Sublime Text Unofficial Documentation

Release 3.0

guillermooo

October 07, 2014

Contents

CHAPTER 1

About This Documentation

This is the unofficial documentation for the Sublime Text editor.

The sublime what? What are you talking about!?

Sublime Text is a text editor for code and prose. It does away with many repetitive tasks so you can focus on your work. And it's fun to use!

If you're coming here for the first time, we encourage you to read through the *Basic Concepts* section before you continue.

Happy learning!

1.1 Contributing to the Documentation

If you want to contribute to this documentation, head over to the GitHub repo. This guide is created with Sphinx.

CHAPTER 2

Installation

The process of installing Sublime Text is different for each platform.

Make sure to read the conditions for use on the official site. **Sublime Text is not free**.

2.1 32 bits or 64 bits?

Choose the 64-bit version if you're running a 64-bit operating system, otherwise the 32-bit version.

On **Windows**, if in doubt, choose the 32-bit version. Modern 64-bit versions of Windows can run 32-bit software.

On **Linux** run this command in your terminal to check your operating system's type:

```
uname -m
```

For **OS X**, you can ignore this section: there is only one version of Sublime Text for OS X.

2.2 Windows

2.2.1 Portable or Not Portable?

Sublime Text comes in two flavors for Windows: normal, and portable. If you need the portable installation, you probably know already. Otherwise, go with the normal one.

Normal installations separate data between two folders: the installation folder proper, and the *data directory*. These concepts are explained later in this guide. Normal installations also integrate Sublime Text with the Windows context menu.

Portable installations will keep all files Sublime Text needs to run in one single folder. You can then move this folder around and the editor will still work.

2.2.2 How to Install the Normal Version of Sublime Text

Download the installer, doubleclick on it and follow the onscreen instructions.

2.2.3 How to Install the Portable Version of Sublime Text

Download the package and uncompress it to a folder of your choice. You will find the *sublime_text.exe* executable inside that folder.

2.3 OS X

Download and open the *.dmg* file, and then drag the Sublime Text 3 bundle into the *Applications* folder.

To create a *symbolic link* to use at the command line.

```
ln -s  "/Applications/Sublime Text.app/Contents/SharedSupport/bin/subl" /usr/local/bin/subl
```

2.4 Linux

You can download the package and uncompress it manually. Alternatively, you can use the command line.

For i386

```
cd ~
wget http://c758482.r82.cf2.rackcdn.com/sublime-text_build-3047_i386.deb
```

For x64

```
cd ~
wget http://c758482.r82.cf2.rackcdn.com/sublime-text_build-3047_amd64.deb
```

For i386

```
cd ~
wget http://c758482.r82.cf2.rackcdn.com/sublime_text_3_build_3047_x32.tar.bz2
tar vxjf sublime_text_3_build_3047_x32.tar.bz2
```

For x64

```
cd ~
wget http://c758482.r82.cf2.rackcdn.com/sublime_text_3_build_3047_x64.tar.bz2
tar vxjf sublime_text_3_build_3047_x64.tar.bz2
```

Now we should move the uncompressed files to an appropriate location.

```
sudo mv Sublime\ Text\ 3 /opt/
```

Lastly, we create a *symbolic link* to use at the command line.

```
sudo ln -s /opt/Sublime\ Text\ 3/sublime_text /usr/bin/sublime
```

In Ubuntu, if you also want to add Sublime Text to the Unity luncher, read on.

First we need to create a new file.

```
sudo sublime /usr/share/applications/sublime.desktop
```

Then copy the following into it.

```
[Desktop Entry]
Version=3.0
Name=Sublime Text 3
# Only KDE 4 seems to use GenericName, so we reuse the KDE strings.
# From Ubuntu's language-pack-kde-XX-base packages, version 9.04-20090413.
GenericName=Text Editor

Exec=sublime
Terminal=false
```

```
Icon=/opt/Sublime Text 3/Icon/48x48/sublime_text.png
Type=Application
Categories=TextEditor;IDE;Development
X-Ayatana-Desktop-Shortcuts=NewWindow

[NewWindow Shortcut Group]
Name=New Window
Exec=sublime -n
TargetEnvironment=Unity
```

If you've registered your copy of Sublime Text, but every time you open it you're asked to enter your license, you should try running this command.

```
sudo chown -R username:username /home/username/.config /sublime-text-3
```

Just replace *username* with your account's username. This should fix the permission error in the case that you opened up Sublime Text as root when you first entered the license.

2.5 Living Dangerously... or Not

Sublime Text has three release *channels*:

- Stable (default)
- Dev
- Nightly

Furthermore, there are separate channels for the Sublime Text 3 Beta, which is only available to registered users.

- 3-Beta (comparable to *Nightly*)
- 3-Dev

If you are working on a NASA project or are on a tight deadline, keep using the stable releases and stop reading here. **Stable releases** are better tested and more reliable for everyday use than the others. **The majority of users will want to use stable releases only.**

The *dev* and *nightly* channels are unstable, which likely means that builds published through them will contain bugs and not work reliably. They are updated more often than stable releases.

Dev builds are available for everyone and are released inbetween stable releases. While not quite ready for everyday use yet, they showcase new features in a mostly unbroken fashion.

Finally, **nightly builds** are the bleeding edge, with frequent updates and also frequent problems of various degrees of severity. They are fun to try out, but do so at your own risk. Nightly builds are **only available for registered users**.

CHAPTER 3

Basic Concepts

3.1 Overview

To fully understand the rest of this guide, you need to be familiar with the concepts presented in this section.

3.2 Conventions

Written from the perspective of a Windows user, most instructions will only require trivial changes to work on other platforms.

Relative paths (e.g. `Packages/User`) start at the Data Directory unless otherwise noted.

We assume default key bindings when indicating keyboard shortcuts. If you're using a non-English keyboard layout, note that **some key bindings won't match your locale's keyboard**. This is due to the way Sublime Text maps keys to commands.

3.3 With Great Power Comes A Lot of Questions

Unquestionably a versatile tool for programmers, you don't need to be one in order to use Sublime Text, or even to configure it extensively. If you're a hacker, however, you are in for a great many pleasant surprises: Sublime Text can be infinitely customized and extended. You can start using it efficiently out of the box, but spending some time tailoring it to your exact needs will make it even better.

This guide will teach you how to configure Sublime Text.

Sublime Text can't be mastered in a day, but it's built on a handful of pervasive ideas that make for a consistent and easily understandable system once all the pieces come together.

In the following paragraphs, we'll outline key aspects that may not click in your mind until you've spent some time using the editor. Experiment, look around in this guide and, eventually, everything will fall into place.

3.4 The *Data* Directory

Nearly all of the interesting files for users live under the data directory. This is a platform-dependent location:

- **Windows**: *%APPDATA%\Sublime Text 3*
- **OS X**: *~/Library/Application Support/Sublime Text 3*

- **Linux**: *~/.config/sublime-text-3*

For **portable installations**, look inside *Sublime Text 3/Data*. Here, the *Sublime Text 3* part refers to the directory to which you've extracted the compressed portable files.

Note that only in portable installations does a directory named *Data* exist. For the remaining installation types, the data directory is the location indicated above.

3.5 The *Packages* Directory

This is a **key directory**: all resources for supported programming and markup languages are stored here. A *package* is a directory or zip file containing related files having a special meaning for Sublime Text.

You can access the packages directory from the main menu (**Preferences | Browse Packages...**), or by means of an API call: `sublime.packages_path()`. In this guide, we refer to this location as *Packages*, *packages path*, *packages folder* or *packages directory*.

3.5.1 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 will never overwrite the contents of `Packages/User` during upgrades.

3.6 The Python Console and the Python API

This information is especially interesting for programmers. For other users, you just need to know that Sublime Text enables users with programming skills to add their own features to the editor. (So go learn how to program; it's great fun!)

Sublime Text comes with an embedded Python interpreter. It's a useful tool to inspect the editor's settings and to quickly test API calls while developing plugins.

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

Confused? Let's try again more slowly:

Python is a programming language known to be easy for beginners and very powerful at the same time. *API* is short for 'Application Programming Interface', which is a fancy way of saying that Sublime Text 3 is prepared to be programmed by the user. Put differently, Sublime Text gives the user access to its internals through Python. Finally, a *console* is a little window inside Sublime Text that lets you type in short snippets of Python code and run them. The console also shows text output by Sublime Text or its plugins.

3.6.1 Your System's Python vs the Sublime Text 3 Embedded Python

Sublime Text 3 comes with its own Python interpreter and it's separate from your system's Python installation.

The embedded interpreter is intended only to interact with the plugin API, not for general development.

3.7 Packages, Plugins, Resources and Other Things That May Not Make Sense to You Now

Almost every aspect of Sublime Text can be extended or customized. For now, this is all you need to understand. This vast flexibility is the reason why you will learn about so many configuration files: there simply must be a place to specify all your preferences.

Among other things, you can modify the editor's behavior, add macros and snippets, extend menus... and even create whole new features –where *feature* means 'anything you can think of'. OK, right, there might be things you can't do, but you're definitely spoiled for choice.

All these configuration files we're referring to are simple text files following a special structure or *format*: JSON predominates, but you'll find some XML files, and Python files too for the more advanced extensibility options.

In this guide, for brevity, we refer collectively to all these disparate configuration files as *resources*.

Sublime Text will look for resources inside the packages folder. And what is a package, you ask? We'll talk at length about them, but the short version is that, to keep things tidy, the editor has a notion of a *package*, which is a folder containing resources that belong together (maybe they all help compose emails faster, write HTML efficiently, enhance the coding experience for C, Ruby, Go...).

3.8 Textmate Compatibility

This information is mainly useful for Textmate expats who've found a new home in Sublime Text. Textmate is an editor for the Mac.

Sublime Text compatibility with Textmate bundles is good excluding commands, which are incompatible. Additionally, Sublime Text requires all syntax definitions to have the *.tmLanguage* extension, and all preferences files to have the *.tmPreferences* extension. This means that *.plist* files will be ignored, even if they are located under a *Syntaxes* or *Preferences* subdirectory.

3.9 Vi/Vim Emulation

This information is mainly useful for dinosaurs and people who like to drop the term RSI in conversations. Vi is an ancient modal editor that lets the user perform all operations from the keyboard. Vim, a modern version of vi, is still in widespread use.

Sublime Text provides vi emulation through the *Vintage* package. The Vintage package is *ignored* by default. Read more about Vintage in the official documentation.

An evolution of Vintage called Vintageous offers a better Vi editing experience and is updated more often than Vintage. Vintageous is an open source project.

3.10 Emacs

This information is hardly useful for anyone. Emacs is... Well, nobody really knows what emacs is, but some people edit text with it.

If you are an emacs user, you're probably not reading this.

3.11 Be Sublime, My Friend

Borrowing from Bruce Lee's wisdom, Sublime Text can become almost anything you need it to be. In skilled hands, blah, blah, blah.

Empty your mind; be sublime, my friend.

CHAPTER 4

Editing

4.1 Overview

Sublime Text is brim-full of editing features. This topic just scratches the surface of what's possible.

4.2 Multiple Selections

Multiple selections let you make sweeping changes to your text efficiently. Any praise about multiple selections is an understatement. This is why:

Select some text and press `Ctrl + D` to **add more** instances. If you want **to skip the current instance**, press `Ctrl + K, Ctrl + D`.

If you go too far, press `Ctrl + U` to **deselect** the current instance.

4.3 Transforming Multiple Selections into Lines

`Ctrl + L` expands the selections to the end of the line. `Ctrl + Shift + L` splits the selections into lines.

You can copy multiple selected lines to a separate buffer, edit them there, select the content again as multiple lines and then paste them back into place in the first buffer.

4.4 Column Selection

You can select a rectangular area of a file. Column selection makes use of multiple selections.

It's possible to add blocks of text to or remove them from the selection.

4.4.1 Using the Mouse

Windows

Select Block	Right Mouse Button +
Add to Selection	Ctrl + Right Mouse Button +
Remove from Selection	Alt + Right Mouse Button +

Linux

Select Block	Right Mouse Button +
Add to Selection	Ctrl + Right Mouse Button +
Remove from Selection	Alt + Right Mouse Button +

OS X

Select Block	Right Mouse Button +
Add to Selection	+ Right Mouse Button +
Remove from Selection	+ + Right Mouse Button +

4.4.2 Using the Keyboard

Windows	Ctrl + Alt + Up and Ctrl + Alt + Down
Linux	Alt + + Up and Alt + + Down
OS X	+ + Up and + + Down

4.5 Other Ways of Selecting Text

The list is long; all available options can be found under **Selection**. To name a few:

- Select subwords (`Alt + Shift + <arrow>`)
- Expand selection to brackets (`Ctrl + Shift + M`)
- Expand selection to indentation (`Ctrl + Shift + J`)
- Expand selection to scope (`Ctrl + Shift + Space`)

4.6 Transposing Things

Need to swap two letters or, better yet, two words? Experiment with `Ctrl + T`.

4.7 And much, much more...

The **Edit**, **Selection**, **Find** and **Goto** menus are good places to look for handy editing tools. You might end up using just a few of them, but the rest will still be there for when you need them.

CHAPTER 5

Search and Replace

Sublime Text features two main types of search:

5.1 Search and Replace - Single File

5.1.1 Searching

To open the **search panel** for buffers, press `Ctrl + F`. Some options and actions available through this panel can be controlled from the keyboard:

Toggle Regular Expressions	`Alt + R`
Toggle Case Sensitivity	`Alt + C`
Toggle Exact Match	`Alt + W`
Find Next	`Enter`
Find Previous	`Shift + Enter`
Find All	`Alt + Enter`

5.1.2 Incremental Search

The **incremental search panel** can be brought up with `Ctrl + I`. The only difference with the regular search panel lies in the behavior of the `Enter` key. In incremental searches, it will select the next match in the file and dismiss the search panel for you. Choosing between this panel or the regular search panel is mainly a matter of preference.

5.1.3 Replacing Text

To open the replace planel, press `Ctrl + H`. Some actions available through this panel can be controlled from the keyboard.

Replace All:	`Ctrl + Alt + Enter`
Replace Next:	`Ctrl + Shift + H`

5.1.4 Tips

Other Ways of Searching in Files

Goto Anything provides the operator `#` to search in the current buffer, see *Goto Anything*.

Other Search-Related Key Bindings

These key bindings work when the search panel is hidden.

Search Forward Using Most Recent Pattern	`F3`
Search Backwards Using Most Recent Pattern	`Shift + F3`
Select All Matches Using Most Recent Pattern	`Alt + F3`

Multiline Search

You can type in multiline search patterns into search panels. To enter newline characters, press `Ctrl + Enter`. Note that search panels are resizable.

5.2 Search and Replace - Multiple Files

5.2.1 Searching

To open the search panel for files, press `Ctrl + Shift + F`. You can use the keyboard to control some search panel options and search actions:

The **Where** field in the search panel determines search scope. You can define scopes in several ways:

- Adding individual directories (Unix-style paths, even on Windows)
- Adding/excluding files based on a pattern
- Adding symbolic locations (`<open folders>`, `<open files>`)

It is also possible to combine these filters using commas; for example:

```
/C/Users/Joe/Top Secret,-\*.html,<open files>
```

Press the **...** button in the search panel to display a menu containing these options.

5.2.2 Results Format

In the search panel, you can customize the display of results with the following options:

- Show in Separate Buffer/Output Panel
- Show Context

5.2.3 Navigating Results

If the search yields matches, you can move through the sequence using the following key bindings:

We'll examine them in turn, but first let's talk about a powerful tool for searching text: **regular expressions**.

5.3 Regular Expressions

Regular Expressions find complex *patterns* in text. To take full advantage of the search and replace facilities in Sublime Text, you should at least learn the basics of regular expressions. In this guide we won't explain how to use regular expressions.

Typing out *regular expression* gets boring fast, and saying it is even more annoying, so it is usually shortened to *regexp* or *regex*.

This is how a regex might look:

```
(?:Sw|P)i(?:tch|s{2})\s(?:it\s)?of{2}!
```

To use regular expressions in Sublime Text, you first need to activate them in the various search panels. The search term will otherwise be interpreted literally.

Sublime Text uses Perl Regular Expression Syntax from the Boost library.

See also:

Boost library documentation for regular expressions Documentation on regular expressions.

Boost library documentation for format strings Documentation on format strings. Note that Sublime Text additionally interprets `\n` as `$n`.

CHAPTER 6

Build Systems (Batch Processing)

See also:

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

Build systems let you run your files through external programs like **make**, **tidy**, interpreters, etc.

Executables called from build systems must be in your `PATH`. For more information about making sure the `PATH` seen by Sublime Text is set correctly, see *Troubleshooting Build Systems*.

6.1 File Format

Build systems are JSON files and have the extension *.sublime-build*.

6.1.1 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 from an external program's output. This information is used to help you navigate through error instances with `F4`.

`selector` If the **Tools | Build System | Automatic** option is set, Sublime Text will automatically find the corresponding build system for the active file by matching `selector` to the file's scope.

In addition to options, you can use some variables in build systems too, as we have done above with `$file`, which expands to the active buffer's filename.

6.2 Where to Store Build Systems

Build systems must be located somewhere under the *Packages* folder (e.g. *Packages/User*). Many packages include their own build systems.

6.3 Running Build Systems

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

CHAPTER 7

File Navigation and File Management

7.1 Goto Anything

Use Goto Anything to **navigate your project's files** swiftly. (More about projects later.)

To open Goto Anything, press `Ctrl+P`. As you type into the input area, names of open files and files in *open directories* will be searched, and a preview of the best match will be shown. This preview is *transient*; that is, it won't become the actual active view until you perform some operation on it. Transient views go away when you press `Esc`. You will find transient views in other situations, for example when single-clicking a file in the sidebar.

Goto Anything lives up to its name –there's more to it than locating files.

7.1.1 Goto Anything Operators

Goto Anything understands a handful of operators. Any of them can be used in combination with file search queries.

Example:

```
island:123
```

This instructs Sublime Text to first search for a file that matches `island` and then go to line 123.

7.1.2 Supported Operators

`@symbol` Searches for **symbol** symbol in the active buffer; bound to `Ctrl+R`.

> Symbols usually are classes or functions, but can target any scope present in the syntax definition. See *Symbols - Syntax Preferences* (XXX to be added). If no symbols are defined, the search will simply fail.

`#term` Fuzzy-searches a word in the file matching **term** and highlights all instances; bound to `Ctrl+;`.

`:line_number` Goes to the specified line number or the end of the file if it exceeds the file limit; bound to `Ctrl+G`.

Note: Searching for symbols will only work if the active file type has symbols defined for it. Symbols are defined in *.tmLanguage* files.

7.2 Sidebar

The sidebar gives you an overview of the active project. Files and folders added to the sidebar will be available in Goto Anything and project-wide actions. actions (like project-wide searches).

Projects and the sidebar are closely related. It's important to note that there's always an active project, whether it's explicit or implicit.

To **toggle** the sidebar, press `Ctrl+K, Ctrl+B`.

The sidebar can be navigated with the arrow keys, but first you need to give it the **focus** by pressing `Ctrl+0`. To return the focus to the view, press `Esc`. Alternatively, you can use the mouse to the same effect.

Files opened from the sidebar create *semi-transient* views. Unlike transient views, *semi-transient* views show up as a new tab. You will be able to tell semi-transient views from other views because their tab text is shown in italics. When a new semi-transient view is opened, any existing semi- transient view in the same pane gets automatically closed.

The sidebar provides basic file management operations through its context menu.

7.3 Projects

Projects group sets of files and folders to keep your work organized. Set up a project by adding folders in a way that suits you, and then save your new configuration. You can add and remove folders to a project with the **Project** menu and the side bar's context menu. Alternatively, you can drag a folder onto a window and it will be added automatically.

To save a project, go to **Project | Save Project As...**.

To switch projects quickly, press `Ctrl+Alt+P`. Using the menu, you can select **Projects | Recent Projects**.

Project data are stored in JSON files with a *.sublime-project* extension. Wherever there's a *.sublime-project* file, you will find an ancillary *.sublime-workspace* file too. The second one is used by Sublime Text and you shouldn't edit it yourself.

Project files can define settings specific to that project. More information in the official documentation.

You can open a project from the **command line** by passing the *.sublime- project* file as an argument to the Sublime Text executable.

Project files are meant to be committed to source code repositories.

7.3.1 Project Definitions

Project definitions are stored in JSON files with a *.sublime-project* extension. Wherever there's a *.sublime-project* file, you will find an ancillary *.sublime-workspace* file too, which contains user specific data, such as the open files and the modifications to each. The latter is used by Sublime Text and isn't meant to be edited by users.

Project definitions support three top level sections: `folders`, for the included folders, `settings`, for settings overrides, and `build_systems`, for project-specific build systems.

```
{
    "folders":
    [
        {
            "path": "src",
            "folder_exclude_patterns": ["backup"]
        },
        {
            "path": "docs",
```

```
            "name": "Documentation",
            "file_exclude_patterns": ["*.css"]
        }
    ],
    "settings":
    {
        "tab_size": 8
    },
    "build_systems":
    [
        {
            "name": "List",
            "cmd": ["ls"]
        }
    ]
}
```

Folders Each folder must have a `path`, and may optionally have a `folder_exclude_patterns` and `file_exclude_patterns` setting. The path may be relative to the project directory, or an absolute path. Folders may also be given a `name` that will appear in the side bar.

Settings A project may define project-specific settings that will only apply to files within that project. Project-specific settings override regular user settings, but not syntax-specific settings.

Almost all settings can be overridden (excluding global settings).

See also:

The Settings Hierarchy A detailed example for the order of precedence for settings.

Settings - Reference Reference of available settings.

Build Systems You can define project-specific build systems in a project definition. In addition to regular build systems, a `name` must be specified for each one. Build systems listed here will be available via the regular **Tools | Build Systems** menu.

See also:

Build Systems - Reference Documentation on build systems and their options.

7.4 Notable Settings Related to The Sidebar and Projects

These options control which files are shown in the sidebar and included in project-wide actions, such as searching files.

folder_exclude_patterns file_exclude_patterns binary_file_patterns

To see a detailed description of their purpose, open the default settings file (*Default/Preferences.sublime-settings*) via the Command Palette (*Ctrl+P*).

7.5 Workspaces

Workspaces can be seen as different *views* into the same project. For example, you may want to have only a selected few files open while working on *Feature A*. Or perhaps you use a different pane layout when you're writing tests, etc. Workspaces help in these situations.

**Workspaces behave very much like projects. To create a new workspace, select **Project | New Workspace for Project. To save the active workspace, select **Project | Save Workspace As....

Workspaces data is stored in JSON files with the *.sublime-workspace* extension.

Contrary to *.sublime-project* files, *.sublime-workspace* files **are not** meant to be shared or edited manually. **Never** commit *.sublime-workspace* files into a source code repository.

To switch between different workspaces, use `Ctrl+Alt+P`, exactly as you do with projects.

As with projects, you can open a workspace from the **command line** by passing the desired *.sublime-workspace* file as an argument to the Sublime Text executable.

7.6 Panes

Panes are groups of views. In Sublime Text you can have multiple panes open at the same time.

To create a new pane, press `Ctrl+K, Ctrl+Up`. To close a pane, press `Ctrl+K, Ctrl+Down`.

Further pane management commands can be found under **View | Layout** and related submenus.

CHAPTER 8

Customizing Sublime Text

Sublime Text can be fully customized. In the following topics, we outline the most common options. In particular, **we don't cover** themes and color schemes, an immensely configurable area of Sublime Text.

8.1 Settings

Sublime Text stores configuration data in *.sublime-settings* files. Flexibility comes at the price of a slightly complex system for applying settings. However, here's a rule of thumb:

Always place your personal settings files under *Packages/User* to guarantee they will take precedence over any other conflicting settings files.

With that out of the way, let's unveil, to please masochistic readers, the mysteries of how settings work.

8.1.1 Format

Settings files use JSON and have the *.sublime-settings* extension.

8.1.2 Types of Settings

The name of each *.sublime-settings* file determines its purpose. Names can be descriptive (like `Preferences (Windows).sublime-settings` or `Minimap.sublime-settings`), or they can be related to what the settings file is controlling. For example, file type settings need to carry the name of the *.tmLanguage* syntax definition for the file type. Thus, for the *.py* file type, whose syntax definition is contained in `Python.tmLanguage`, the corresponding settings files would be called `Python.sublime-settings`.

Also, some settings files only apply to specific platforms. This can be inferred from the file names, e.g. `Preferences (`*`platform`*`).sublime-settings`. Valid names for *platform* are `Windows`, `Linux`, `OSX`.

This is **important**: Platform-specific settings files in the `Packages/User` folder are ignored. This way, you can be sure a single settings file overrides all the others.

Settings changes are usually updated in real time, but you may have to restart Sublime Text in order to load *new* settings files.

8.1.3 How to Access and Edit Common Settings Files

Unless you need very fine-grained control over settings, you can access the main configuration files through the **Preferences | Settings - User** and **Preferences | Settings - More** menu items. Editing **Preferences | Settings -**

Default is discouraged, because changes will be reverted with every update to the software. However, you can use that file for reference: it contains comments explaining the purpose of all available global and file type settings.

8.1.4 Order of Precedence of *.sublime-settings* Files

The same settings file (such as `Python.sublime-settings`) can appear in multiple places. All settings defined in identically named files will be merged together and overwritten according to predefined rules. See *Merging and Order of Precedence* for more information.

Let us remember again that any given settings file in `Packages/User` ultimately overrides every other settings file of the same name.

In addition to settings files, Sublime Text maintains *session* data—settings for the particular set of files being currently edited. Session data is updated as you work on files, so if you adjust settings for a particular file in any way (mainly through API calls), they will be recorded in the session and will take precedence over any applicable *.sublime-settings* files.

To check the value of a setting for a particular file being edited, use `view.settings().get("setting_name")` from the console.

Finally, it's also worth noting that some settings may be automatically adjusted for you. Keep this in mind if you're puzzled about some setting's value. For instance, this is the case for certain whitespace-related settings and the `syntax` setting.

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

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

See The Settings Hierarchy for a full example of the order of precedence.

8.1.5 Global Editor Settings and Global File Settings

These settings are stored in `Preferences.sublime-settings` and `Preferences (platform).sublime-settings` files. The defaults can be found in `Packages/Default`.

Valid names for *platform* are `Windows`, `Linux`, `OSX`.

8.1.6 File Type Settings

If you want to target a specific file type, name the *.sublime-settings* file after the file type's syntax definition. 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 +:file:*Packages/Python*, but there can be multiple settings files in separate locations for the same file type.

Similarly to global settings, one can establish platform-specific settings for file types. For example, `Python (Linux).sublime-settings` would only be consulted only under Linux.

Also, let us emphasize that under `Pakages/User` only `Python.sublime-settings` would be read, but not any `Python (platform).sublime-settings` variant.

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

8.1.7 The Settings Hierarchy

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

- `Packages/Default/Preferences.sublime-settings`
- `Packages/Default/Preferences (Windows).sublime-settings`
- `Packages/AnyOtherPackage/Preferences.sublime-settings`
- `Packages/AnyOtherPackage/Preferences (Windows).sublime-settings`
- `Packages/User/Preferences.sublime-settings`
- Settings from the current project
- `Packages/Python/Python.sublime-settings`
- `Packages/Python/Python (Windows).sublime-settings`
- `Packages/User/Python.sublime-settings`
- Session data for the current file
- Auto-adjusted settings

8.1.8 Where to Store User Settings (Once Again)

Whenever you want to save settings, especially if they should be preserved between software updates, place the corresponding *.sublime-settings* file in `Packages/User`.

8.2 Indentation

See also:

Indentation Official Sublime Text Documentation.

8.3 Key Bindings

See also:

Reference for key bindings Complete documentation on key bindings.

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

8.3.1 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" }
]
```

8.3.2 Defining and Overriding Key Bindings

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

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

8.3.3 Advanced Key Bindings

Simple key bindings consist of a key combination and a command to be executed. However, there are more complex syntaxes for passing arguments and 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 whether 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, unless you are cycling through snippet fields, pressing `ESC` 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`.)

Keys combinations

You can create a key binding which will be triggered only if a combination of multiple keys is stroked in sequence. To use it, you just have to add a second value in the `keys` array:

```
{ "keys": ["ctrl+k", "ctrl+v"], "command": "paste_from_history" }
```

Here, to trigger the command `paste_from_history`, you have to press `Ctrl` + `k` first, release the key `k`, then press the key `v`.

Note: this example is a default key binding, so you don't need to add it to your config file and you can try it right now !

8.4 Menus

No documentation is available about this topic.

CHAPTER 9

Extending Sublime Text

The following topics show various ways in which Sublime Text can be extended with additional functionality.

9.1 Commands

Commands are ubiquitous in Sublime Text: key bindings, menu items and macros all work through the command system. They are found in other places too.

Some commands are implemented in the editor's core, but many of them are provided as Python plugins. Every command can be called from a Python plugin.

9.1.1 Command Dispatching

Normally, commands are bound to the application object, a window object or a view object. Window objects, however, will dispatch commands based on input focus, so you can issue a view command from a window object and the correct view instance will be found for you.

9.1.2 Anatomy of a Command

Commands have a name separated by underscores (snake_case) like `hot_exit`, and can take a dictionary of arguments whose keys must be strings and whose values must be JSON types. Here are a few examples of commands run from the Python console:

```
view.run_command("goto_line", {"line": 10})
view.run_command('insert_snippet', {"contents": "<$SELECTION>"})
view.window().run_command("prompt_select_project")
```

See also:

Reference for commands Command reference.

9.2 Macros

Macros are a basic automation facility comprising 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 *.sublime-macro* extension. Sublime Text ships with a few macros providing core functionality, such as line and word deletion. You can find these under **Tools | Macros** or in +:file:*Packages/Default*.

9.2.1 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. Now you will 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 remember only the latest recorded macro. Also, macros only record commands sent to the buffer: window-level commands, such creating a new file, will be ignored.

9.2.2 How to Edit Macros

As an alternative to recording a macro, you can edit it by hand. Just save a new file with the `.sublime-macro` extension under `Packages/User` and add commands to it. Macro files have this format:

```
[
    {"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 `\`.

9.2.3 Where to Store Macros

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

Macro files can be run by the `run_macro_file` command. See *Commands* section for more information about commands.

9.3 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 a skeleton for it.

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.

9.3.1 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 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 to get a literal `$`, you have to escape it like this: `\$`.
- When writing a snippet that contains indentation, always use tabs. When the snippet is inserted, the tabs will be transformed into spaces if the option `translateTabsToSpaces` is `true`.
- The `content` must be included in a `<![CDATA[...]]>` section. Snippets won't work if you don't do this!
- The `content` of your snippet must not contain `]]>` because this string of characters will prematurely close the `<![CDATA[...]]>` section, resulting in an XML error. To work around this pitfall, you can insert an undefined variable into the string like this: `]]$NOT_DEFINED>`. This modified string passes through the XML parser without closing the content element's `<![CDATA[...]]>` section, but Sublime Text will replace `$NOT_DEFINED` with an empty string before inserting the snippet into your file. In other words, `]]$NOT_DEFINED>` in your snippet file `content` will be written as `]]>` when you trigger the snippet.

`tabTrigger` Defines the sequence of keys that must be pressed to insert this snippet. After typing this sequence, the snippet will kick in as soon as you hit the `Tab` key.

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 file 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.

9.3.2 Snippet Features

Environment Variables

Snippets have access to contextual information in the form of environment variables. The values of the variables listed below are set automatically by Sublime Text.

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

$PARAM1 .. $PARAMn	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 cursor's line when the snippet was triggered.
$TM_CURRENT_WORD	Word under the cursor when the snippet was triggered.
$TM_FILENAME	Name of the file being edited, including extension.
$TM_FILEPATH	Path to the file being edited.
$TM_FULLNAME	User's user name.
$TM_LINE_INDEX	Column where the snippet is being inserted, 0 based.
$TM_LINE_NUMBER	Row where the snippet is being inserted, 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 after 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. By default, this is the end of the snippet.

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 in real time with the same value.

```
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".

Placeholders

By expanding the field syntax a little bit, you can define default values for a field. Placeholders are useful whenever there's a general case for your snippet, but you still want to keep it customizable.

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

Variables can be used as placeholders:

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

And you can nest placeholders within other placeholders too:

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

Substitutions

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. Of course, the tab stop you want to use as variable has to be mirrored somewhere else in the snipept.

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 documentation for regular expressions.

format_string See the Boost library documentation 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: =========
```

Another more complex example can translate snail_case to Tile Case With Spaces. Basically, it combines two separate expressions and replaces into one. It also illustrates that replaces may occur before the actual tabstop.

```
Transformation: ${1/^(\w)|(?:_(\w))/(?1\u$1:)(?2 \u$2:)/g}
      Original: ${1:text_in_snail_case}

# Output:

Transformation: Text In Snail Case
      Original: text_in_snail_case
```

9.4 Completions

See also:

Sublime Text Documentation Official documentation on this topic.

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

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

- Snippets
- API-injected completions
- Buffer contents

However, the most explicit way Sublime Text provides you to feed it completions is by means of `.sublime-completions` files. This topic deals with the creation of a `.sublime-completions` file as well as with the interactions between all sources for completions.

9.4.1 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 example (with 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.

completions Array of *completions*.

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": "foo", "contents": "foobar" }
```

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

> You can use a `\t` tab character to separate the trigger from a brief description on what the completion is about, it will be displayed right-aligned and slightly grayed and does not affect the trigger itself.

contents Text to be inserted in the buffer. Can use *Snippet Features*.

9.4.2 Sources for Completions

These are the sources for completions the user can control:

- *Snippets*
- *.sublime-completions*
- API-injected completions via `EventListener.on_query_completions()`

Additionally, other completions are folded into the final list:

- Words in the 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 completion sources, a fuzzy match is performed. Furthermore, snippets always lose with fuzzy matches.

But this is relevant only when the completion is inserted automatically. When a list of completions is shown, snippets will still be listed alongside the other items, even if the prefix only partially matches the snippets' tab triggers.

9.4.3 How to Use Completions

There are two methods for using completions. Even though, when screening them, the priority given to completions always stays the same, the two methods produce different results, as explained next.

Completions can be inserted in two ways:

- through the completions list (`Ctrl+spacebar`), or
- By pressing `Tab`.

The Completions List

To use the completions list:

- Press `Ctrl+spacebar` to open
- Optionally, press `Ctrl+spacebar` again to select next entry or use up and down arrow keys
- Press `Enter` or `Tab` to validate selection (depending on the `auto_complete_commit_on_tab`)

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

The completions list may work in two ways: by bringing up a list of suggested words to be completed, or by inserting the best match directly. The automatic insertion will only be done if the list of completion candidates can be narrowed down to one unambiguous choice given the current prefix.

If the choice of best completion is ambiguous, an interactive list will be presented to the user. Unlike other items, snippets in this list are displayed in this format: *`tab_trigger`*`\t`*`name`*.

Completions with multiple cursors

Sublime Text can also handle completions with multiple cursors but will only open the completion list when all cursors share the same text between the current cursor position and the last word separator character (e.g. `.` or a line break).

Working example (`|` represents one cursor):

```
l|
some text with l|
l| and.l|
```

Not working example:

```
l|
some text with la|
l| andl|
```

Selections are essentially ignored, only the position of the cursor matters. Thus, `e|[-some selection] example`, with `|` as the cursor and `[...]` as the current selection, completes to `example|[-some selection] example`.

`Tab`-Completed Completions

If you want to be able to tab-complete completions, the setting `tab_completion` must be set to `true` (default). 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. This means, unlike the case of the completions list, that Sublime Text will always make the decision for you. The rules for selecting the best completion are the same as described above, but in case of ambiguity, Sublime Text will insert the item it deems most suitable.

Inserting a Literal Tab Character

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

9.5 Command Palette

See also:

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

9.5.1 Overview

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

By default, the command palette includes many useful commands, and provides convenient access to individual settings as well as settings files.

9.5.2 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`.

9.5.3 How to Use the Command Palette

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

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

9.6 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* that divide the text in a buffer into 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, as well as 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*.

9.6.1 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. It lives in a public Mercurial repository at Bitbucket.

Download the latest `.sublime-package` file and install it 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.

9.6.2 File format

Sublime Text uses property list (Plist) files to store syntax definitions. However, because editing XML files is a cumbersome task, we'll use JSON instead, and convert it to Plist format 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 always keep in mind their differing needs in regards to escape sequences, etc.

9.6.3 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 bound to the snippet and looks at the caret's position in the file. If the caret's current position matches the snippet's scope selector, Sublime Text fires it 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, while 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 inherits the idea of scopes from Textmate, a text editor for Mac. Textmate's online manual contains further information about scope selectors that's useful for Sublime Text users too.

9.6.4 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 therefore will 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.

9.6.5 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 the phrase *to style* to mean *to break down a source code file into scopes*. Keep in mind, however, that colors are a different thing from syntax definitions and that scopes have many more uses besides syntax highlighting.

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

- Variables (`$PARAM1`, `$USER_NAME`...)
- Simple fields (`$0`, `$1`...)
- Complex fields with placeholders (`${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 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 in your `Packages/User` folder as a `.JSON-tmLanguage` file.

You now should 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 use the name of the programming language 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, use `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` A container for your patterns.

For our example, fill 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.

9.6.6 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

Matches 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 find matches.

`name` The name of the scope that should be applied to any 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 above, they are simply XML files, but in 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
- Sublime Text will reload the changes to the syntax definition

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

Fine Tuning Matches

You might have noticed, for instance, that the entire text in `$PARAM1` is styled the same way. Depending on your needs or your personal preferences, you may want the `$` 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 that is clearly delimited by start and end marks.

Literal strings enclosed by quotation marks or other delimiting constructs are better dealt with by 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 that includes 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). Effectively, this will create nested scopes for `beginCaptures`, `endCaptures` and `patterns` defined within this rule. Optional.

patterns An array of patterns to match **only** against the begin-end's content—they aren't matched against the text consumed by `begin` or `end` themselves.

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 their appearance here.

We've seen above that fields can be nested. In order to account for this, we need to style nested fields recursively. That's what the `include` rule does when we furnish it the `$self` value: it recursively applies our entire syntax definition the text captured by our begin-end rule. This portion excludes the text individually consumed by the regexes for `begin` and `end`.

Remember, matched text is consumed; thus, it is excluded from the next match attempt.

To finish off complex fields, we'll style placeholders 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 then we can 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 place the second rule after any others matching the `$` character, since otherwise you may not get the desired results.

Also, even after adding these two additional rules, note that our recursive begin-end rule from above continues to work 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.

9.7 Plugins

See also:

API Reference More information on the Python API.

Plugins Reference More information about plugins.

This section is intended for users with programming skills.

Sublime Text can be extended through Python plugins. Plugins build features by reusing existing commands or creating new ones. Plugins are a logical entity, rather than a physical one.

9.7.1 Prerequisites

In order to write plugins, you must be able to program in Python. At the time of this writing, Sublime Text used Python 3.

9.7.2 Where to Store Plugins

Sublime Text will look for plugins only in these places:

- `Installed Packages` (only *.sublime-package* files)
- `Packages`
- `Packages/<pkg_name>/`

As a consequence, any plugin nested deeper in `Packages` won't be loaded.

Keeping plugins directly under `Packages` is discouraged. Sublime Text sorts packages in a predefined way before loading them, so if you save plugin files directly under `Packages` you might get confusing results.

9.7.3 Your First Plugin

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

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 the newly created buffer.

9.7.4 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!")
```

Both the `sublime` and `sublime_plugin` modules are provided by Sublime Text; they are not part of the Python standard library.

As we mentioned earlier, plugins reuse or create *commands*. Commands are an essential building block in Sublime Text. They are simply Python classes that can be called in similar ways from different Sublime Text facilities, like the plugin API, menu files, macros, etc.

Sublime Text Commands derive from the `*Command` classes defined in `sublime_plugin` (more on this later).

The rest of the code in our example is concerned with particulars of `TextCommand` or with the API. We'll discuss those topics in later sections.

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

Conventions for Command Names

You may have noticed that our command is named `ExampleCommand`, but we passed the string `example` to the API call instead. This is necessary because Sublime Text standardizes command names by stripping the `Command` suffix and separating `PhrasesLikeThis` with underscores, like so: `phrases_like_this`.

New commands should follow the same naming pattern.

9.7.5 Types of Commands

You can create the following types of commands:

- Window commands (`sublime_plugin.WindowCommand`)
- Text commands (`sublime_plugin.TextCommand`)

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

Shared Traits of Commands

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

Note: Parameters to commands must be valid JSON values due to how ST serializes them internally.

Window Commands

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

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

The `.run()` method of a window command doesn't require any positional parameter.

Window commands are able to route text commands to their window's active view.

Text Commands

Text commands operate at the view level, so they require a view to exist in order to be available.

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

The `.run()` method of text commands requires an `edit` instance as its first positional argument.

Text Commands and the `edit` Object

The edit object groups modifications to the view so that undo and macros work sensibly.

Note: Contrary to older versions, Sublime Text 3 doesn't allow programmatic control over edit objects. The API is in charge of managing their life cycle. Plugin creators must ensure that all modifying operations occur inside the `.run` method of new text commands. To call existing commands, you can use `view.run_command(<cmd_name>, <args>)` or similar API calls.

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's Autocomplete service and then feeds it to the Sublime Text 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: Make sure you don't keep this plugin around after trying it or it will interfere with the autocompletion system.

See also:

`EventListener.on_query_completions()` Documentation on the API event used in this example.

9.7.6 Learning the API

In order to create plugins, you need to get acquainted with the Sublime Text API 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.

In particular, the `Packages/Default` contains many examples of undocumented commands and API calls. Note that you will first have to extract its content to a folder if you want to take a look at the code within. As an exercise, you can try creating a build system to do that on demand, and a project file to be able to peek at the sample code easily.

9.8 Packages

Packages are simply folders under `Packages`, or zip archives with the *.sublime-package* extension saved under `Installed Packages`.

Here's a list of typical resources that can be found inside packages:

- build systems (`.sublime-build`)
- key maps (`.sublime-keymap`)
- macros (`.sublime-macro`)
- menus (`.sublime-menu`)
- plugins (`.py`)
- syntax 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 *$PATH_TO_SUBLIME_TEXTPackagesLanguage - English.sublime-package* as a data store for English dictionaries.

9.8.1 Types of Packages

In this guide, we classify packages under different categories. This classification is artificial and useful just for clarity when discussing this topic. Sublime Text doesn't use this classification in any way.

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

shipped packages Included in every installation, though technically not required. They enhance Sublime Text out of the box. May have been contributed by users or third parties.

user packages Installed by the user to extend Sublime Text's functionaility. They are not part of any Sublime Text installation, and are always contributed by users or third parties.

installed packages Packages stored under *Installed Packages* as *.sublime-package*'s

It's worth noting that by *third party* we mainly refer to users of other editors, such as Textmate.

9.8.2 Package Installation

Ultimately, installing a package is simply a matter of copying a folder containing Sublime Text resources to `Packages`, or a *.sublime-package* file to *Installed Packages*. The only thing that varies is how you obtain and 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` folder and restart Sublime Text. If the `Installed Packages` folder doesn't exist, you can create it.

Note that `.sublime-package` files simply are `.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 free of charge on public repositories like GitHub and Bitbucket.

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

9.8.3 Packages and Magic

Sublime Text deals with packages without much hidden magic. There are two notable exceptions: Macros defined in any package automatically appear under **Tools | Macros | <Your Package>**, and snippets from any package appear under **Tools | Snippets | <Your Package>**.

However, Sublime Text follows some rules for packages. For instance, `Package/User` will never be clobbered during updates to 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* receive special treatment when merging files (e.g. *.sublime-keymap* and *.sublime-settings* files). Before merging can take place, the files have to be arranged in some order. To that end, Sublime Text sorts them alphabetically by name, with the exception of the *Default* and *User* folders. Files contained in *Default* will always go to the front of the list, and those in *User*, to the end.

9.8.4 Ignored Packages

To temporarily disable packages, you can add them to the *ignored_packages* list in your *Packages/User/Preferences.sublime-settings* file.

9.8.5 Restoring Packages

Sublime Text keeps a copy of all installed packages so it can recreate them as needed. This means it can reinstall core packages, shipped packages and, potentially, user packages alike. However, only user packages installed as `sublime-packages` are added to its registry of installed packages. Packages installed in alternative ways will be lost completely 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 that the `Installed Packages` folder will be deleted too, so you'll lose all your installed packages.

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

Reverting Sublime Text to a fresh state solves many problems that appear to be due to bugs in Sublime Text but are in fact caused by misbehaving plugins.

9.8.6 The `Installed Packages` Directory

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

CHAPTER 10

Command Line Usage

See also:

OS X Command Line Official Sublime Text Documentation

CHAPTER 11

Reference

This section contains concise technincal information about Sublime Text. It is intended mainly as a quick reference for advanced users who want to modify Sublime Text's default behavior.

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

11.1 Syntax Definitions

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

11.1.1 Compatibility with Textmate

Generally, Sublime Text syntax definitions are compatible with Textmate language files.

11.1.2 File Format

Textmate syntax definitions are Plist files with the `tmLanguage` extension. However, for convenience in this reference document, JSON is shown instead.

```
{ "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 the 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 An array of file type extensions for which this syntax automatically should be 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.

11.1.3 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": "\\}"
    }
```

11.1.4 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" }
    ]
  },
}
```

11.1.5 Escape Sequences

Be sure to escape JSON/XML sequences as needed.

11.2 Build Systems

Build systems let you run your files through external programs without leaving Sublime Text, and see the output they generate.

Build systems consist of one –or optionally three– parts:

- Configuration data in JSON format (the *.sublime-build* file contents)
- Optionally, a Sublime Text command driving the build process
- A Sublime Text command driving the build process
- An optional, external executable file (script or binary file)

Essentially, *.sublime-build* files are configuration data for an external program, as well as for a Sublime Text command (just mentioned). In them, you specify the switches, options and environment information you want forwarded.

The Sublime Text command then receives the data stored in the *.sublime-build* file. At this point, it can do whatever it needs to do, to *build* the files. By default, build systems will use the `exec` command implemented by `Packages/Default/exec.py`. As we'll explain below, you can override this command.

Finally, the external program may be a shell script you've created to process your files, or a well-known utility like `make` or `tidy`. Usually, these executable files will receive paths to files or directories, along with switches and options to run with.

Note that build systems can but don't need to call external programs; a valid build system could be implemented entirely in Python in a Sublime Text command.

11.2.1 File Format

.build-system files use JSON. Here's an example:

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

Build system-specific options

These options are standard for all build systems.

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

Used to override the default build system command. Note that if you choose to override the default command for build systems, you can add arbitrary variables in the *.sublime-build* file.

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.

windows, osx and linux Optional. Allow specification of OS-specific options which will override the default settings. These accept a dict of Arbitrary options each.

See Platform-specific Options.

variants Optional. A list of dictionaries of options to override the main build system's options. Variant names will appear in the Command Palette for easy access if the build system's selector matches for the active file.

See Variants.

name **Only valid inside a variant** (see `variants`). Identifies variant build systems. If `name` is *Run*, the variant will show up under the **Tools | Build System** menu and be bound to `Ctrl+Shift+B`.

Arbitrary options

Due to the `target` setting a build system can contain literally any option (key) that is not one of the options already listed above.

Please note that all the options below are from the default implementation of `exec` (see *exec command*). If you change the `target` option, these can no longer be relied on.

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

On Windows, GUIs are supressed.

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

line_regex Optional. If `file_regex` doesn't match on the current line, but `line_regex` exists, 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.

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`.

env Optional. Dictionary of environment variables to be merged with the current process' before passing them to `cmd`.

Use this element, for example, to add or modify environment variables without modifying your system's settings.

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.

Use this option to add directories to `PATH` without having to modify your system's settings.

syntax Optional. When provided, the build system output will be formatted with the provided syntax definition.

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: *filename*, *line number*, *column number* and *error message*. Use groups in the pattern to capture this information. The *filename* 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

The `windows`, `osx` and `linux` elements let you provide platform-specific data in the build system. 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.

Variants

Here's a contrived example of a build system with variants

```
{
    "selector": "source.python",
    "cmd": ["date"],

    "variants": [

        { "name": "List Python Files",
          "cmd": ["ls -l *.py"],
          "shell": true
        },

        { "name": "Word Count (current file)",
          "cmd": ["wc", "$file"]
        },

        { "name": "Run",
          "cmd": ["python", "-u", "$file"]
        }
    ]
}
```

Given these settings, `Ctrl+B` would run the *date* command, `Crtl+Shift+B` would run the Python interpreter and the remaining variants would appear in the *Command Palette* as `Build: name` whenever the build system was active.

11.2.2 Build System Variables

Build systems expand the following variables in *.sublime-build* files:

`$file_path`	The directory of the current file, e.g., *C:\Files*.
`$file`	The full path to the current file, e.g., *C:\Files\Chapter1.txt*.
`$file_name`	The name portion of the current file, e.g., *Chapter1.txt*.
`$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.

Placeholders 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*.

See also:

Snippets Documentation on snippets and their variable features.

11.2.3 Running Build Systems

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

11.2.4 Troubleshooting Build Systems

Build systems will look for executables in your `PATH`, unless you specify an absolute path to the executable. Therefore, your `PATH` variable must be correctly set.

On some operating systems, the value of `PATH` may vary between terminal windows and graphical applications. Thus, in your build system, even if the command you are using works in the command line, it may not work from Sublime Text. This is due to user profiles in shells.

To solve this issue, make sure you set the desired `PATH` so that graphical applications such as Sublime Text can find it. See the links below for more information.

Alternatively, you can use the `path` key in *.sublime-build* files to override the `PATH` used to locate the executable specified in `cmd`. This new value for `PATH` will be in effect only as long as your build system is running. After that, the old `PATH` will be restored.

See also:

Managing Environment Variables in Windows Search Microsoft knowledge base for this topic.

Setting environment variables in OSX StackOverflow topic.

11.3 Key Bindings

Key bindings map key presses to commands.

11.3.1 File Format

Key bindings are stored in `.sublime-keymap` files and defined in JSON. All key map filenames 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 a next snippet field is available.

has_prev_field Returns `true` if a previous snippet field is 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 just to the text following the caret.

preceding_text Restricts the test just 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 just to the selected text.

selector Returns the current scope.

panel_has_focus Returns `true` if the current focus is on a panel.

panel Returns `true` if the panel given as operand is visible.

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).

11.3.2 Command Mode

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

11.3.3 Bindable Keys

Keys may be specified literally or by name. Here's 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

If you're developing a package, keep this in mind:

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

In both cases, the user's ability to insert non-ASCII characters would be compromised.

If you are the end-user, you are free to remap those key combinations.

11.3.4 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.

11.3.5 International Keyboards

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

11.3.6 Troubleshooting

To enable command logging, see sublime.log_commands(flag). This may help in debugging key maps.

11.4 Settings (Reference)

+.. warning:

```
This page may be outdated and contain wrong or not all information. However,
you can find most of the available settings with a brief description in the
default settings file (**Preferences | Settings - Default** or   :file:'Default/Preferences.sublime-s
```

See also:

Customization - Settings A detailed overview on settings in Sublime Text and their order of precedence.

11.4.1 Global Settings

These settings can only be modified from `Preferences.sublime-settings` and `Preferences (platform).sublime-settings`.

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

`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.

`hot_exit` Exiting the application or window with an associated project with `hot_exit` enabled will cause it to close immediately without prompting. Unsaved modifications and open files will be preserved and restored when next starting.

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

`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.

`close_windows_when_empty` Close windows as soon as the last file is closed, unless there's a folder open within the window.

`show_full_path` Show the full path to files in the title bar.

`preview_on_click` If `true`, preview file contents when clicking on a file in the side bar. Double clicking or editing the preview will open the file and assign it a tab.

`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.

binary_file_patterns Excludes the matching files from GoTo Anything and Find in Files but not the side bar.

show_tab_close_buttons If `false`, hides the tabs' close buttons until the mouse hovers 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 whether a new window is created or not.

ignored_packages A list of packages that will be ignored (not loaded).

11.4.2 File Settings

Whitespace and Indentation

auto_indent Toggles automatic indentation.

tab_size Number of spaces a tab is considered 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 automatically will 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 (such as `[79, 89, 99]`) or a single numeric value (for example, `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 `word_wrap` is set to `true`.

indent_subsequent_lines If set to `false`, wrapped lines will not be indented. Only takes effect if `word_wrap` 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 this to `false` if you'd rather have brackets highlighted only 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 save files automatically 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 divide words for actions like advancing the cursor, etc. Not used for every context where a notion of a word separator is useful (for example, word wrapping). In some contexts, the text might be tokenized based on other criteria (for example, 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 (such as **:path:'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 detected automatically .

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 and result_line_regex Regular expressions 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*.

result_base_dir Folder to start looking for offending files 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 folder for the view.

Input Settings

command_mode If set to `true`, the buffer will ignore key strokes. Useful when emulating Vim's modal behavior.

11.5 Command Palette

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

11.5.1 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 differs from other areas of the editor due to different implementations in the lower layers.

11.5.2 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.

11.6 Plugins

See also:

API Reference More information on the Python API.

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

11.6.1 Where to Store Plugins

Sublime Text will look for plugins in these places:

- `Packages`
- `Packages/<pkg_name>`
- `.sublime-package` files

Plugin files nested deeper in `Packages` won't be loaded.

All plugins should live inside a folder of their own and not directly under `Packages`. This will spare you confusions when Sublime Text attempts to sort packages for loading.

11.6.2 Conventions for Command Names

By convention, Sublime Text command class names are suffixed with `Command` and written as `NamesLikeThisCommand`.

However, command names are automatically transformed from `NamesLikeThisCommand` to `name_like_this`. Thus, `ExampleCommand` would become `example`, and `AnotherExampleCommand` would become `another_example`.

In names for classes defining commands, use `NameLikeThisCommand`. To call a command from the API, use the standardized `name_like_this`.

11.6.3 Types of Commands

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

Instances of `WindowCommand` have a `.window` attribute pointing to the window instance that created them. Similarly, instances of `TextCommand` have a `.view` attribute.

Shared Traits for Commands

All commands must implement a `.run()` method.

All commands can receive an arbitrarily long number of keyword arguments that must be valid JSON types.

11.6.4 How to Call Commands from the API

Depending on the type of command, use a reference to a `View` or a `Window` and call `<object>.run_command('command_name')`. In addition to the command's name, `.run_command` accepts a dictionary whose keys are the names of valid parameters for said command:

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

11.6.5 Command Arguments

All user-provided arguments to commands must be valid JSON types.

11.6.6 Text Commands and the `edit` Object

Text commands receive and `edit` object passed to them by Sublime Text.

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.

Contrary to earlier versions of Sublime Text, the `edit` object's life time is now managed solely by the editor. Plugin authors must ensure to perform all editing operations within the `run()` method of text commands so that macros and repeating commands work as expected.

To call other commands from your own commands, use the `run_command()` function.

11.6.7 Responding to Events

Any subclass of `EventListener` will be able to respond to events. You cannot make a class derive both from `EventListener` and from any other type of command.

A Word of Warning about `EventListener`

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

11.6.8 Sublime Text and the Python Standard Library

Sublime Text ships with a trimmed down standard library.

11.6.9 Automatic Plugin Reload

Sublime Text will reload top-level Python modules as they change (perhaps because you are editing a *.py* file within *Packages*). By contrast, Python subpackages won't be reloaded automatically, and this can lead to confusion while you're developing plugins. Generally speaking, it's best to restart Sublime Text after you've made changes to plugin files, so all changes can take effect.

11.6.10 Multithreading

Only the `set_timeout()` function is safe to call from different threads.

11.7 Python API

See also:

Official Documentation API documentation.

11.7.1 Missing in the official docs

There are quite a few things that are not (yet) documented in the official docs, this section tries to solve this.

Index

`sublime` module

class `sublime.`**`Window`**

This class represents windows in Sublime Text and provides an interface of methods to interact with them. For all available methods, see the official documentation.

`set_layout`(*layout*)

Changes the tile-based panel layout of view groups.

Parameters **layout** (*dict*) – specifies the new layout, see below

Returns None

Expects a dictionary like this:

```
{"cols": [float], "rows": [float], "cells": [[int]]}
```

where `[type]` represents a list of *type*.

cols A list of the column separators (floating point numbers), should start with `0` (left) and end with `1` (right).

rows A list of the row separators (floating point numbers), should start with `0` (top) and end with `1` (bottom).

cells A list of cell lists which describe a cell's boundaries each. Cells can be imagines as rectangles with the rows and cols specified along in this dictionary. Think like this:

```
[x1, y1, x2, y2]
```

where all values are integers respectively and map to the *cols* and *rows* indicies. Thus, a cell with `[0, 0, 1, 2]` translates to a cell from the top left to the first column and the second row separator (in a 2x2 grid this would be bottom center).

Note: **rows** and **cols** are not tested for boundaries and they are not adjusted either. Thus, it is possible to specify values lower than `0` or higher than `1` and Sublime Text will in fact treat them accordingly. That means you can crop views or create borders. It is not known whether the "background color" of these empty spaces can be modified, the default is black. Use at your own risk!

The order of column or row separators is not checked either. If you, for example, use a reversed column list like `[1, 0.5, 0]` you get to see two black panels. Sublime Text is unusable in this state.

Examples:

```
# A 2-column layout with a separator in the middle
window.set_layout({
    "cols": [0, 0.5, 1],
    "rows": [0, 1],
    "cells": [[0, 0, 1, 1], [1, 0, 2, 1]]
})

# A 2x2 grid layout with all separators in the middle
window.set_layout({
    "cols": [0, 0.5, 1],
    "rows": [0, 0.5, 1],
    "cells": [[0, 0, 1, 1], [1, 0, 2, 1],
              [0, 1, 1, 2], [1, 1, 2, 2]]
})

# A 2-column layout with the separator in the middle and the right
# column being split in half
window.set_layout({
    "cols": [0, 0.5, 1],
    "rows": [0, 0.5, 1],
    "cells": [[0, 0, 1, 2], [1, 0, 2, 1],
                            [1, 1, 2, 2]]
})
```

class `sublime.`**`View`**

Similar to `Window`, this class represents views in Sublime Text and provides an interface of methods to interact with them. For all available methods, see the official documentation.

`match_selector`(*point*, *selector*)

Matches the scope at `point` against the specified `selector`.

Parameters

- **point** (*int*) – Point in the view whose scope the selector should be matched against.
- **selector** (*str*) – A scope selector.

Returns bool Whether the selector matches or not.

Equivalent to:

```
view.score_selector(point, selector) != 0
# or
sublime.score_selector(view.scope_name(point), selector) != 0
```

sublime_plugin module

class sublime_plugin.**EventListener**

on_query_completions(*view*, *prefix*, *locations*)
Called whenever the completion list is requested.

This accounts for all views and all windows, so in order to provide syntax-specific completions you should test the current scope of `locations` with `match_selector()`.

view A `View` instance for which the completions should be made.

prefix The text entered so far. This is only until the next word separator.

locations Array of points in `view` where the completion should be inserted. This can be interpreted as the current selection.

If you want to handle completions that depend on word separator characters you need to test each location individually. See *Completions with multiple cursors* on how Sublime Text handles completions with multiple cursors.

Return value Expects two (three) formats for return values:

1. `[[trigger, contents], ...]`

   A **list** of completions similar to *Trigger-based Completions* but without mapping keys. *trigger* may use the `\\t` description syntax.

   Note: In Sublime Text 3, completions may also consist of plain strings instead of the trigger-contents-list.

2. `([[trigger, contents], ...], flags)`

   Basically the same as above but wrapped in a 2-sized **tuple**. The second element, the *flags*, may be a bitwise OR combination of these flags:

   sublime.INHIBIT_WORD_COMPLETIONS Prevents Sublime Text from adding its word completions to the completion list after all plugins have been processed.

   sublime.INHIBIT_EXPLICIT_COMPLETIONS XXX What does this do?

   Flags are shared among all completions, once set by one plugin you can not revert them.

3. Anything else (e.g. `None`)

   No effect.

Example: See *Another Plugin Example: Feeding the Completions List* for an example on how to use this event.

11.7.2 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

11.8 Commands

11.8.1 Overview

This list of commands is a work in progress.

11.8.2 About Paths in Command Arguments

Some commands take paths as parameters. Among these, some support snippet-like syntax, while others don't. A command of the first kind would take a parameter like `$packages/SomeDir/SomeFile.ext` whereas a command of the second kind would take a parameter like `Packages/SomeDir/SomeFile.ext`.

Generally, newer commands support the snippet-like syntax.

Commands expect UNIX-style paths if not otherwise noted, including on Windows (for example, `/c/Program Files/Sublime Text 2/sublime_plugin.py`).

Often, relative paths in arguments to commands are assumed to start at the `Data` directory.

Variables in Paths as Arguments

The same variables available to build systems are expanded in arguments to commands. See *Build System Variables* for more information.

11.8.3 Commands

Note: This list is incomplete.

---

build Runs a build system.

- **variant** [String]: Optional. The name of the variant to be run.

set_build_system Changes the current build system.

- **file** [String]: Path to the build system. If empty, Sublime Text tries to automatically find an appropriate build systems from specified selectors.
- **index** [Int]: Used in the **Tools | Build System** menu but otherwise probably not useful.

new_build_system Creates a new buffer and inserts a build system template.

toggle_save_all_on_build Toggles whether all open files should be saved before starting the build.

run_macro_file Runs a *.sublime-macro* file.

- **file** [String]: Relative path to the macro file.

insert_snippet Inserts a snippet from a string or *.sublime-snippet* file.

- **contents** [String]: Snippet as a string to be inserted. Remember that backslashes `\` have to be escaped, like in every other JSON string.

- **name** [String]: Relative *path* to the *.sublime-snippet* file to be inserted.

See also:

Snippets Documentation on snippets and their variable features.

insert Inserts a string.

- **characters** [String]: String to be inserted.

move Advances the caret by predefined units.

- **by** [Enum]: Values: *characters*, *words*, *word_ends*, *subwords*, *subword_ends*, *lines*, *pages*, *stops*.
- **forward** [Bool]: Whether to advance or reverse in the buffer.
- **word_begin** [Bool]
- **empty_line** [Bool]
- **punct_begin** [Bool]
- **separators** [Bool]
- **extend** [Bool]: Whether to extend the selection. Defaults to `false`.

move_to Advances the caret to predefined locations.

- **to** [Enum]: Values: *bol*, *eol*, *bof*, *eof*, *brackets*.
- **extend** [Bool]: Whether to extend the selection. Defaults to `false`.

open_file Opens the specified file.

- **file** [String]: Absolute or relative *path* to the file to be opened. Relative paths will originate from the recently
- **contents** [String]: This string will be written to the new buffer if the file does not exist. accessed directory (e.g. the directory of the currently opened file).

open_dir Opens the specified directory with the default file manager.

- **dir** [String]: The directory to open.

open_file_settings Opens the syntax-specific user settings file for the current syntax.

new_window Opens a new window.

close_window Closes the active window.

switch_file Switches between two files with the same name and different extensions.

- **extensions** [String]: Extensions (without leading dot) for which switching will be enabled.

close Closes the active view.

close_file Closes the active view and, under certain circumsances, the whole application. XXX Sounds kinda wrong.

exit Exits the whole application with all open windows.

reopen_last_file Reopens the last closed file.

save Saves the active file.

- **encoding** [String]: The file encoding to save as.

prompt_save_as Prompts for a new file name and saves the active file.

save_project_as Prompts for a new file name and saves the current project.

prompt_select_project Opens a popup with recently accessed projects where you can fuzzy-search.

prompt_open_project Prompts for a project file to open as a project.

close_project Closes the current project.

prompt_add_folder Prompts for a folder to add to the current project.

close_folder_list Removes all folders from the current project.

refresh_folder_list Reloads all folders in the current project and updates the side bar.

toggle_sidebar Shows or hides the sidebar.

toggle_show_open_files Shows ot hides the open files in the sidebar.

toggle_status_bar Shows or hides the status bar.

toggle_full_screen Toggles full screen mode on or off.

toggle_distraction_free Toggles distraction free mode on or off.

toggle_tabs Shows or hides the tab bar.

toggle_minimap Shows or hides the minimap.

left_delete Deletes the character right before the caret.

right_delete Deletes the character right after the caret.

undo Undoes the latest action.

redo Reapplies the latest undone action.

redo_or_repeat Performs the latest action again.

soft_undo Undoes each action stepping through granular edits.

soft_redo Redoes each action stepping through granular edits.

cut Removes the selected text and sends it to the system clipboard. Put differently, it cuts.

copy Sends the selected text to to the system clipboard.

paste Inserts the clipboard contents after the caret.

- **clipboard** [String]: May be *selection*. XXX what other values are allowed?

paste_and_indent Inserts the clipboard contents after the caret and indents contextually.

select_lines Adds a line to the current selection.

- **forward** [Bool]: Whether to add the next or previous line. Defaults to `true`.

scroll_lines Scrolls lines in the view.

amount [Float]: Positive values scroll lines down and negative values scroll lines up.

prev_view Switches to the previous view.

next_view Switches to the next view.

next_view_in_stack Switches to the most recently active view.

previous_view_in_stack Switches to the view that was active before the most recently active view.

select_all Select the view's content.

split_selection_into_lines Splits the selection into multiple selections, one on each line.

single_selection Collapses multiple selections into a single selection.

clear_fields Breaks out of the active snippet field cycle.

hide_panel Hides the active panel.

- **cancel** [Bool]: Notifies the panel to restore the selection to what it

was when the panel was opened. (Only incremental find panel.)

hide_overlay Hides the active overlay. Show the overlay using the show_overlay command.

hide_auto_complete Hides the auto complete list.

insert_best_completion

Inserts the best completion that can be inferred from the current context.
XXX Probably useless. XXX

- **default** [String]: String to insert failing a best completion.

replace_completion_with_next_completion XXX Useless for users. XXX

reindent Corrects indentation of the selection with regular expressions set in the syntax's preferences. The base indentation will be that of the line before the first selected line. Sometimes does not work as expected.

indent Increments indentation of selection.

unindent Unindents selection.

detect_indentation Guesses the indentation from the current file.

next_field Advances the caret to the text snippet field in the current snippet field cycle.

prev_field Moves the caret to the previous snippet field in the current snippet field cycle.

commit_completion

Inserts into the buffer the item that's currently selected in the auto complete list.
XXX Probably not useful for users. XXX

toggle_overwrite Toggles overwriting on or off.

expand_selection Extends the selection up to predefined limits.

- **to** [Enum]: Values: *bol*, *hardbol*, *eol*, *hardeol*, *bof*, *eof*, *brackets*, *line*, *tag*, *scope*, *indentation*.

close_tag Surrounds the current inner text with the appropiate tags.

toggle_record_macro Starts or stops the macro recorder.

run_macro Runs the macro stored in the macro buffer.

save_macro Prompts for a fiel path to save the macro in the macro buffer to.

show_overlay Shows the requested overlay. Use the **hide_overlay** command to hide it.

- **overlay [Enum]:** The type of overlay to show. Possible values:
  - *goto*: Show the *Goto Anything* overlay.
  - *command_palette*: Show the *Command Palette*.
- **show_files** [Bool]: If using the goto overlay, start by displaying files rather than an empty widget.
- **text** [String]: The initial contents to put in the overlay.

show_panel Shows a panel.

- **panel** [Enum]: Values: *incremental_find*, *find*, *replace*, *find_in_files*, *console* or *output.<panel_name>*.
- **reverse** [Bool]: Whether to search backwards in the buffer.
- **toggle** [Bool]: Whether to hide the panel if it's already visible.

find_next Finds the next occurrence of the current search term.

find_prev Finds the previous occurrence of the current search term.

find_under_expand Adds a new selection based on the current selection or expands the selection to the current word.

find_under_expand_skip Adds a new selection based on the current selection or expands the selection to the current word while removing the current selection.

find_under Finds the next occurrence of the current selection or the current word.

find_under_prev Finds the previous occurrence of the current selection or the current word.

find_all_under Finds all occurrences of the current selection or the current word.

slurp_find_string Copies the current selection or word into the "find" field of the find panel.

slurp_replace_string Copies the current selection or word into the "replace" field of the find and replace panel.

next_result Advance to the next captured result.

prev_result Move to the previous captured result.

toggle_setting Toggles the value of a boolean setting. This value is view-specific.

- **setting** [String]: The name of the setting to be toggled.

set_setting Set the value of a setting. This value is view-specific.

- **setting** [String]: The name of the setting to changed.
- **value** [*]: The value to set to.

set_line_ending Changes the line endings of the current file.

- **type** [Enum]: *windows*, *unix*, *cr*

next_misspelling Advance to the next misspelling

prev_misspelling Move to the previous misspelling.

swap_line_down Swaps the current line with the line below.

swap_line_up Swaps the current line with the line above.

toggle_comment Comments or uncomments the active lines, if available.

- **block** [Bool]: Whether to insert a block comment.

join_lines Joins the current line with the next one.

duplicate_line Duplicates the current line.

auto_complete Opens the auto complete list.

replace_completion_with_auto_complete XXX Useless for users. XXX

show_scope_name Shows the name for the caret's scope in the status bar.

exec Runs an external process asynchronously. On Windows, GUIs are supressed.

`exec` is the default command used by build systems, thus it provides similar functionality. However, a few options in build systems are taken care of by Sublime Text internally so they list below only contains parameters accepted by this command.

- **cmd** [[String]]
- **file_regex** [String]
- **line_regex** [String]

- **working_dir** [String]
- **encoding** [String]
- **env** [{String: String}]
- **path** [String]
- **shell** [Bool]
- **kill** [Bool]: If `True` will simply terminate the current build process. This is invoked via *Build: Cancel* command from the *Command Palette*.
- **quiet** [Bool]: If `True` information less running about prints the command.

See also:

Arbitrary Options for build systems Detailed documentation on all other available options.

transpose Makes selections or characters swap places.

With selection: The contents of the selected regions are circulated.
Without selection: Swaps adjacent characters and moves the caret forward by 1.

sort_lines Sorts lines.

- **case_sensitive** [Bool]: Whether the sort should be case sensitive.

sort_selection Sorts lines in selection.

- **case_sensitive** [Bool]: Whether the sort should be case sensitive.

permute_lines XXX

- **operation** [Enum]: *reverse*, *unique*, *shuffle* ...?

permute_selection XXX

- **operation** [Enum]: *reverse*, *unique*, *shuffle* ...?

set_layout Changes the group layout of the current window. This command uses the same pattern as `Window.set_layout()`, see there for a list and explanation of parameters.

focus_group Gives focus to the top-most file in the specified group.

- **group** [Int]: The group index to focus. This is determined by the order of `cells` items from the current layout (see `Window.set_layout()`).

move_to_group Moves the current file to the specified group.

- **group** [Int]: The group index to focus. See **focus_group** command.

select_by_index Focuses a certain tab in the current group.

- **index** [Int]: The tab index to focus.

next_bookmark Select the next bookmarked region.

prev_bookmark Select the previous bookmarked region.

toggle_bookmark Sets or unsets a bookmark for the active region(s). (Bookmarks can be accessed via the regions API using `"bookmarks"` as the key.)

select_bookmark Selects a bookmarked region in the current file.

- **index** [Int]

clear_bookmarks Removes all bookmarks.

select_all_bookmarks Selects all bookmarked regions.

wrap_lines Wraps lines. By default, it wraps lines at the first ruler's column.

- **width** [Int]: Specifies the column at which lines should be wrapped.

upper_case Makes the selection upper case.

lower_case Makes the selection lower case.

title_case Capitalizes the selection's first character and turns the rest into lower case.

swap_case Swaps the case of each character in the selection.

set_mark XXX

select_to_mark XXX

delete_to_mark XXX

swap_with_mark XXX

clear_bookmarks XXX

- **name** [String]: e.g. `"mark"`.

yank XXX

show_at_center Scrolls the view to show the selected line in the middle of the view and adjusts the horizontal scrolling if necessary.

increase_font_size Increases the font size.

decrease_font_size Decreases the font size.

reset_font_size Resets the font size to the default

Note: This essentially removes the entry from your User settings, there might be other places where this has been "changed".

fold Folds the current selection and displays `...` instead. Unfold arrows are added to the lines where a region has been folded.

unfold Unfolds all folded regions in the selection or the current line if there is none.

fold_by_level Scans the whole file and folds everything with an indentation level of `level` or higher. This does not unfold already folded regions if you first fold by level 2 and then by 3, for example. Sections with cursors are not folded.

- **level** [Int]: The level of indentation that should be folded. `0` is equivalent to running **unfold_all**.

fold_tag_attributes Folds all tag attributes in XML files, only leaving the tag's name and the closing bracket visible.

unfold_all Unfolds all folded regions.

context_menu Shows the context menu.

open_recent_file Opens a recently closed file.

- **index** [Int]

open_recent_folder Opens a recently closed folder.

- **index** [Int]

open_recent_project Opens a recently closed project.

- **index** [Int]

clear_recent_files Deletes records of recently accessed files and folders.

clear_recent_projects Deletes records of recently accessed projects.

reopen Reopens the current file.

- **encoding** [String]: The file encoding the file should be reopened with.

clone_file Clones the current view into the same tab group, both sharing the same buffer. That means you can drag one tab to another group and every update to one view will be visible in the other one too.

revert Undoes all unsaved changes to the file.

expand_tabs XXX

- **set_translate_tabs** [Bool]

unexpand_tabs XXX

- **set_translate_tabs** [Bool]

new_plugin Creates a new buffer and inserts a plugin template (a text command).

new_snippet Creates a new buffer and inserts a snippet template.

open_url Opens the specified url with the default browser.

- **url** [String]

show_about_window I think you know what this does.

11.8.4 Discovering Commands

There are several ways to discover a command's name in order to use it as a key binding, in a macro, as a menu entry or in a plugin.

- Browsing the default key bindings at **Preferences | Key Bindings - Default**. If you know the key binding whose command you want to inspect you can just search for it using the *search panel*. This, of course, also works in the opposite direction.
- ``sublime.log_commands(True)``

  Running the above in the console will tell Sublime Text to print the command's name in the console whenever a command is run. You can practically just enter this, do whatever is needed to run the command you want to inspect and then look at the console. It will also print the passed arguments so you can basically get all the information you need from it. When you are done, just run the function again with `False` as parameter.
- Inspecting *.sublime-menu* files. If your command is run by a menu item, browse the default menu file at `Packages/Default/Main.sublime-menu`. You will find them quick enough once you take a look at it, or see the *menu documentation*.
- Similar to menus you can do exactly the same with *.sublime-command* files. See *Completions* for some documentation on completion files.

11.9 Keyboard Shortcuts - Windows/Linux

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

11.9.1 Editing

Windows

Ctrl + Alt + Up	Column selection up
Ctrl + Alt + Down	Column selection down

Linux

Alt + + Up	Column selection up
Alt + + Down	Column selection down

11.9.2 Navigation/Goto Anywhere

Keypress	Command
Ctrl + P	Quick-open files by name
Ctrl + R	Goto symbol
Ctrl + ;	Goto word in current file
Ctrl + G	Goto line in current file

11.9.3 General

Keypress	Command
Ctrl + + P	Command prompt
Ctrl + KB	Toggle side bar
Ctrl + + Alt + P	Show scope in status bar

11.9.4 Find/Replace

Keypress	Command
Ctrl + F	Find
Ctrl + H	Replace
Ctrl + + F	Find in files

11.9.5 Tabs

Keypress	Command
Ctrl + + t	Open last closed tab
Ctrl + PgUp	Cycle up through tabs
Ctrl + PgDn	Cycle down through tabs
Ctrl +	Find in files
Ctrl + W	Close current tab
Alt + [NUM]	Switch to tab number [NUM] where [NUM] <= number of tabs

11.9.6 Split window

Keypress	Command
Alt + + 2	Split view into two columns
Alt + + 1	Revert view to single column
Alt + + 5	Set view to grid (4 groups)
Ctrl + [NUM]	Jump to group where num is 1-4
Ctrl + + [NUM]	Move file to specified group where num is 1-4

11.9.7 Bookmarks

Keypress	Command
Ctrl + F2	Toggle bookmark
F2	Next bookmark
+ F2	Previous bookmark
Ctrl + + F2	Clear bookmarks

11.9.8 Text manipulation

Keypress	Command
Ctrl + KU	Transform to Uppercase
Ctrl + KL	Transform to Lowercase

11.10 Keyboard Shortcuts - OSX

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

11.10.1 Editing

Keypress	Command
+ X	Delete line
+	Insert line after
+ +	Insert line before
+ + ↑	Move line/selection up
+ + ↓	Move line/selection down
+ L	Select line - Repeat to select next lines
+ D	Select word - Repeat select others occurrences
+ + ↑	Extra cursor on the line above
+ + ↓	Extra cursor on the line below
+ M	Jump to closing parentheses Repeat to jump to opening parentheses
+ + M	Select all contents of the current parentheses
+ K, + K	Delete from cursor to end of line
+ K +	Delete from cursor to start of line
+]	Indent current line(s)
+ [	Un-indent current line(s)
+ + D	Duplicate line(s)
+ J	Join line below to the end of the current line
+ /	Comment/un-comment current line
+ + /	Block comment current selection
+ Y	Redo, or repeat last keyboard shortcut command
+ + V	Paste and indent correctly
+ Space	Select next auto-complete suggestion
+ U	Soft undo; jumps to your last change before undoing change when repeated
+ + Up	Column selection up
+ + Down	Column selection down
+ + W	Wrap Selection in html tag

11.10.2 Navigation/Goto Anywhere

Keypress	Command
+ P	Quick-open files by name
+ R	Goto symbol
	Goto word in current file
+ G	Goto line in current file

11.10.3 General

Keypress	Command
+ + P	Command prompt
+ K, + B	Toggle side bar
+ + P	Show scope in status bar

11.10.4 Find/Replace

Keypress	Command
+ F	Find
+ + F	Replace
+ + F	Find in files

11.10.5 Tabs

Keypress	Command
+ + t	Open last closed tab
^ + Tab	Cycle up through tabs
+ ^ + Tab	Cycle down through tabs
	Find in files

11.10.6 Split window

Keypress	Command
+ + 2	Split view into two columns
+ + 1	Revert view to single column
+ + 5	Set view to grid (4 groups)
+ [NUM]	Jump to group where num is 1-4
+ + [NUM]	Move file to specified group where num is 1-4

11.10.7 Bookmarks

Keypress	Command
+ F2	Toggle bookmark
F2	Next bookmark
+ F2	Previous bookmark
+ + F2	Clear bookmarks

11.10.8 Text manipulation

CHAPTER 12

Glossary

buffer Data of a loaded file and additional metadata, associated with one or more views. The distinction between *buffer* and *view* is technical. Most of the time, both terms can be used interchangeably.

view Graphical display of a buffer. Multiple views can show the same buffer.

plugin A feature impemented in Python, which can consist of a single command or multiple commands. It can be contained in one *.py* file or many *.py* files.

package This term is ambiguous in the context of Sublime Text, because it can refer to a Python package (unlikely), a folder inside `Packages` or a *.sublime-package* file. Most of the time, it means a folder inside `Packages` containing resources that belong together, which build a new feature or provide support for a programming or markup language.

panel An input/output widget, such as a search panel or the output panel.

overlay An input widget of a special kind. For example, Goto Anything is an overlay.

S

B

E

M

O

P

S

V

W