Sublime Text Unofficial Documentation

Release 2.0

guillermooo

July 04, 2016

Contents

Warning: Development of Sublime Text has moved on to version 3.

As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

About This Documentation

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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

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!

Before you continue, we encourage you to read through the Basic Concepts section.

Happy learning!

1.1 Contributing to the Documentation

If you are known to Sublime Text and want to contribute to this documentation, head over to the github repo. We use Sphinx to create these pages.

Furthermore, for every individual page in this documentation there are three github-related links in the left navigation column. Pick one appropriate to your needs.

Installation

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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 2 bundle into the *Applications* folder.

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\ 2.0.1.tar.bz2
tar vxjf Sublime\ Text\ 2.0.1.tar.bz2
```

For x64

```
cd ~
wget http://c758482.r82.cf2.rackcdn.com/Sublime Text 2.0.1 x64.tar.bz2
tar vxjf Sublime\ Text\ 2.0.1\ x64.tar.bz2
```

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

```
sudo mv Sublime\ Text\ 2 /opt/
```

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

```
sudo ln -s /opt/Sublime\ Text\ 2/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=2.0.1
Name=Sublime Text 2
# 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 2/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-2
```

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 users who own a licence:

- 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.

Lastly, **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. Nighlty builds are **only available for registered users**.

Basic Concepts

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

Here we'll explain concepts that the reader needs to be familiar with in order to fully understand the contents of this guide.

3.1 Conventions in This Guide

This guide is written from the perspective of a Windows user, but most instructions should require only 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. Due to the way Sublime Text maps keys to commands, **some key bindings won't match your locale's keyboard layout**.

3.2 With Great Power Come Many Questions

Sublime Text is a very extensible and customizable editor. It does many things out of the box, but if you spend some time tailoring it to your exact needs, it will give you superpowers. This guide will teach you all you need to know to configure Sublime Text.

In the following paragraphs, we'll outline some aspects that won't click in your mind until you've spent some time using Sublime Text. Keep exploring the editor and looking around in this guide, and everything will fall into place at some point.

Sublime Text is certainly a versatile tool for programmers, but you don't need to be one to use it, or even to configure it to make it the perfect tool for your writing. If you're a hacker, however, you are about to spend the remainder of your day playing around with this editor.

3.3 The *Data* Directory

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

- **Windows**: `%APPDATA%\Sublime Text 2`
- **OS X**: `~/Library/Application Support/Sublime Text 2`
- **Linux**: `~/.config/sublime-text-2`

For **portable installations**, look inside `Sublime Text 2/Data`. Here, the *Sublime Text 2* part refers to the directory to which you've extracted the contents of the compressed file containing Sublime Text 2.

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

3.4 The *Packages* Directory

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

You can access the packages directory from the Sublime Text 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*.

3.4.1 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.

3.5 The Python Console and the Python API

This information is especially interesting for programmers. For the rest of Sublime Text users, you just need to know that it 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 an useful tool to inspect Sublime Text settings and to quickly test API calls while you're writing plugins.

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

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 is prepared to be programmed by the user. Put differently, Sublime Text gives the user access to its internals through Python. Lastly, a *console* is a little window inside Sublime Text which lets you type in short snippets of Python code and run them. The console also shows text output by Sublime Text or its plugins.

3.5.1 Your System's Python vs the Sublime Text Embedded Python

On **Windows** and **Linux**, Sublime Text comes with its own Python interpreter and it's separate from your system's Python installation.

On **OS X**, the system Python is used instead. Modifying your system version of Python, such as replacing it with the MacPorts version, can cause problems for Sublime Text.

The embedded interpreter is intended only to interact with the plugin API, not for general development. A few plugins may run into issues because the embedded or used interpreters are not the same on every OS.

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

For now, just keep in mind that almost everything in Sublime Text can be adapted to your needs. This vast flexibility is the reason why you will learn about so many settings files: there simply must be a place to specify all your preferences.

Configuration files in Sublime Text let you change the editor's behavior, add macros, snippets or create new features –where *feature* means 'anything you can think of'. OK, maybe not *anything*, but Sublime Text definitely hands you over a good deal of control.

These settings files simply are text files following a special structure or *format*: JSON predominates, but you'll find XML files too.

In this guide, we refer collectively to all these disparate configuration files as *resources*. Sublime Text will look for resources inside the packages directory. To keep things tidy, the editor has a notion of a *package*, which is a directory containing resources that belong together (maybe they all help write emails faster or code in a certain programming language).

3.7 Textmate Compatibility

This information is mainly useful for Textmate expats who are now using Sublime Text. Textmate is an editor for the Mac.

Sublime Text is fairly compatible with Textmate bundles with the notable exception of commands. 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.

3.8 Vi 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.

3.9 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.

3.10 Be Sublime, My Friend

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

Empty your mind; be sublime, my friend.

Editing

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

4.1 Overview

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

4.2 Column Selection

Column Selection can be used to select a rectangular area of a file. Column selection doesn't operate via a separate mode, instead it makes use of multiple selections.

You can use additive selections to select multiple blocks of text, or subtractive selections to remove a block.

4.2.1 Using the Mouse

Windows

- Right Mouse Button +
- OR: Middle Mouse Button
- Add to selection: `Ctrl`
- Subtract from selection: `Alt`

Linux

- Right Mouse Button +
- Add to selection: `Ctrl`
- Subtract from selection: `Alt`

OS X

- Left Mouse Button +
- OR: Middle Mouse Button

- Add to selection:
- Subtract from selection: +

4.2.2 Using the Keyboard

Windows: `Ctrl + Alt + Up` and `Ctrl + Alt + Down`

Linux: `Alt +  + Up` and `Alt +  + Down`

OS X: ` +  + Up` and ` +  + Down`

4.3 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.

4.4 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.

4.5 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 when you need them... warning:

Search and Replace

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

Sublime Text features two main types of search:

5.1 Search and Replace - Single File

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

5.1.1 Searching

To open the **search panel** for buffers, press `Ctrl + F`. Some options in the search panel and search actions can be controlled with 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 buffer 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

You can open the replace planel with `Ctrl + H`.

Replace All:	Ctrl + Alt + Enter

5.1.4 Tips

Other Ways of Searching in Buffers

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

Other Key Bindings to Search in Buffers

These keybindings 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 a multiline search pattern. To enter a newline character, press Ctrl + Enter in the search panel. Note that the search panel is resizable too.

5.2 Search and Replace - Multiple Files

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the latest branch in the panel on the bottom left and consider updating Sublime Text.

5.2.1 Searching

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

Toggle Regular Expressions	Alt + R
Toggle Case Sensitivity	Alt + C
Toggle Exact matches	Alt + W
Find Next	Enter

5.2.2 Search Scope

The **Where** field in the search panel determines where to search. You can define the scope of the search 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>)

You can combine these filters separing them with 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.

5.2.3 Results Format

In the search panel, you can find the following options to customize the results format:

- Show in Separate Buffer/Output Panel
- Show Context

5.2.4 Navigating Results

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

Next match	F4
Previous match	Shift + F4

We'll examine them in turn, but let's talk about a powerful tool for searching text first: **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 learn at least the basics of regular expressions. In this guide we will not explain how to use regular expressions.

Typing out *regular expression* gets boring fast, and saying it actually is even more annoying, so instead nerds usually shorten that to *regexp* or *regex*.

This is how a regex might look like:

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

To use regular expressions, you need to activate them first in the various search panels. Othwerwise, the search term will 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.

Build Systems (Batch Processing)

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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**.

File Navigation and File Management

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

7.1 Goto Anything

Goto Anything lets you **navigate files** swiftly. Open it with `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 buffer until you perform some operation on it. Transient views go away when you press `Esc`. You will find transient views in other situations, e. g. 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 directives

There are a few special directives for Goto Anything which will point you to other places than just the beginning of a file. Any of these directives can be used in combination with file search queries and will be applied on the currently selected file or on the file you are currently editing if you haven't specified any filename search term.

Directives are invoked with a special character, e. g. `:`, and all text after that will be interpreted by the directive. Example:

```
island:123
```

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

Here is a list of the supported directives:

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

> Symbols usually are classes or functions but can be anything defined by the syntax definition. See *Symbols - Syntax Preferences* (XXX to be added). In return, they might not be defined at all and searching for symbols will fail in this case.

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

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

7.2 Sidebar

The sidebar gives you an overview of your project. Files and folders added to the sidebar will be available in *Goto Anything* and project-wide actions. Projects and the sidebar are closely related. There's always an open project, whether it's implicit or explicit.

To **open or close** the sidebar, press `Ctrl+K, Ctrl+B`.

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

The sidebar also provides basic file management operations through the context menu.

7.3 Projects

Projects group sets of files and directories you need to work on as a unit. Once you've set up your project the way that suits you by adding folders, save it and give it a name. Project files use the *.sublime-project* extension. You can add and remove folders to a project with the **Project** menu and the side bar's context menu. Futhermore, you can drag folders onto a window and they will be added automatically.

To save a project, choose **Project | Save Project As...**.

To quickly switch between projects, press `Ctrl+Alt+P`. Alernatively you can browse **Projects | Recent Projects**.

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

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 you shouldn't edit it yourself.

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

```
{
    "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` setting, to set how they're displayed on the side bar.

Settings A project may define project-specific settings which only apply to (open) files within that project. Project-specific settings override regular user settings but not syntax-specific settings.

You can override almost all settings (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...

Customizing Sublime Text

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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

8.1 Settings

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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 that they will take precedence over any other conflicting settings files.

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

8.1.1 Format

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

8.1.2 Types of Settings

The purpose of each *.sublime-settings* file is determined by its name. These 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 *.tm-Language* 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 for 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 might 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. You should not edit **Preferences | Settings - Default**, 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 a setting's current value for a particular file, use `view.settings().get("`*setting_name*`")` from the console.

Lastly, it's also worth noting that some settings may be adjusted automatically 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.

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 file:*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 `Packages/Python`, but there can be multiple settings files for the same file type in separate locations.

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

Also, 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 the same filet type.

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

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

See also:

Indentation Official Sublime Text Documentation.

8.3 Key Bindings

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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

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

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 to pass arguments and provide contextual awareness.

Passing Arguments

Arguments are specified in the `args` key:

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

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

Contexts

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

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

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

8.4 Menus

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the latest branch in the panel on the bottom left and consider updating Sublime Text.

No documenation available about this topic yet.

But here's Bruce Lee screaming...

Extending Sublime Text

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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

9.1 Commands

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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, like `hot_exit` and can take a dictionary of arguments whose keys must be strings and whose values must be JSON types. Here's 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

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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

Macro files are JSON files with the *.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 `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. You will now be able to run the recorded macro by pressing `Ctrl+Shift+q` or save it to a file by selecting **Tools | Save macro...**.

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

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 extension `.sublime-macro` under `PackagesUser` 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 they 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

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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, adapting it to their context.

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

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

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 document. 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. Sublime Text automatically sets the values of the variables listed below.

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

$PARAM1, $PARAM2...	Arguments passed to the `insert_snippet` command. (Not covered here.)
$SELECTION	The text that was selected when the snippet was triggered.
$TM_CURRENT_LINE	Content of the 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, enabling you to resume normal editing.

If you want to control where the exit point should be, use the `$0` mark. By default, the exit point 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 still 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 placeholder 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 snippet.

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: =========
```

9.4 Completions

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

See also:

Sublime Text Documentation Official documentation on this topic.

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

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

- Snippets

- API-injected completions

- Buffer contents

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

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.

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>" },
                { "trigger": "script\t<script src=\"...\" />",
                  "contents": "<script src=\"$1\" />" },

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

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

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`), and
- 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\tname*.

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 prefix.

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

Inserting a Literal Tab Character

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

9.5 Command Palette

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in
> the panel on the bottom left and consider updating Sublime Text.

See also:

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

9.5.1 Overview

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

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, so whenever you open it, you won't always see all the commands defined in every `.sublime-commands` file.

9.6 Syntax Definitions

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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. Follow the installation notes in the "Getting Started" section of the readme.

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 YAML 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 or YAML 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, many XML tags 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. Especially Color Schemes make excessive usage of scopes to style every aspect of a language in the desired color.

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. Rules are applied in the following order:

1. The rule that matches at the first position in a line
2. The rule that comes first in the array

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 whole `.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 (`$`, `<`...)

Here are the elements we don't want to style because they are too complex for this example:

- Variable Substitution (`${1/Hello/Hi/g}`)

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 `.YAML-tmLanguage` file.

You now should see a file like this:

```
# [PackageDev] target_format: plist, ext: tmLanguage
---
name: Syntax Name
scopeName: source.syntax_name
fileTypes: []
uuid: 0da65be4-5aac-4b6f-8071-1aadb970b8d9

patterns:
-
...
```

Let's examine the key elements.

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 (without the leading dot). When opening files of these types, Sublime Text will automatically activate this syntax definition for them.

uuid This is a unique identifier for this syntax definition. Each new syntax definition gets its own uuid. Even though Sublime Text itself ignores it, don't modify this.

patterns A container for your patterns.

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

```
# [PackageDev] target_format: plist, ext: tmLanguage
---
name: Sublime Snippet (Raw)
scopeName: source.ssraw
fileTypes: [ssraw]
uuid: 0da65be4-5aac-4b6f-8071-1aadb970b8d9

patterns:
-
...
```

Note: YAML is not a very strict format, but can cause headaches when you don't know its conventions. It supports single and double quotes, but you may also omit them as long as the content does not create another YAML literal. 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 YAML to Plist. This will also cover the first commented line in the template.

The `---` and `...` are optional.

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.

Matches

Matches take this form:

```
match: (?i:m)y \s+[Rr]egex
name: string.format
comment: This comment is optional.
```

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.

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. It looks like this:

```
# [PackageDev] target_format: plist, ext: tmLanguage
---
name: Sublime Snippet (Raw)
scopeName: source.ssraw
fileTypes: [ssraw]
uuid: 0da65be4-5aac-4b6f-8071-1aadb970b8d9

patterns:
-
...
```

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+
```

We can then build our pattern like this:

```
name: keyword.other.ssraw
match: \$\d+
comment: Tab stops like $1, $2...
```

> **Choosing the Right Scope Name**
>
> Naming scopes isn't obvious sometimes. Check the Textmate naming conventions for guidance on scope names. AAAPackageDev automatically provides completions for scope names according to these conventions. It is important to re-use the basic categories outlined there if you want to achieve the highest compatibility with existing colors.
> Color schemes 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 (like for CSS or Markdown). This means that two color schemes 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:

```
# [PackageDev] target_format: plist, ext: tmLanguage
---
name: Sublime Snippet (Raw)
scopeName: source.ssraw
fileTypes: [ssraw]
uuid: 0da65be4-5aac-4b6f-8071-1aadb970b8d9

patterns:
- comment: Tab stops like $1, $2...
  name: keyword.other.ssraw
  match: \$\d+
...
```

Note: You should use two spaces for indent. This is the recommended indent for YAML and lines up with lists like shown above.

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 Plist XML files.

Follow these steps to perform the conversion:

- Make sure that `Automatic` is selected in **Tools | Build System**, or select `Convert to ...`

- Press `F7`

- A `.tmLanguage` file will be generated for you in the same folder as your `.YAML-tmLanguage` file

- Sublime Text will reload the changes to the syntax definition

In case you are wondering why AAAPackageDev knows what you want to convert your file to: It's specified in the first commente line.

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.

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

Repeat the above steps to update the `.tmLanguage` file.

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`:

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

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.

Note: Writing 1 on a new line and pressing tab will autocomplete to `'1':  {name:  }` thanks to AAAPackageDev.

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

Note: As with ususal regular expressions and substítutions, the capture group `'0'` applies to the whole match.

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:
contentName:
begin:
beginCaptures:
  '0': {name: }
  # ...
end:
endCaptures:
  '0': {name: }
  # ...
patterns:
- name:
  match:
# ...
```

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

name Just like with simple captures this sets the following scope name to the whole match, including `begin` and `end` marks. Effectively, this will create nested scopes for `beginCaptures`, `endCaptures` and `patterns` defined within this rule. Optional.

contentName Unlike the `name` this only applies a scope name to the enclosed text. Optional.

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.

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. Optional.

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

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

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 `}`. We need to wrap the begin pattern into quotes because otherwise the trailing `:` would indicate the parser to expect another dictionary key. `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** to 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 and can't be matched again.

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. Note that this doesn't work if we made the pattern greedy (`.+`) because this includes possible nested references.

Note: We could've used `contentName:   string.other.ssraw` instead of the last pattern but this way we introduce the importance of ordering and how matches are consumed.

Final Touches

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

```
- comment: Sequences like \$, \> and \<
  name: constant.character.escape.ssraw
  match: \\[$<>]

- comment: Unescaped and unmatched magic characters
  name: invalid.illegal.ssraw
  match: '[$<>]'
```

The only hard thing here is not forgetting that `[]` enclose arrays in YAML and thus must be wrapped in quotes. 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 it will be consumed and result in every following expression not matching.

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:

```
# [PackageDev] target_format: plist, ext: tmLanguage
---
name: Sublime Snippet (Raw)
scopeName: source.ssraw
fileTypes: [ssraw]
uuid: 0da65be4-5aac-4b6f-8071-1aadb970b8d9

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

- comment: Variables like $PARAM1, $TM_SELECTION...
  name: keyword.other.ssraw
  match: \$([A-Za-z][A-Za-z0-9_]+)
```

```
  captures:
    '1': {name: constant.numeric.ssraw}

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

- comment: Sequences like \$, \> and \<
  name: constant.character.escape.ssraw
  match: \\[$<>]

- comment: Unescaped and unmatched magic characters
  name: invalid.illegal.ssraw
  match: '[$<>]'
...
```

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

Note: If you previously used JSON for syntax definitions you are still able to do this because AAAPackageDev is backwards compatible.

If you want to consider switching to YAML (either from JSON or directly from Plist), it provides a command named `AAAPackageDev:  Convert to YAML and Rearrange Syntax Definition` which will automatically format the resulting YAML in a pleasurable way.

See also:

Syntax Definitions Reference for snytax definitions

9.7 Plugins

Warning: Development of Sublime Text has moved on to version 3.
As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

See also:

API Reference More information on the Python API.

Plugins Reference More information about plugins.

Sublime Text 2 is programmable with Python scripts. Plugins reuse existing commands or create new ones to build a feature. Plugins are 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.

9.7.2 Where to Store Plugins

Sublime Text 2 will only look for plugins in these places:

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

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

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

9.7.3 Your First Plugin

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

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

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

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

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

9.7.4 Analyzing Your First Plugin

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

```python
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 2.

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

The rest of the code is concerned with the 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 rather an inconvenient way of using plugins, 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 both through key bindings and by other means, just like other commands.

Conventions for Command Names

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

New commands should follow the CamelCase pattern for class names.

9.7.5 Types of Commands

You can create the following types of commands:

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

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

Shared Traits of Commands

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

Application Commands

Application commands derive from `sublime_plugin.ApplicationCommand` and can be executed with `sublime.run_command()`.

Window Commands

Window commands operate at the window level. This doesn't mean 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 does not take any required arguments.

Text Commands

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

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

The `.run()` method of a text command needs to accept an `edit` instance as the first positional argument.

Text Commands and the `edit` Object

The edit object groups any modifications to the view so as to enable undo and macros to work sensibly.

You are responsible for creating and closing edit objects. To do so, you can call `view.begin_edit()` and `edit.end_edit()`. For convenience, the currently open `edit` object gets passed to text commands' `run` method automatically. Additionally, many `View` methods require an edit object.

Responding to Events

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

Another Plugin Example: Feeding the Completions List

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

```python
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 too. In particular, the `Packages/Default` folder contains many examples of undocumented commands and API calls.

9.8 Packages

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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

Here's a list of the typical resources living 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 `PackagesLanguage - English` as a data store for English dictionaries.

9.8.1 Types of Packages

In this guide, in order to talk about packages, we divide them into groups. This division is artificial, and just useful for clarity when discussing this topic. Sublime Text doesn't use this division in any way.

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

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

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

installed packages Any package that, if deleted, Sublime Text will be able to restore.

Let's emphasize again that you don't need to memorize this classification.

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

9.8.2 Installation of Packages

There are two main ways to install packages:

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

Ultimately, installing a package is simply a matter of copying a folder containing Sublime Text resources to :file:`Packages`. The only thing that changes from one system to another is how you copy these files.

Installing Packages vs Installed Packages

Note that "installing a package" actually doesn't make that package a Sublime Text installed package. *Installed packages* are *.sublime-package* files residing in the `Installed Packages` folder. In this guide, we use *install a package* to mean copying a package to :file:`Packages`.

Sublime Text can restore any package located in `Installed Packages`, but can't automatically restore the packages 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 Google Code, 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 quite simply, 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, as mentioned at the beginning, 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` also 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 folders `Default` and `User`. Files contained in `Default` will always go to the front of the list and, those in `User`, to the end.

9.8.4 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, though, 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.

9.8.5 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`.

9.8.6 The `Pristine Packages` Directory

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

Command Line Usage

Warning: Development of Sublime Text has moved on to version 3.

As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

See also:

OS X Command Line Official Sublime Text Documentation

Reference

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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

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

11.1 Syntax Definitions

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

> **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, YAML is shown instead.

Additionally, Sublime Text also understands the `hidden-tmLanguage` extension, which can not be selected by the user but only by set by plugins. "Find in Files" makes use of this. The downsite is that these can not be included by import statements in other language definitions.

```
---
name: Sublime Snippet (Raw)
scopeName: source.ssraw
fileTypes: [ssraw]
uuid: 0da65be4-5aac-4b6f-8071-1aadb970b8d9
```

```
patterns:
- comment: Tab stops like $1, $2...
  name: keyword.other.ssraw
  match: \$\d+

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

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

- name: constant.character.escape.ssraw
  match: \\[$<>]

- name: invalid.illegal.ssraw
  match: '[$<>]'
...
```

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 markup and everything else.

fileTypes This is a list of file extensions (without the leading dot). When opening files of these types, Sublime Text will automatically activate this syntax definition for them. Optional.

uuid Unique indentifier for this syntax definition. Currently ignored.

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 or re-using the same pattern. Optional.

11.1.3 The Patterns Array

Elements contained in the `patterns` array.

match Contains the following elements:

match	Pattern to search for.
name	Optional. 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 (note that `0` refers to the whole match):

0..n	Name of the group referenced. Must be a string.
name	Scope to be assigned to the group.

Examples:

```
# Simple

- comment: Sequences like \$, \> and \<
  name: constant.character.escape.ssraw
  match: \\[$<>]

# With captures

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

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 complete 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 (only begin and end are required):

name	Scope name for the content including the markers.
contentName	Scope name for the content excluding the markers.
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	Array of patterns to be matched against the content.

Example:

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

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:

```
repository:

  # Simple elements
  elementName:
    match: some regexp
    name:  some.scope.somelang

  # Complex elements
  otherElementName:
    patterns:
    - match: some regexp
      name:  some.scope.somelang
    - match: other regexp
      name:  some.other.scope.somelang
```

Examples:

```
repository:
  numericConstant:
    patterns:
    - name: constant.numeric.double.powershell
      match: \d*(?<!\.)(\.)\d+(d)?(mb|kb|gb)?
      captures:
        '1': {name: support.constant.powershell}
        '2': {name: support.constant.powershell}
        '3': {name: keyword.other.powershell}
    - name: constant.numeric.powershell
      match: (?<!\w)\d+(d)?(mb|kb|gb)?(?!\w)
      captures:
        '1': {name: support.constant.powershell}
        '2': {name: keyword.other.powershell}

  scriptblock:
    name: meta.scriptblock.powershell
    begin: \{
    end: \}
    patterns:
    - include: $self
```

11.1.5 Escape Sequences

Be sure to escape JSON/XML sequences as needed.

For YAML, additionally make sure that you didn't unintentionally start a new scalar by not using quotes for your strings. Examples that **won't work** as expected:

```
match: [aeiou]

include: #this-is-actually-a-comment

match: "#"\w+""
```

11.2 Build Systems

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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

- optionally, an external executable file (script, 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 explained 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"
}
```

11.2.2 Options

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's `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.

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.

11.2.3 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.

Place Holders for Variables

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

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

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

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

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

See also:

Snippets Documentation on snippets and their variable features.

11.2.4 Running Build Systems

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

11.2.5 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 for `PATH` will vary from a terminal window to a graphical application. Thus, even if the command you are using in your build system 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 only be in effect for 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

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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_al
                { "key": "following_text", "operator": "regex_contains", "operand": "^\\}", "match_al
        ]
}
```

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:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

> **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 `Default/Preferences.sublime-settings`).

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 is hovered over the tab.

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

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 to be equal to.

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

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

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

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

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

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

Visual Settings

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

font_face Font face to be used for editable text.

font_size Size of the font for editable text.

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

gutter Toggles display of gutter.

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

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

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

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

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

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

line_numbers Toggles display of line numbers in the gutter.

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

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

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

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

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

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

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

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

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

Automatic Behavior

auto_match_enabled Toggles automatic pairing of quotes, brackets, etc.

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

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

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

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

System and Miscellaneous Settings

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

spell_check Toggles the spell checker.

dictionary Word list to be used by the spell checker. Accepts a path rooted at the data directory (e. g.: :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 automatically detected.

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

tab_completion Determines whether pressing `Tab` will insert completions.

Build and Error Navigation Settings

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

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

File and Directory Settings

default_dir Sets the default save directory for the view.

Input Settings

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

11.5 Command Palette

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the latest branch in the panel on the bottom left and consider updating Sublime Text.

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

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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 2 will look for plugins in these places:

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

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

All plugins should live inside a folder of their own and not directly under `Packages`.

11.6.2 Conventions for Command Names

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

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

For class definition names, use `CamelCasedPhrasesCommand`. To call a command from the API, use the normalized name (`snake_cased_phrases`).

11.6.3 Types of Commands

- `sublime_plugin.ApplicationCommand`
- `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, but they all must be valid JSON types.

11.6.4 How to Call Commands from the API

Use a reference to a `View` or a `Window`, or `sublime` depending on the type of command, and call `object.run_command('command_name')`. In addition, commands accept a dictionary whose keys are the names of valid parameters for them:

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

11.6.5 Command Arguments

All user-provided arguments to commands must be valid JSON types. Only Sublime Text itself can pass other types of arguments to commands (such as edit objects, view instances, etc.).

11.6.6 Text Commands and the `edit` Object

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

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

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

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

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

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

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 2 to become unresponsive, especially in events triggered frequently, like `on_modified` and `on_selection_modified`. Be careful of how much work is done in these and don't implement events you don't need, even if they just `pass`.

11.6.8 Python and the Standard Library

Sublime Text ships with a trimmed down standard library. The *Tkinter*, *multiprocessing* and *sqlite3* modules are among the missing ones.

11.6.9 Automatic Plugin Reload

Sublime Text will reload top-level Python modules from packages as they change (perhaps because you are editing a *.py* file). 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

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

See also:

Official Documentation for Sublime Text 2 API documentation.

Official Documentation for Sublime Text 3 API documentation.

11.7.1 Missing in the official docs

This section tries to address missing topics in the official documentation.

Index

module *sublime*

- class *Window*
    - *set_layout()*
- class *View*
    - *match_selector()*

module *sublime_plugin*

- class *EventListener*
    - *on_query_completions()*

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 group layout of the current window.

Expects a dictionary like this:

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

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

cols A list of the column seperators (*float*), should start with 0 (left) and end with 1 (right).

rows A list of the row seperators (*float*), 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* indices. 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. Thus, even though it makes zero sense to have a values lower than 0 or higher than 1 it is possible to specify them and Sublime Text will in fact treat them accordingly. Furthermore, it is possible to have the first value not be 0 and the last not be 1, the remaining space will simply be black in this case. Don't try this at home!

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 seperator 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.
> >
> > 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**

> A wrapper class for events. Subclass and define the methods you want to receive events on and you are done, no registering necessary.

> **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.

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

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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, even on Windows (e. g. `/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 still incomplete. While there are a few commands that are just not useful to a user (or even package developer) there are also a few undocumented commands or commands without a discription.

---

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]

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`.

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.

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 accessed directory (e.g. the directory of the currently opened file).

- **contents** [String]: This string will be written to the new buffer if the file does not exist.

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.

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_menu Shows or hides the menu 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 Unsurprisingly, it 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 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_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_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 prefer a block comment.

join_lines Joins the current line with the next one.

duplicate_line Duplicates the current line or selections if any.

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` prints less information about running the command.

> **See also:**

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

transpose Makes stuff dance (swap places).

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 Focusses 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. Only focusses on the first selection if multiple selections have been made

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.

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.

> • **level** [Int]: The level of indentation that should be folded.

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:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the latest branch in the panel on the bottom left and consider updating Sublime Text.

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

11.9.1 Editing

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

Windows

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

Linux

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

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
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:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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

11.10.1 Editing

Keypress	Command
+ X	Cut 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
+ 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

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

Keypress	Command
+ K, + U	Transform to Uppercase
+ K, + L	Transform to Lowercase

Glossary

> **Warning:** Development of Sublime Text has moved on to version 3.
> As a result, **this branch for Sublime Text 2 will not be updated any more**. Please select the `latest` branch in the panel on the bottom left and consider updating Sublime Text.

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

buffer, **91**

E

EventListener (class in sublime_plugin), 77

M

match_selector() (sublime.View method), 77

O

on_query_completions()   (sublime_plugin.EventListener
          method), 77
overlay, **91**

P

package, **91**
panel, **91**
plugin, **91**

S

set_layout() (sublime.Window method), 76
sublime (module), 75
sublime_plugin (module), 77

V

view, **91**
View (class in sublime), 76

W

Window (class in sublime), 76