can resume normal editing.

If you want to control where the exit point should be, use the `$0` mark.

You can break out of the field cycle any time by pressing `Esc`.

Mirrored Fields

Identical field markers mirror each other: when you edit the first one, the rest will be populated with the same value in real time.

```
First Name: $1
Second Name: $2
Address: $3
User name: $1
```

In this example, "User name" will be filled out with the same value as "First Name".

Place Holders

By expanding the field syntax a little bit, you can define default values for a field. Place holders are useful when there's a general case for your snippet but you still want to keep its customization convenient.

```
First Name: ${1:Guillermo}
Second Name: ${2:López}
Address: ${3:Main Street 1234}
User name: $1
```

Variables can be used as place holders:

```
First Name: ${1:Guillermo}
Second Name: ${2:López}
Address: ${3:Main Street 1234}
User name: ${4:$TM_FULLNAME}
```

And you can nest place holders within other place holders too:

```
Test: ${1:Nested ${2:Placeholder}}
```

Substitutions

> **Warning:** This section is a draft and may contain inaccurate information.

In addition to the place holder syntax, tab stops can specify more complex operations with substitutions. Use substitutions to dynamically generate text based on a mirrored tab stop.

The substitution syntax has the following syntaxes:

- `${var_name/regex/format_string/}`
- `${var_name/regex/format_string/options}`

var_name The variable name: 1, 2, 3…

regex Perl-style regular expression: See the Boost library reference for regular expressions.

format_string See the Boost library reference for format strings.

options

Optional. May be any of the following:

i Case-insensitive regex.

g Replace all occurrences of `regex`.

m Don't ignore newlines in the string.

With substitutions you can, for instance, underline text effortlessly:

```
      Original: ${1:Hey, Joe!}
Transformation: ${1/./=/g}

# Output:

      Original: Hey, Joe!
Transformation: =========
```

5.3 Completions

See also:

Reference for completions Complete documentation on all available options.

Sublime Text Documentation Official documentation on this topic.

Completions provide functionality in the spirit of IDEs to suggest terms and insert snippets. Completions work through the completions list or, optionally, by pressing `Tab`.

Note that completions in the broader sense of *words that Sublime Text will look up and insert for you* are not limited to completions files, because other sources contribute to the list of words to be completed, namely:

- Snippets
- API-injected completions
- Buffer contents

However, `.sublime-completions` files are the most explicit way Sublime Text provides you to feed it completions. This topic deals with the creation of `.sublime-completions` files as well as with the interaction between all sources for completions.

5.3.1 File Format

Completions are JSON files with the `.sublime-completions` extension. Entries in completions files can contain either snippets or plain strings.

Example

Here's an excerpt from the HTML completions:

```
{
        "scope": "text.html - source - meta.tag, punctuation.definition.tag.begin",

        "completions":
```

```
        [
                { "trigger": "a", "contents": "<a href=\"$1\">$0</a>" },
                { "trigger": "abbr", "contents": "<abbr>$0</abbr>" },
                { "trigger": "acronym", "contents": "<acronym>$0</acronym>" }
        ]
}
```

scope Determines when the completions list will be populated with this list of completions. See *Scopes* for more information.

In the example above, we've used trigger-based completions only, but completions files support simple completions too. Simple completions are just plain strings. Expanding our example with a few simple completions, we'd end up with a list like so:

```
{
        "scope": "text.html - source - meta.tag, punctuation.definition.tag.begin",

        "completions":
        [
                { "trigger": "a", "contents": "<a href=\"$1\">$0</a>" },
                { "trigger": "abbr", "contents": "<abbr>$0</abbr>" },
                { "trigger": "acronym", "contents": "<acronym>$0</acronym>" },

                "ninja",
                "robot",
                "pizza"
        ]
}
```

5.3.2 Sources for Completions

Completions not only originate in `.sublime-completions` files. This is the exhaustive list of sources for completions:

- Snippets
- API-injected completions
- `.sublime-completions` files
- Words in buffer

Priority of Sources for Completions

This is the order in which completions are prioritized:

- Snippets
- API-injected completions
- `.sublime-completions` files
- Words in buffer

Snippets will always win if the current prefix matches their tab trigger exactly. For the rest of the completions sources, a fuzzy match is performed. Also, snippets will always lose against a fuzzy match. Note that this is only relevant if the completion is going to be inserted automatically. When the completions list is shown, snippets will be listed along the other items, even if the prefix only partially matches the snippets' tab triggers.

5.3.3 How to Use Completions

There are two methods to use completions, and although the priority given to completions when screening them is always the same, there is a difference in the result that will be explained below.

Completions can be inserted in two ways:

- through the completions list (`Ctrl+spacebar`);
- by pressing `Tab`.

The Completions List

The completions list (`Ctrl+spacebar`) may work in two ways: by bringing up a list of suggested words to be completed, or by inserting the best match directly.

If the choice of best completion is ambiguous, an interactive list will be presented to the user, who will have to select an item himself. Unlike other items, snippets in this list are displayed in this format: `<tab_trigger> : <name>`, where `<tab_trigger>` and `<name>` are variable.

The completion with `Ctrl+spacebar` will only be automatic if the list of completion candidates can be narrowed down to one unambiguous choice given the current prefix.

`Tab`-completed Completions

If you want to be able to tab-complete completions, the setting `tab_completion` must be set to `true`. By default, `tab_completion` is set to `true`. Snippet tab-completion is unaffected by this setting: they will always be completed according to their tab trigger.

With `tab_completion` enabled, completion of items is always automatic, which means that, unlike in the case of the completions list, Sublime Text will always make a decision for you. The rules to select the best completion are the same as above, but in case of ambiguity, Sublime Text will still insert the item deemed most suitable.

Inserting a Literal Tab Character

When `tab_completion` is enabled, you can press `Shift+Tab` to insert a literal tab character.

5.4 Command Palette

See also:

Reference for Command Palette Complete documentation on the command palette options.

5.4.1 Overview

The *command palette* is an interactive list bound to `Ctrl+Shift+P` whose purpose is to execute commands. The command palette is fed entries with commands files. Usually, commands that don't warrant creating a key binding of their own are good candidates for inclusion in a `.sublime-commands` file.

5.4.2 File Format (Commands Files)

Commands files use JSON and have the `.sublime-commands` extension.

Here's an excerpt from `Packages/Default/Default.sublime-commands`:

```
[
    { "caption": "Project: Save As", "command": "save_project_as" },
    { "caption": "Project: Close", "command": "close_project" },
    { "caption": "Project: Add Folder", "command": "prompt_add_folder" },

    { "caption": "Preferences: Default File Settings", "command": "open_file", "args": {"file": "${pa
    { "caption": "Preferences: User File Settings", "command": "open_file", "args": {"file": "${packa
    { "caption": "Preferences: Default Global Settings", "command": "open_file", "args": {"file": "$
    { "caption": "Preferences: User Global Settings", "command": "open_file", "args": {"file": "${pac
    { "caption": "Preferences: Browse Packages", "command": "open_dir", "args": {"dir": "$packages"}
]
```

`caption` Text for display in the command palette.

`command` Command to be executed.

`args` Arguments to pass to `command`.

5.4.3 How to Use the Command Palette

1. Press `Ctrl+Shift+P`
2. Select command

The command palette filters entries by context, so whenever you open it, you won't always see all the commands defined in every `.sublime-commands` file.

5.5 Syntax Definitions

Syntax definitions make Sublime Text aware of programming and markup languages. Most noticeably, they work together with colors to provide syntax highlighting. Syntax definitions define *scopes* in a buffer that divide the text in named regions. Several editing features in Sublime Text make extensive use of this fine-grained contextual information.

Essentially, syntax definitions consist of regular expressions used to find text, and more or less arbitrary, dot-separated strings called *scopes* or *scope names*. For every occurrence of a given regular expression, Sublime Text gives the matching text its corresponding *scope name*.

5.5.1 Prerequisites

In order to follow this tutorial, you will need to install AAAPackageDev, a package intended to ease the creation of new syntax definitions for Sublime Text. AAAPackageDev lives on a public Mercurial repository at Bitbucket.

Download the latest `.sublime-package` file and double click on it if you're running a full installation of Sublime Text, or perform a manual installation as described in *Installation of .sublime-package Files*.

Mercurial and Bitbucket

Mercurial is a distributed version control system (DVCS). Bitbucket is an online service that provides hosting for Mercurial repositories. If you want to install Mercurial, there are freely available command-line and graphical clients.

5.5.2 File format

Sublime uses property list files (Plist) to store syntax definitions. Because editing XML files is a cumbersome task, though, we'll be using JSON instead and converting it to Plist afterwards. This is where the AAAPackageDev package mentioned above comes in.

Note: If you experience unexpected errors during this tutorial, chances are AAAPackageDev is to blame. Don't immediately think your problem is due to a bug in Sublime Text.

By all means, do edit the Plist files by hand if you prefer to work in XML, but keep always in mind the differing needs with regards to escape sequences, etc.

5.5.3 Scopes

Scopes are a key concept in Sublime Text. Essentially, they are named text regions in a buffer. They don't do anything by themselves, but Sublime Text peeks at them when it needs contextual information.

For instance, when you trigger a snippet, Sublime Text checks the scope the snippet's bound to and looks at the caret's position in the file. If the caret's current scope matches the snippet's scope selector, Sublime Text fires the snippet off. Otherwise, nothing happens.

Scopes vs Scope Selectors

There's a slight difference between *scopes* and *scope selectors*: scopes are the names defined in a syntax definition, whilst scope selectors are used in items like snippets and key bindings to target scopes. When creating a new syntax definition, you care about scopes; when you want to constrain a snippet to a certain scope, you use a scope selector.

Scopes can be nested to allow for a high degree of granularity. You can drill down the hierarchy very much like with CSS selectors. For instance, thanks to scope selectors, you could have a key binding activated only within single quoted strings in python source code, but not inside single quoted strings in any other language.

Sublime Text implements the idea of scopes from Texmate, a text editor for Mac. Textmate's online manual contains further information about scope selectors that's useful for Sublime Text users too.

5.5.4 How Syntax Definitions Work

At their core, syntax definitions are arrays of regular expressions paired with scope names. Sublime Text will try to match these patterns against a buffer's text and attach the corresponding scope name to all occurrences. These pairs of regular expressions and scope names are known as *rules*.

Rules are applied in order, one line at a time. Each rule consumes the matched text region, which will therefore be excluded from the next rule's matching attempt (save for a few exceptions). In practical terms, this means that

you should take care to go from more specific rules to more general ones when you create a new syntax definition. Otherwise, a greedy regular expression might swallow parts you'd like to have styled differently.

Syntax definitions from separate files can be combined, and they can be recursively applied too.

5.5.5 Your First Syntax Definition

By way of example, let's create a syntax definition for Sublime Text snippets. We'll be styling the actual snippet content, not the `.sublime-snippet` file.

Note: Since syntax definitions are primarily used to enable syntax highlighting, we'll use *to style* as in *to break down a source code file into scopes*. Keep in mind, however, that colors are a different thing to syntax definitions and that scopes have many more uses besides syntax highlighting.

These are the elements we want to style in a snippet:

- Variables (`$PARAM1`, `$USER_NAME...`)
- Simple fields (`$0`, `$1...`)
- Complex fields with place holders (`${1:Hello}`)
- Nested fields (`${1:Hello ${2:World}!}`)
- Escape sequences (`\\$`, `\\<...`)
- Illegal sequences (`$`, `<...`)

Note: Before continuing, make sure you've installed the AAAPackageDev package as explained further above.

Creating A New Syntax Definition

To create a new syntax definition, follow these steps:

- Go to **Tools | Packages | Package Development | New Syntax Definition**
- Save the new file to your `Packages/User` folder as `Sublime Snippets (Raw).JSON-tmLanguage`.

You should now see a file like this:

```
{ "name": "Syntax Name",
  "scopeName": "source.syntax_name",
  "fileTypes": [""],
  "patterns": [
  ],
  "uuid": "ca03e751-04ef-4330-9a6b-9b99aae1c418"
}
```

Let's examine now the key elements.

`uuid` Located at the end, this is a unique identifier for this syntax definition. Each new syntax definition gets its own uuid. Don't modify them.

`name` The name that Sublime Text will display in the syntax definition drop-down list Use a short, descriptive name. Typically, you will be using the programming language's name you are creating the syntax definition for.

`scopeName` The top level scope for this syntax definition. It takes the form `source.<lang_name>` or `text.<lang_name>`. For programming languages, use `source`. For markup and everything else, `text`.

fileTypes This is a list of file extensions. When opening files of these types, Sublime Text will automatically activate this syntax definition for them.

patterns Container for your patterns.

For our example, fill in the template with the following information:

```
{    "name": "Sublime Snippet (Raw)",
     "scopeName": "source.ssraw",
     "fileTypes": ["ssraw"],
     "patterns": [
     ],
     "uuid": "ca03e751-04ef-4330-9a6b-9b99aae1c418"
}
```

Note: JSON is a very strict format, so make sure to get all the commas and quotes right. If the conversion to Plist fails, take a look at the output panel for more information on the error. We'll explain later how to convert a syntax definition in JSON to Plist.

5.5.6 Analyzing Patterns

The `patterns` array can contain several types of elements. We'll look at some of them in the following sections. If you want to learn more about patterns, refer to Textmate's online manual.

Regular Expressions' Syntax In Syntax Definitions

Sublime Text uses Oniguruma's syntax for regular expressions in syntax definitions. Several existing syntax definitions make use of features supported by this regular expression engine that aren't part of perl-style regular expressions, hence the requirement for Oniguruma.

Matches

They take this form:

```
{ "match": "[Mm]y \s+[Rr]egex",
  "name": "string.ssraw",
  "comment": "This comment is optional."
}
```

match A regular expression Sublime Text will use to try and find matches.

name Name of the scope that should be applied to the occurrences of `match`.

comment An optional comment about this pattern.

Let's go back to our example. Make it look like this:

```
{ "name": "Sublime Snippet (Raw)",
  "scopeName": "source.ssraw",
  "fileTypes": ["ssraw"],
  "patterns": [
  ],
  "uuid": "ca03e751-04ef-4330-9a6b-9b99aae1c418"
}
```

That is, make sure the `patterns` array is empty.

Now we can begin to add our rules for Sublime snippets. Let's start with simple fields. These could be matched with a regex like so:

```
\$[0-9]+
# or...
\$\d+
```

However, because we're writing our regex in JSON, we need to factor in JSON's own escaping rules. Thus, our previous example becomes:

```
\\$\\d+
```

With escaping out of the way, we can build our pattern like this:

```
{ "match": "\\$\\d+",
  "name": "keyword.source.ssraw",
  "comment": "Tab stops like $1, $2..."
}
```

Choosing the Right Scope Name

Naming scopes isn't obvious sometimes. Check the Textmate online manual for guidance on scope names. It is important to re-use the basic categories outlined there if you want to achieve the highest compatibility with existing colors.
Colors have hardcoded scope names in them. They could not possibly include every scope name you can think of, so they target the standard ones plus some rarer ones on occasion. This means that two colors using the same syntax definition may render the text differently!
Bear in mind too that you should use the scope name that best suits your needs or preferences. It'd be perfectly fine to assign a scope like `constant.numeric` to anything other than a number if you have a good reason to do so.

And we can add it to our syntax definition too:

```
{   "name": "Sublime Snippet (Raw)",
    "scopeName": "source.ssraw",
    "fileTypes": ["ssraw"],
    "patterns": [
        { "match": "\\$\\d+",
          "name": "keyword.source.ssraw",
          "comment": "Tab stops like $1, $2..."
        }
    ],
    "uuid": "ca03e751-04ef-4330-9a6b-9b99aae1c418"
}
```

We're now ready to convert our file to `.tmLanguage`. Syntax definitions use Textmate's `.tmLanguage` extension for compatibility reasons. As explained further above, they are simply XML files in the Plist format.

Follow these steps to perform the conversion:

- Select `Json to tmLanguage` in **Tools | Build System**
- Press `F7`
- A `.tmLanguage` file will be generated for you in the same folder as your `.JSON-tmLanguage` file
- Restart Sublime Text so all your changes can take effect

Note: Sublime Text cannot reload syntax definitions automatically upon their modification.

You have now created your first syntax definition. Next, open a new file and save it with the extension `.ssraw`. The buffer's syntax name should switch to "Sublime Snippet (Raw)" automatically, and you should get syntax highlighting if you type `$1` or any other simple snippet field.

Let's proceed to creating another rule for environment variables.

```
{ "match": "\\$[A-Za-z][A-Za-z0-9_]+",
  "name": "keyword.source.ssraw",
  "comment": "Variables like $PARAM1, $TM_SELECTION..."
}
```

Repeat the steps above to update the `.tmLanguage` file and restart Sublime Text.

Fine Tuning Matches

You might have noticed that the entire text in `$PARAM1`, for instance, is styled the same way. Depending on your needs or your personal preferences, you may want the `$` to stand out. That's where `captures` come in. Using captures, you can break a pattern down into components to target them individually.

Let's rewrite one of our previous patterns to use `captures`:

```
{ "match": "\\$([A-Za-z][A-Za-z0-9_]+)",
  "name": "keyword.ssraw",
  "captures": {
      "1": { "name": "constant.numeric.ssraw" }
   },
  "comment": "Variables like $PARAM1, $TM_SELECTION..."
}
```

Captures introduce complexity to your rule, but they are pretty straightforward. Notice how numbers refer to parenthesized groups left to right. Of course, you can have as many capture groups as you want.

Arguably, you'd want the other scope to be visually consistent with this one. Go ahead and change it too.

Begin-End Rules

Up to now we've been using a simple rule. Although we've seen how to dissect patterns into smaller components, sometimes you'll want to target a larger portion of your source code clearly delimited by start and end marks.

Literal strings enclosed in quotation marks and other delimited constructs are better dealt with with begin-end rules. This is a skeleton for one of these rules:

```
{ "name": "",
  "begin": "",
  "end": ""
}
```

Well, at least in their simplest version. Let's take a look at one including all available options:

```
{ "name": "",
  "begin": "",
  "beginCaptures": {
    "0": { "name": "" }
  },
  "end": "",
```

```
  "endCaptures": {
    "0": { "name": "" }
  },
  "patterns": [
     {  "name": "",
        "match": ""
                  }
  ],
  "contentName": ""
}
```

Some elements may look familiar, but their combination might be daunting. Let's see them individually.

`begin` Regex for the opening mark for this scope.

`end` Regex for the end mark for this scope.

`beginCaptures` Captures for the `begin` marker. They work like captures for simple matches. Optional.

`endCaptures` Same as `beginCaptures` but for the `end` marker. Optional.

`contentName` Scope for the whole matched region, from the begin marker to the end marker, inclusive. This will effectively create nested scopes for `beginCaptures`, `endCaptures` and `patterns` defined within this rule. Optional.

`patterns` An array of patterns to match against the begin-end content **only** —they are not matched against the text consumed by `begin` or `end`.

We'll use this rule to style nested complex fields in snippets:

```
{ "name": "variable.complex.ssraw",
   "begin": "(\\$)(\\{)([0-9]+):",
   "beginCaptures": {
       "1": { "name": "keyword.ssraw" },
       "3": { "name": "constant.numeric.ssraw" }
   },
   "patterns": [
       { "include": "$self" },
       {  "name": "string.ssraw",
          "match": "."
       }
   ],
   "end": "\\}"
}
```

This is the most complex pattern we'll see in this tutorial. The `begin` and `end` keys are self-explanatory: they define a region enclosed between `${<NUMBER>:` and `}`. `beginCaptures` further divides the begin mark into smaller scopes.

The most interesting part, however, is `patterns`. Recursion and the importance of ordering have finally made an appearance here.

We've seen further above that fields can be nested. In order to account for this, we need to recursively style nested fields. That's what the `include` rule does when furnished the `$self` value: it recursively applies our entire syntax definition to the portion of text contained in our begin-end rule, excluding the text consumed by both `begin` and `end`.

Remember that matched text is consumed and is excluded from the next match attempt.

To finish off complex fields, we'll style place holders as strings. Since we've already matched all possible tokens inside a complex field, we can safely tell Sublime Text to give any remaining text (`.`) a literal string scope.

Final Touches

Lastly, let's style escape sequences and illegal sequences, and wrap up.

```
{   "name": "constant.character.escape.ssraw",
    "match": "\\\\(\\$|\\>|\\<)"
},

{   "name": "invalid.ssraw",
    "match": "(\\$|\\<|\\>)"
}
```

The only hard thing here is getting the number of escape characters right. Other than that, the rules are pretty straightforward if you're familiar with regular expressions.

However, you must take care to put the second rule after any others matching the `$` character, since otherwise you may not get the desired result.

Also, note that after adding these two additional rules, our recursive begin-end rule above keeps working as expected.

At long last, here's the final syntax definition:

```
{    "name": "Sublime Snippet (Raw)",
     "scopeName": "source.ssraw",
     "fileTypes": ["ssraw"],
     "patterns": [
         { "match": "\\$(\\d+)",
           "name": "keyword.ssraw",
           "captures": {
               "1": { "name": "constant.numeric.ssraw" }
            },
           "comment": "Tab stops like $1, $2..."
         },

         { "match": "\\$([A-Za-z][A-Za-z0-9_]+)",
           "name": "keyword.ssraw",
           "captures": {
               "1": { "name": "constant.numeric.ssraw" }
            },
           "comment": "Variables like $PARAM1, $TM_SELECTION..."
         },

         { "name": "variable.complex.ssraw",
           "begin": "(\\$)(\\{)([0-9]+):",
           "beginCaptures": {
               "1": { "name": "keyword.ssraw" },
               "3": { "name": "constant.numeric.ssraw" }
            },
            "patterns": [
               { "include": "$self" },
               { "name": "string.ssraw",
                 "match": "."
               }
            ],
            "end": "\\}"
         },

         { "name": "constant.character.escape.ssraw",
           "match": "\\\\(\\$|\\>|\\<)"
         },
```

```
        { "name": "invalid.ssraw",
          "match": "(\\$|\\>|\\<)"
        }
    ],
    "uuid": "ca03e751-04ef-4330-9a6b-9b99aae1c418"
}
```

There are more available constructs and code reuse techniques, but the above explanations should get you started with the creation of syntax definitions.

5.6 Plugins

See also:

API Reference More information on the Python API.

Plugins Reference More information about plugins.

Sublime Text 2 is programmable with Python scripts. Plugins reuse existing commands or create new ones to build a feature. Plugins are rather a logical entity than a physical one.

5.6.1 Prerequisites

In order to write plugins, you must be able to program in Python.

5.6.2 Where to Store Plugins

Sublime Text 2 will look for plugins in these places:

- `Packages`
- `Packages/<pkg_name>/`

Consequently, any plugin nested deeper in `Packages` won't be loaded.

Keeping plugins right under `Packages` is discouraged, because Sublime Text sorts packages in a predefined way before loading them. Thus, you might get confusing results if your plugins live outside of a package.

5.6.3 Your First Plugin

Let's write a "Hello, World!" plugin for Sublime Text 2:

1. Select **Tools | New Plugin...** in the menu.
2. Save to `Packages/User/hello_world.py`.

You've just written your first plugin. Let's put it to use:

1. Create a new buffer (`Ctrl+n`).
2. Open the python console (`Ctrl+``).
3. Type: `view.run_command("example")` and press enter.

You should see the text "Hello, World!" in your new buffer.

5.6.4 Analyzing Your First Plugin

The plugin created in the previous section should look roughly like this:

```
import sublime, sublime_plugin

class ExampleCommand(sublime_plugin.TextCommand):
    def run(self, edit):
        self.view.insert(edit, 0, "Hello, World!")
```

The `sublime` and `sublime_plugin` modules are both provided by Sublime Text 2.

New commands derive from the `*Command` classes defined in `sublime_plugin` (more on this later).

The rest of the code is concerned with particulars of the `TextCommand` or the API that we'll discuss in the next sections.

Before moving on, though, we'll look at how we called the new command: We first opened the python console, and then issued a call to `view.run_command()`. This is a rather inconvenient way of using plugins, but it's often useful when you're in the development phase. For now, keep in mind that your commands can be accessed through key bindings or other means, just as other commands are.

Conventions for Command Names

You might have noticed that our command is defined with the name `ExampleCommand`, but we pass the string `example` to the API call instead. This is necessary because Sublime Text 2 normalizes command names by stripping the `Command` suffix and separating `CamelCasedPhrases` with underscores, like this: `camel_cased_phrases`.

New commands should follow the pattern mentioned above for class names.

5.6.5 Types of Commands

You can create the following types of commands:

- Application commands (`ApplicationCommand`)
- Window commands (`WindowCommand`)
- Text commands (`TextCommand`)

When writing plugins, consider your goal and choose the appropriate type of commands for your plugin.

Shared Traits of Commands

All commands need to implement a `.run()` method in order to work. Additionally, they can receive and arbitrarily long number of keyword parameters.

Application Commands

Application commands derive from `sublime_plugin.ApplicationCommand`. Due to the status of the API at the time of this writing, we won't discuss application commands any further at the moment.

Window Commands

Window commands operate at the window level. This doesn't mean that you cannot manipulate views from window commands, but rather that you don't need views to exist in order for window commands to be available. For instance, the built-in command `new_file` is defined as a `WindowCommand` so it works too when no view is open. Requiring a view to exisit in that case wouln't make sense.

Window command instances have a `.window` attribute pointing to the window instance that created them.

Text Commands

Text commands operate at the buffer level and they require a buffer to exist in order to be available.

View command instances have a `.view` attribute pointing to the view instance that created them.

Text Commands and the `edit` Object

The edit object groups modifications to the view so undo and macros work in a sensible way. You are responsible for creating and closing edit objects. To do so, you can call `view.begin_edit()` and `edit.end_edit()`. Text commands get passed an open `edit` object in their `run` method for convenience. Additionally, many `View` methods require an edit object.

Responding to Events

Any command deriving from `EventListener` will be able to respond to events.

Another Plugin Example: Feeding the Completions List

Let's create a plugin that fetches data from Google Autocomplete service and feeds it to Sublime Text 2 completions list. Please note that as ideas for plugins go, this a very bad one.

```
import sublime, sublime_plugin

from xml.etree import ElementTree as ET
from urllib import urlopen

GOOGLE_AC = r"http://google.com/complete/search?output=toolbar&q=%s"

class GoogleAutocomplete(sublime_plugin.EventListener):
    def on_query_completions(self, view, prefix, locations):
        elements = ET.parse(
                        urlopen(GOOGLE_AC % prefix)
                    ).getroot().findall("./CompleteSuggestion/suggestion")

        sugs = [(x.attrib["data"],) * 2 for x in elements]

        return sugs
```

Note: Please make sure you don't keep this plugin around after trying it. It will interefere with the autocompletions look-up chain.

5.6.6 Learning the API

In order to create plugins, you need to get acquainted with the Python API Sublime Text 2 exposes, and the available commands. Documentation on both is scarce at the time of this writing, but you can read existing code and learn from it too. In particular, the `Packages/Default` folder contains many examples of undocumented commands and API calls.

5.7 Packages

Packages are simply directories under `Packages`. They exist mainly for organizational purposes, but Sublime Text follows a few rules when dealing with them. More on this later.

Here's a list of typical resources living inside packages:

- build systems (`.sublime-build`)
- key maps (`.sublime-keymap`)
- macros (`.sublime-macro`)
- menus (`.sublime-menu`)
- plugins (`.py`)
- preferences (`.tmPreferences`)
- settings (`.sublime-settings`)
- syntax definitions (`.tmLanguage`)
- snippets (`.sublime-snippet`)
- themes (`.sublime-theme`)

Some packages may include support files for other packages or core features. For example, the spell checker uses `PackagesLanguage - English` as a data store for English dictionaries.

5.7.1 Types of Packages

In order to talk about packages in this guide, we'll divide them in groups. This division is artificial and for the sake of clarity in this topic. Sublime Text doesn't use it in any way.

core packages Sublime Text requires these packages in order to work.

shipped packages Sublime Text includes these packages in every installation, although they are not technically required. Shipped packages enhance Sublime Text out of the box. They may have been contributed by users or third parties.

user packages These packages are installed by the user to further extend Sublime Text. They are not part of any Sublime Text installation and are always contributed by users or third parties.

installed packages Any package that Sublime Text can restore if deleted.

Let's emphasize again that you don't need to memorize this classification. Also, it's worth noting that by *third party* we mainly refer to other editors' users, like Textmate's.

5.7.2 Installation of Packages

There are two main ways to install packages:

- `.sublime-package` files
- version control systems

Ultimately, installing a package consists simply in placing a directory containing Sublime Text resources under `Packages`. The only thing that changes from one system to another is how you copy these files.

Installing Packages vs Installed Packages

Note that installing a package doesn't actually make that package an installed package. *Installed packages* are `.sublime-package` files residing in the `Installed Packages` directory. In this guide, we use *to install a package* to mean to copy a package to `Packages`.
Sublime Text can restore any package located in `Installed Packages`, but not every package located in `Packages`.

Installation of `.sublime-package` Files

Copy the `.sublime-package` file to the `Installed Packages` directory and restart Sublime Text. If the `Installed Packages` doesn't exist, you can create it.

Note that `.sublime-package` files are simply `.zip` archives with a custom file extension.

Installation of Packages from a Version Control System

Explaining how to use version control systems (VCSs) is outside the scope of this guide, but there are many user packages available for free on public repositories like Google Code, GitHub and Bitbucket.

Also, there is a Sublime Text organization at GitHub open to contributors.

5.7.3 Packages and Magic

There's little invisible magic involved in the way Sublime Text deals with packages. Two notable exceptions are that macros defined in any package appear under **Tools | Macros | <Your Package>**, and snippets from any package appear under **Tools | Snippets | <Your Package>**.

As mentioned at the beginning, however, there are some rules for packages. For instance, `Package/User` will never be clobbered during updates of the software.

The `User` Package

Usually, unpackaged resources are stored in `Packages/User`. If you have a few loose snippets, macros or plugins, this is a good place to keep them.

Merging and Order of Precedence

`Packages/Default` and `Packages/User` also receive a special treatment when merging files (e. g. `.sublime-keymap` and `.sublime-settings` files). Before the merging can take place, the files have to be

arranged in an order. To that end, Sublime Text sorts them alphabetically by name with the exception of files contained in `Default` and `User`: `Default` will always go to the front of the list, and `User` to the end.

5.7.4 Restoring Packages

Sublime Text keeps a copy of all installed packages so it can recreate them when needed. This means it will be able to reinstall core packages, shipped packages and user packages alike. However, only user packages installed as a `sublime-package` are added to the registry of installed packages. Packages installed in alternative ways will be completely lost if you delete them.

Reverting Sublime Text to Its Default Configuration

To revert Sublime Text to its default configuration, delete the data directory and restart the editor. Keep in mind, though, that the `Installed Packages` directory will be deleted too, so you will lose all installed packages.

Always make sure to back up your data before taking an extreme measure like this one.

5.7.5 The `Installed Packages` Directory

You will find this directory in the data directory. It contains a copy of every `sublime-package` installed. Used to restore `Packages`.

5.7.6 The `Pristine Packages` Directory

You will find this directoy in the data directory. It contains a copy of every shipped and core package. Used to restore `Packages`.

CHAPTER 6

FAQ

See also:

Sublime Text 1 FAQ Sublime Text 1 FAQ.

Sublime Text 2 FAQ Sublime Text 2 FAQ.

CHAPTER 7

Reference

In this section you will find concise information about many aspects of Sublime Text.

If you're looking for a slow-paced introduction to any of these topics, try the general index.

7.1 Snippets

7.1.1 Compatibility with Textmate

Sublime Text snippets are generally compatible with Textmate snippets.

7.1.2 File Format

Snippet files are XML files with the `sublime-snippet` extension.

```
<snippet>
    <content><![CDATA[]]></content>
    <tabTrigger></tabTrigger>
    <scope></scope>
    <description></description>
</snippet>
```

content Actual snippet content.

tabTrigger Implicit keybinding for this snippet. Last key (implicit) is `TAB`.

scope Scope selector to activate this snippet.

description User friendly description for menu item.

7.1.3 Escape Sequences

\$ Literal `$`.

7.1.4 Environment Variables

`$PARAM1 .. $PARAMn`	Arguments passed to the `insertSnippet` command.
`$SELECTION`	The text that was selected when the snippet was triggered.
`$TM_CURRENT_LINE`	Content of the line the cursor was in when the snippet was triggered.
`$TM_CURRENT_WORD`	Current word under the cursor when the snippet was triggered.
`$TM_FILENAME`	File name of the file being edited including extension.
`$TM_FILEPATH`	File path to the file being edited.
`$TM_FULLNAME`	User's user name.
`$TM_LINE_INDEX`	Column the snippet is being inserted at, 0 based.
`$TM_LINE_NUMBER`	Row the snippet is being inserted at, 1 based.
`$TM_SELECTED_TEXT`	An alias for `$SELECTION`.
`$TM_SOFT_TABS`	`YES` if `translateTabsToSpaces` is true, otherwise `NO`.
`$TM_TAB_SIZE`	Spaces per-tab (controlled by the `tabSize` option).

7.1.5 Fields

Mark positions to cycle through by pressing `TAB` or `SHIFT + TAB`.

Syntax: `$1` .. `$n`

`$0` Exit mark. Position at which normal text editing should be resumed. By default, Sublime Text implicitly sets this mark at the end of the snippet's `content` element.

Fields with the same name mirror each other.

7.1.6 Place Holders

Fields with a default value.

Syntax: `${1:PLACE_HOLDER}` .. `${n:PLACE_HOLDER}`

Fields and place holders can be combined and nested within other place holders.

7.1.7 Substitutions

Syntax:

- `${var_name/regex/format_string/}`
- `${var_name/regex/format_string/options}`

`var_name` The field's name to base the substitution on: 1, 2, 3...

`regex` Perl-style regular expression: See the Boost library documentation for regular expressions.

`format_string` See the Boost library documentation for format strings.

`options`

Optional. Any of the following:

`i` Case-insensitive regex.

`g` Replace all occurrences of `regex`.

`m` Don't ignore newlines in the string.

7.2 Syntax Definitions

Warning: This topic is a draft and may contain wrong information.

7.2.1 Compatibility with Textmate

Sublime Text syntax definitions are generally compatible with Textmate language files.

7.2.2 File Format

Syntax definitions are files in the Plist format with the `tmLanguage` extension. In this reference files, however, JSON is used instead and converted into Plist.

```
{ "name": "Sublime Snippet (Raw)",
  "scopeName": "source.ssraw",
  "fileTypes": ["ssraw"],
  "patterns": [
      { "match": "\\$(\\d+)",
        "name": "keyword.ssraw",
        "captures": {
            "1": { "name": "constant.numeric.ssraw" }
         },
        "comment": "Tab stops like $1, $2..."
      },
      { "match": "\\$([A-Za-z][A-Za-z0-9_]+)",
        "name": "keyword.ssraw",
        "captures": {
            "1": { "name": "constant.numeric.ssraw" }
         },
        "comment": "Variables like $PARAM1, $TM_SELECTION..."
      },
      { "name": "variable.complex.ssraw",
        "begin": "(\\$)(\\{)([0-9]+):",
        "beginCaptures": {
            "1": { "name": "keyword.control.ssraw" },
            "3": { "name": "constant.numeric.ssraw" }
         },
         "patterns": [
            { "include": "$self" },
            { "name": "string.ssraw",
              "match": "."
            }
         ],
         "end": "\\}"
      },
      { "name": "constant.character.escape.ssraw",
        "match": "\\\\(\\$|\\>|\\<)"
      },
      { "name": "invalid.ssraw",
        "match": "(\\$|\\>|\\<)"
      }
  ],
  "uuid": "ca03e751-04ef-4330-9a6b-9b99aae1c418"
}
```

name Descriptive name for the syntax definition. Shows up in the syntax definition dropdown menu located in the bottom right of Sublime Text interface. It's usually the name of the programming language or equivalent.

scopeName Name of the top-level scope for this syntax definition. Either `source.<lang>` or `text.<lang>`. Use `source` for programming languages and `text` for everything else.

fileTypes And array of file type extensions for which this syntax should be automatically activated. Include the extensions without the leading dot.

uuid Unique indentifier for this syntax definition. Currently ignored.

foldingStartMarker Currently ignored. Used for code folding.

foldingStopMarker Currently ignored. Used for code folding.

patterns Array of patterns to match against the buffer's text.

repository Array of patterns abstracted out from the patterns element. Useful to keep the syntax definition tidy as well as for specialized uses like recursive patterns. Optional.

7.2.3 The Patterns Array

Elements contained in the `patterns` array.

match Contains the following elements:

`match`	Pattern to search for.
`name`	Scope name to be assigned to matches of `match`.
`comment`	Optional. For information only.
`captures`	Optional. Refinement of `match`. See below.

In turn, `captures` can contain *n* of the following pairs of elements:

`0..n`	Name of the group referenced.
`name`	Scope to be assigned to the group.

Examples:

```
// Simple

{ "name": "constant.character.escape.ssraw",
  "match": "\\\\(\\$|\\>|\\<)"
  "comment". "Sequences like \$, \> and \<"
}

// With captures

{ "match": "\\$(\\d+)",
  "name": "keyword.ssraw",
  "captures": {
      "1": { "name": "constant.numeric.ssraw" }
   },
  "comment": "Tab stops like $1, $2..."
}
```

include Includes items in the repository, other syntax definitions or the current one.

References:

$self	The current syntax definition.
#itemName	itemName in the repository.
source.js	External syntax definitions.

Examples:

```
// Requires presence of DoubleQuotedStrings element in the repository.
{ "include": "#DoubleQuotedStrings" }

// Recursively includes the current syntax definition.
{ "include": "$self" }

// Includes and external syntax definition.
{ "include": "source.js" }
```

`begin..end` Defines a scope potentially spanning multiple lines

Contains the following elements:

`begin`	The start marker pattern.
`end`	The end marker pattern.
`name`	Scope name for the whole region.
`beginCaptures`	`captures` for `begin`. See `captures`.
`endCaptures`	`captures` for `end`. See `captures`.
`patterns`	`patterns` to be matched against the content.
`contentName`	Scope name for the content excluding the markers.

Example:

```
{ "name": "variable.complex.ssraw",
  "begin": "(\\$)(\\{)([0-9]+):",
  "beginCaptures": {
      "1": { "name": "keyword.control.ssraw" },
      "3": { "name": "constant.numeric.ssraw" }
   },
   "patterns": [
      { "include": "$self" },
      { "name": "string.ssraw",
        "match": "."
      }
   ],
   "end": "\\}"
}
```

7.2.4 Repository

Can be referenced from `patterns` or from itself in an `include` element. See `include` for more information.

The repository can contain the following elements:

- Simple elements:

```
"elementName": {
  "match":  "some regexp",
  "name":   "some.scope.somelang"
}
```

- Complex elements:

```
"elementName": {
  "patterns": [
    { "match":  "some regexp",
      "name":   "some.scope.somelang"
```

```
        },
        { "match":  "other regexp",
          "name":   "some.other.scope.somelang"
        }
      ]
    }
```

Examples:

```
"repository": {
  "numericConstant": {
    "patterns": [
      { "match":  "\\d*(?<!\\.)(\\.)\\d+(d)?(mb|kb|gb)?",
        "name":   "constant.numeric.double.powershell",
        "captures": {
          "1": { "name": "support.constant.powershell" },
          "2": { "name": "support.constant.powershell" },
          "3": { "name": "keyword.other.powershell" }
        }
      },
      { "match":  "(?<!\\w)\\d+(d)?(mb|kb|gb)?(?!\\w)",
        "name":   "constant.numeric.powershell",
        "captures": {
          "1": { "name": "support.constant.powershell" },
          "2": { "name": "keyword.other.powershell" }
        }
      }
    ]
  },
  "scriptblock": {
    "begin":  "\\{",
    "end":    "\\}",
    "name":   "meta.scriptblock.powershell",
    "patterns": [
      { "include": "$self" }
    ]
  },
}
```

7.2.5 Escape Sequences

Be sure to escape JSON/XML sequences as needed.

7.3 Build Systems

Build systems run external programs to process your files, and print their output to the output panel. Ultimately, they call `subprocess.Popen`.

Essentially, build systems are configuration data for an external program. In them, you specify the switches, options and environment information you want passed to the executable.

Optionally, you can override the default *middleman* between the configuration data and the external program. For example, you could implement the build system entirely in a Sublime Text plugin, without ever calling an external program. You can do this because this *middleman* is simply a Sublime Text plugin, implemented in `Packages/Default/exec.py`.

7.3.1 File Format

Build systems use JSON. Here's an example:

```
{
    "cmd": ["python", "-u", "$file"],
    "file_regex": "^[ ]*File \"(...*?)\", line ([0-9]*)",
    "selector": "source.python"
}
```

7.3.2 Options

cmd Array containing the command to run and its desired arguments. If you don't specify an absolute path, the external program must be in your `PATH`, one of your system's environmental variables.

> **Note:** Under Windows, GUIs are supressed.

file_regex Optional. Regular expression (Perl-style) to capture error output of `cmd`. See next section for more details.

line_regex Optional. If `file_regex` doesn't match on the current line, but `line_regex` is specified, and it does match on the current line, then walk backwards through the buffer until a line matching `file regex` is found, and use these two matches to determine the file and line to go to.

selector Optional. Used when **Tools | Build System | Automatic** is set to `true`. Sublime Text uses this scope selector to find the appropriate build system for the active view automatically.

working_dir Optional. Directory to change the current directory to before running `cmd`. The original current directory is restored afterwards.

encoding Optional. Output encoding of `cmd`. Must be a valid python encoding. Defaults to `UTF-8`.

target Optional. Sublime Text command to run. Defaults to `exec` (`Packages/Default/exec.py`). It receives the configuration data specified in the `.build-system` file.

env Optional. Dictionary of environment variables to be merged with the current process' that will be passed to `cmd`.

shell Optional. If `true`, `cmd` will be run through the shell (`cmd.exe`, `bash...`).

path Optional. This string will replace the current process' `PATH` before calling `cmd`. The old `PATH` value will be restored after that. It's useful to add directories to `PATH` that you don't have or want in your regular `PATH` variable everywhere.

Capturing Error Output with `file_regex`

The `file_regex` option uses a Perl-style regular expression to capture up to four fields of error information from the build program's output, namely: *file name*, *line number*, *column number* and *error message*. Use groups in the pattern to capture this information. The *file name* field and the *line number* field are required.

When error information is captured, you can navigate to error instances in your project's files with `F4` and `Shift+F4`. If available, the captured *error message* will be displayed in the status bar.

Platform-specific Options

`windows`, `osx` and `linux` are additional options to provide platform-specific data. Here's an example:

```
{
    "cmd": ["ant"],
    "file_regex": "^ *\\[javac\\] (.+):([0-9]+):() (.*)$",
    "working_dir": "${project_path:${folder}}",
    "selector": "source.java",

    "windows":
    {
        "cmd": ["ant.bat"]
    }
}
```

In this case, `ant` will be executed for every platform except Windows, where `ant.bat` will be used instead.

7.3.3 Variables

Build systems expand the following variables:

`$file`	The full path to the current file, e. g., `C:\Files\Chapter1.txt`.
`$file_path`	The directory of the current file, e. g., `C:\Files`.
`$file_name`	The name portion of the current file, e. g., `Chapter1.txt`.
`$file_extension`	The extension portion of the current file, e. g., `txt`.
`$file_base_name`	The name only portion of the current file, e. g., `Document`.
`$packages`	The full path to the `Packages` folder.
`$project`	The full path to the current project file.
`$project_path`	The directory of the current project file.
`$project_name`	The name portion of the current project file.
`$project_extension`	The extension portion of the current project file.
`$project_base_name`	The name only portion of the current project file.

Place Holders for Variables

Features found in snippets can be used with these variables. For example:

```
${project_name:Default}
```

This will emit the name of the current project if there is one, otherwise *Default*.

```
${file/\.php/\.txt/}
```

This will emit the full path of the current file, replacing *.php* with *.txt*.

7.3.4 Running Build Systems

Select the desired build system from **Tools | Build System**, and then select **Tools | Build** or press `F7`.

7.3.5 Troubleshooting Build Systems

External programs used in build systems need to be in your `PATH`. As a quick test, you can try to run them from the command line first and see whether they work. Note, however, that your shell's `PATH` variable might differ to that seen by Sublime Text due to your shell's profile. Remember that you can use the `path` option in a `.build-system` file to add directories to `PATH` without changing your system's settings.

See also:

Managing Environment Variables in Windows Search Microsoft for this topic.

7.4 Key Bindings

Key bindings map key presses to commands.

7.4.1 File Format

Key bindings are stored in `.sublime-keymap` files and defined in JSON. All key map file names need to follow this pattern: `Default (<platform>).sublime-keymap`. Otherwise, Sublime Text will ignore them.

Platform-Specific Key Maps

Each platform gets its own key map:

- `Default (Windows).sublime-keymap`
- `Default (OSX).sublime-keymap`
- `Default (Linux).sublime-keymap`

Separate key maps exist to abide by different vendor-specific HCI guidelines.

Structure of a Key Binding

Key maps are arrays of key bindings. Below you'll find valid elements in key bindings.

`keys` An array of case-sensitive keys to be pressed. Modifiers can be specified with the + sign. Chords are built by adding elements to the array, e. g. `["ctrl+k","ctrl+j"]`. Ambiguous chords are resolved with a timeout.

`command` Name of the command to be executed.

`args` Dictionary of arguments to be passed to `command`. Keys must be the names of parameters to `command`.

`context` Array of contexts to selectively enable the key binding. All contexts must be true for the key binding to trigger. See *Structure of a Context* below.

Here's an example illustrating most of the features outlined above:

```
{ "keys": ["shift+enter"], "command": "insert_snippet", "args": {"contents": "\n\t$0\n"}, "context":
        [
                { "key": "setting.auto_indent", "operator": "equal", "operand": true },
                { "key": "selection_empty", "operator": "equal", "operand": true, "match_all": true
                { "key": "preceding_text", "operator": "regex_contains", "operand": "\\{$", "match_a
                { "key": "following_text", "operator": "regex_contains", "operand": "^\\}", "match_a
        ]
}
```

Structure of a Context

`key` Name of a context operand to query.

`operator` Type of test to perform against `key`.

operand Value against which the result of `key` is tested.

match_all Requires the test to succeed for all selections. Defaults to `false`.

Context Operands

auto_complete_visible Returns `true` if the autocomplete list is visible.

has_next_field Returns `true` if there's a next snippet field available.

has_prev_field Returns `true` if there's a previous snippet field available.

num_selections Returns the number of selections.

overlay_visible Returns `true` if any overlay is visible.

panel_visible Returns `true` if any panel is visible.

following_text Restricts the test to the text following the caret.

preceding_text Restricts the test to the text preceding the caret.

selection_empty Returns `true` if the selection is an empty region.

setting.x Returns the value of the `x` setting. `x` can be any string.

text Restricts the test to the line the caret is in.

selector Returns the current scope.

Context Operators

equal, not_equal Test for equality.

regex_match, not_regex_match Match against a regular expression.

regex_contains, not_regex_contains Match against a regular expression (containment).

7.4.2 Command Mode

Sublime Text provides a `command_mode` setting to prevent key presses from being sent to the buffer. This is useful to emulate Vim's modal behavior.

7.4.3 Bindable Keys

Keys may be specified literally or by name. Below you'll find the list of valid names:

- `up`
- `down`
- `right`
- `left`
- `insert`
- `home`
- `end`

- pageup
- pagedown
- backspace
- delete
- tab
- enter
- pause
- escape
- space
- keypad0
- keypad1
- keypad2
- keypad3
- keypad4
- keypad5
- keypad6
- keypad7
- keypad8
- keypad9
- keypad_period
- keypad_divide
- keypad_multiply
- keypad_minus
- keypad_plus
- keypad_enter
- clear
- f1
- f2
- f3
- f4
- f5
- f6
- f7
- f8
- f9
- f10

- f11
- f12
- f13
- f14
- f15
- f16
- f17
- f18
- f19
- f20
- sysreq
- break
- context_menu
- browser_back
- browser_forward
- browser_refresh
- browser_stop
- browser_search
- browser_favorites
- browser_home

Modifiers

- shift
- ctrl
- alt
- super (Windows key, Command key...)

Warning about Bindable Keys

- Ctrl+Alt+<alphanum> should not be used on any Windows key bindings.
- Option+<alphanum> should not be used on any OS X key bindings.

In both cases, the users ability to insert non-ascii characters would be compromised.

7.4.4 Keeping Key Maps Organized

Sublime Text ships with default key maps under Packages/Default. Other packages may include their own key map files. The recommended storage location for your personal key map is Packages/User.

See *Merging and Order of Precedence* for information about how Sublime Text sorts files for merging.

7.4.5 International Keyboards

Due to the way Sublime Text maps key names to physical keys, there might be a mismatch between the two.

7.4.6 Troubleshooting

See sublime.log_commands(flag) to enable command logging. It may help when debugging key maps.

7.5 Settings (Reference)

7.5.1 Global Settings

Target file: `Global.sublime-settings`.

`theme` Theme to be used. Accepts a file base name (e. g.: `Default.sublime-theme`).

`remember_open_files` Determines whether to reopen the buffers that were open when Sublime Text was last closed.

`folder_exclude_patterns` Excludes the matching folders from the side bar, GoTo Anything, etc.

`file_exclude_patterns` Excludes the matching files from the side bar, GoTo Anything, etc.

`scroll_speed` Set to `0` to disable smooth scrolling. Set to a value between `0` and `1` to scroll slower, or set to a value larger than `1` to scroll faster.

`show_tab_close_buttons` If `false`, hides the tabs' close buttons until the mouse is hovered over the tab.

`mouse_wheel_switches_tabs` If `true`, scrolling the mouse wheel will cause tabs to switch if the cursor is in the tab area.

`open_files_in_new_window` OS X only. When filters are opened from Finder, or by dragging onto the dock icon, this controls if a new window is created or not.

7.5.2 File Settings

Target files: `Base File.sublime-settings`, `<file_type>.sublime-settings`.

Whitespace and Indentation

`auto_indent` Toggles automatic indentation.

`tab_size` Number of spaces a tab is considered to be equal to.

`translate_tabs_to_spaces` Determines whether to replace a tab character with `tab_size` number of spaces when `Tab` is pressed.

`use_tab_stops` If `translate_tabs_to_spaces` is `true`, will make `Tab` and `Backspace` insert/delete `tab_size` number of spaces per key press.

`trim_automatic_white_space` Toggles deletion of white space added by `auto_indent`.

`detect_indentation` Set to `false` to disable detection of tabs vs. spaces whenever a buffer is loaded. If set to `true`, it will automatically modify `translate_tabs_to_spaces` and `tab_size`.

`draw_white_space` Valid values: `none`, `selection`, `all`.

trim_trailing_white_space_on_save Set to `true` to remove white space on save.

Visual Settings

color_scheme Sets the colors used for text highlighting. Accepts a path rooted at the data directory (e. g.: `Packages/Color Scheme - Default/Monokai Bright.tmTheme`).

font_face Font face to be used for editable text.

font_size Size of the font for editable text.

font_options Valid values: `bold`, `italic`, `no_antialias`, `gray_antialias`, `subpixel_antialias`, `directwrite` (Windows).

gutter Toggles display of gutter.

rulers Columns in which to display vertical rules. Accepts a list of numeric values (e. g. `[79, 89, 99]` or a single numeric value (e. g. `79`).

draw_minimap_border Set to `true` to draw a border around the minimap's region corresponding to the the view's currently visible text. The active color scheme's `minimapBorder` key controls the border's color.

highlight_line Set to `false` to stop highlighting lines with a cursor.

line_padding_top Additional spacing at the top of each line, in pixels.

line_padding_bottom Additional spacing at the bottom of each line, in pixels.

scroll_past_end Set to `false` to disable scrolling past the end of the buffer. If `true`, Sublime Text will leave a wide, empty margin between the last line and the bottom of the window.

line_numbers Toggles display of line numbers in the gutter.

word_wrap If set to `false`, long lines will be clipped instead of wrapped. Scroll the screen horizontally to see the clipped text.

wrap_width If greater than `0`, wraps long lines at the specified column as opposed to the window width. Only takes effect if `wrap_width` is set to `true`.

indent_subsequent_lines If set to `false`, wrapped lines will not be indented. Only takes effect if `wrap_width` is set to `true`.

draw_centered If set to `true`, text will be drawn centered rather than left-aligned.

match_brackets Set to `false` to disable underlining the brackets surrounding the cursor.

match_brackets_content Set to `false` is you'd rather only highlight the brackets when the cursor is next to one.

match_brackets_square Set to `false` to stop highlighting square brackets. Only takes effect if `match_brackets` is `true`.

match_bracktets_braces Set to `false` to stop highlighting curly brackets. Only takes effect if `match_brackets` is `true`.

match_bracktets_angle Set to `false` to stop highlighting angle brackets. Only takes effect if `match_brackets` is `true`.

Automatic Behavior

auto_match_enabled Toggles automatic pairing of quotes, brackets, etc.

save_on_focus_lost Set to true to automatically save files when switching to a different file or application.

find_selected_text If `true`, the selected text will be copied into the find panel when it's shown.

word_separators Characters considered to separate words in actions like advancing the cursor, etc. They are not used in all contexts where a notion of a word separator is useful (e. g.: word wrapping). In such other contexts, the text might be tokenized based on other criteria (e. g. the syntax definition rules).

ensure_newline_at_eof_on_save Always adds a new line at the end of the file if not present when saving.

System and Miscellaneous Settings

is_widget Returns `true` if the buffer is an input field in a dialog as opposed to a regular buffer.

spell_check Toggles the spell checker.

dictionary Word list to be used by the spell checker. Accepts a path rooted at the data directory (e. g.: `Packages/Language - English/en_US.dic`). You can add more dictionaries.

fallback_encoding The encoding to use when the encoding can't be determined automatically. ASCII, UTF-8 and UTF-16 encodings will be automatically detected.

default_line_ending Determines what characters to use to designate new lines. Valid values: `system` (OS-dependant), `windows` (CRLF) and `unix` (LF).

tab_completion Determines whether pressing `Tab` will insert completions.

Build and Error Navigation Settings

result_file_regex Regular expression used to extract error information from some output dumped into a view or output panel. Follows the same rules as error capturing in build systems.º

result_line_regex Regular expression used to extract error information from some output dumpºed into a view or output panel. Follows the same rules as error capturing in build systems.

result_base_dir Directory to start looking for offending files in based on information extracted with `result_file_regex` and `result_line_regex`.

build_env List of paths to add to build systems by default.

File and Directory Settings

default_dir Sets the default save directory for the view.

Input Settings

command_mode If set to `true`, the buffer will ignore key strokes. Useful to emulate Vim.

7.6 Command Palette

The command palette is fed entries with `.sublime-commands` files.

7.6.1 File Format (`.sublime-commands` Files)

Here's an excerpt from `Packages/Default/Default.sublime-commands`:

```
[
    { "caption": "Project: Save As", "command": "save_project_as" },
    { "caption": "Project: Close", "command": "close_project" },
    { "caption": "Project: Add Folder", "command": "prompt_add_folder" },

    { "caption": "Preferences: Default File Settings", "command": "open_file", "args": {"file": "${pa
    { "caption": "Preferences: User File Settings", "command": "open_file", "args": {"file": "${packa
    { "caption": "Preferences: Default Global Settings", "command": "open_file", "args": {"file": "$
    { "caption": "Preferences: User Global Settings", "command": "open_file", "args": {"file": "${pac
    { "caption": "Preferences: Browse Packages", "command": "open_dir", "args": {"dir": "$packages"}
]
```

`caption` Text for display in the command palette.

`command` Command to be executed.

`args` Arguments to pass to `command`. Note that to locate the packages folder you need to use a snippet-like variable: `${packages}` or $packages. This different to other areas of the editor due to different implementations of the lower level layers.

7.6.2 How to Use the Command Palette

1. Press `Ctrl+Shift+P`
2. Select command

Entries are filtered by current context. Not all entries will be visible at all times.

7.7 Completions

Completions provide an IDE-like functionality to insert dynamic content through the completions list or by pressing `Tab`.

7.7.1 File Format

Completions are JSON files with the `.sublime-completions` extension.

7.7.2 Structure of a Completions List

`scope` Determines whether the completions are to be sourced from this file. See *Scopes* for more information.

`completions` Array of completions.

Here's an excerpt from the html completions:

```
{
        "scope": "text.html - source - meta.tag, punctuation.definition.tag.begin",

        "completions":
        [
                { "trigger": "a", "contents": "<a href=\"$1\">$0</a>" },
```

```
                { "trigger": "abbr", "contents": "<abbr>$0</abbr>" },
                { "trigger": "acronym", "contents": "<acronym>$0</acronym>" }

        ]
}
```

7.7.3 Types of Completions

Plain Strings

Plain strings are equivalent to an entry where the `trigger` is identical to the `contents`:

```
"foo"

# is equivalent to:

{ "trigger": "foo", "contents": "foo" }
```

Trigger-based Completions

`trigger` Text that will be displayed in the completions list and will cause the `contents` to be inserted when validated.

`contents` Text to be inserted in the buffer. Can use snippet features.

7.7.4 Sources for Completions

These are the sources for completions the user can control:

- `.sublime-completions`
- `EventListener.on_query_completions()`

Additionally, other completions are folded into the final list:

- Snippets
- Words in the buffer

Priority of Sources for Completions

- Snippets
- API-injected completions
- `.sublime-completions` files
- Words in buffer

Snippets will only be automatically completed against an exact match of their tab trigger. Other sources for completions are filtered with a case insensitve fuzzy search instead.

7.7.5 The Completions List

To use the completions list:

- Press `Ctrl+spacebar` to open
- Optionally, press `Ctrl+spacebar` again to select next entry
- Press `Enter` or `Tab` to validate selection

Note: The current selection in the completions list can actually be validated with any punctuation sign that isn't itself bound to a snippet.

Snippets show up in the completions list following the pattern: `<tab_trigger> : <name>`. For the other completions, you will just see the text to be inserted.

If the list of completions can be narrowed down to one choice, the autocomplete dialog will be bypassed and the corresponding content will be inserted straight away according to the priority rules stated above.

7.7.6 Enabling and Disabling Tab Completion for Completions

The `tab_completion` setting is `true` by default. Set it to `false` if you want `Tab` to stop sourcing the most likely completion. This setting has no effect on triggers defined in `.sublime-snippet` files, so snippets will always be inserted after a `Tab`.

With `tab_completion` on, The same order of priority as stated above applies, but, unlike in the case of the completions list, Sublime Text will always insert a completion, even if faced with an ambiguous choice.

Inserting a Literal Tab

If `tab_completion` is `true`, you can press `Shift+Tab` after a prefix to insert a literal tab character.

7.8 Plugins

See also:

API Reference More information on the Python API.

Plugins are Python scripts implementing `*Command` classes from `sublime_plugin`.

7.8.1 Where to Store Plugins

Sublime Text 2 will look for plugins in these places:

- `Package`
- `Packages/<pkg_name>`

Any plugin nested deeper in `Packages` won't be loaded.

7.8.2 Conventions for Command Names

Sublime Text 2 command class names are suffixed by convention with `Command` and written as `CamelCasedPhrases`.

However, Sublime Text 2 transforms the class name from `CamelCasedPhrases` to `camel_cased_phrases`. So `ExampleCommand` would turn into `example` and `AnotherExampleCommand` would turn into `another_example`.

For class definition names, use `CamelCasedPhrasesCommand`; to call a command from the API, use the transformed name (`camel_cased_phrases`).

7.8.3 Types of Commadns

- `sublime_plugin.ApplicationCommand`
- `sublime_plugin.WindowCommand`
- `sublime_plugin.TextCommand`
- `sublime_plugin.EventListener`

Window command instances have a `.window` attribute pointing to the window instance that created them. Similarly, view command instances have a `.view` attribute.

Shared Traits for Commands

All commands must implement a `.run()` method. All commands can receive and arbitrarily long number of keyword arguments.

7.8.4 How to Call Commands from the API

Use a reference to a `View`, a `Window` or `sublime` depending on the type of command, and call `object.run_command('command_name')`. In addition, you can pass a dictionary where keys are names of parameters to `command_name`.

```
window.run_command("echo", {"Tempus": "Irreparabile", "Fugit": "."})
```

7.8.5 Text Commands and the `edit` Object

The two API functions of interest are `view.begin_edit()`, which takes an optional command name and an optional dictionary of arguments, and `view.end_edit()`, which finishes the edit.

All actions done within an edit are grouped as a single undo action. Callbacks such as `on_modified()` and `on_selection_modified()` are called when the edit is finished.

It's important to call `view.end_edit()` after each `view.begin_edit()`, otherwise the buffer will be in an inconsistent state. An attempt will be made to fix it automatically if the edit object gets collected, but that often doesn't happen when you expect, and will result in a warning printed to the console. In other words, you should always bracket an edit in a `try..finally` block.

The command name passed to `begin_edit()` is used for repeat, macro recording, and for describing the action when undoing/redoing it. If you're making an edit outside of a `TextCommand`, you should almost never supply a command name.

If you have created an edit object, and call a function that creates another one, that's fine: the edit is only considered finished when the outermost call to `end_edit()` runs.

As well as grouping modifications, you can use edit objects for grouping changes to the selection, so they're undone in a single step.

7.8.6 Responding to Events

Any subclass of `EventListener` will be able to respond to events.

A Word of Warning about `EventListener`

Expensive operations in event listeners can cause Sublime Text 2 to become unresponsive, especially in events triggered frequently, like `on_modified` and `on_selection_modified`. Be careful of how much work is done in those and do not implement events you don't need, even if they just `pass`.

7.9 Python API

See also:

Official Documentation API documentation.

7.9.1 Exploring the API

A quick way to see the API in action:

1. Add `Packages\Default` (**Preferences | Browse Packages...**) to your project.
2. `CTRL + SHIFT + F`
3. Enter `*.py` in the **In Files:** field
4. Check `Use Buffer` option
5. Search API name
6. `F4`
7. Study relevant source code