# Sublime Text Unofficial Documentation

## *Release 3.0*

**guillermooo**

July 16, 2016

## Contents

# Backers 2014

Backers 2014

## 2.1 Installation

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

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

### 2.1.1 32 bits or 64 bits?

#### OS X

You can ignore this section: there is only one version of Sublime Text for OS X.

#### Windows

You should be able to run the 64-bit version if you are using a modern version Windows. If you are having trouble running the 64-bit version, try the 32-bit version.

#### Linux

Run this command in your terminal to check your operating system's type:

```
uname -m
```

### 2.1.2 Windows

#### Portable or Not Portable?

Sublime Text comes in two flavors for Windows: normal, and portable. Most users should be better served by a normal installation. Use the portable version only if you know you need it.

**Normal installations** separate data between two folders: the installation folder proper, and the *data directory* (user-specific directory for data; explained later in this guide). Normal installations also integrate Sublime Text with File Explorer.

**Portable installations** keep all files needed by Sublime Text in a single folder. This folder can be moved around and the editor will still work.

### How to Install the Normal Version of Sublime Text

1. Download the installer
2. Double click on the installer

### How to Install the Portable Version of Sublime Text

1. Download the compressed files
2. Unzip them to a folder of your choice

You will find the *sublime_text.exe* executable inside that folder.

## 2.1.3 OS X

1. Download *.dmg* file
2. Open *.dmg* file
3. Drag the Sublime Text 3 bundle into the *Applications* folder

To create a *symbolic link* to use at the command line issue the following command at the terminal:

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

## 2.1.4 Linux

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

### Ubuntu

**For i386**

```
cd ~
wget http://c758482.r82.cf2.rackcdn.com/sublime-text_build-3083_i386.deb
```

**For x64**

```
cd ~
wget http://c758482.r82.cf2.rackcdn.com/sublime-text_build-3083_amd64.deb
```

### Other Linux Distributions

**For i386**

```
cd ~
wget http://c758482.r82.cf2.rackcdn.com/sublime_text_3_build_3083_x32.tar.bz2
tar vxjf sublime_text_3_build_3083_x32.tar.bz2
```

**For x64**

```
cd ~
wget http://c758482.r82.cf2.rackcdn.com/sublime_text_3_build_3083_x64.tar.bz2
tar vxjf sublime_text_3_build_3083_x64.tar.bz2
```

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

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

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

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

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

First we need to create a new file.

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

Then copy the following into it.

```
[Desktop Entry]
Version=1.0
Name=Sublime Text 3
# Only KDE 4 seems to use GenericName, so we reuse the KDE strings.
# From Ubuntu's language-pack-kde-XX-base packages, version 9.04-20090413.
GenericName=Text Editor

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

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

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

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

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

## 2.1.5 Release Channels

At the time of this writing, two major versions of Sublime Text exist: Sublime Text 2 and Sublime Text 3. Generally speaking, Sublime Text 3 is the better choice. Even though it's technically in beta, it's as stable as Sublime Text 2 and has more features.

Use Sublime Text 2 only if you have found issues running Sublime Text 3 or you depend on any package not available for Sublime Text 3.

### Getting Sublime Text 3

Sublime Text 3 currently has two release *channels*:

- Beta (default, recommended)
- Dev

**Beta releases** are better tested and more reliable for everyday use than development builds. **The majority of users should only use beta releases.**

The *dev* channel is unstable: dev builds may contain bugs and not work reliably. Dev builds are updated more often than beta releases.

**Dev builds are only available to registered users.**

### Getting Sublime Text 2

We recommend Sublime Text 3, but if you have chosen to use Sublime Text 2 you can download it here.

## 2.2 Editing

### 2.2.1 Overview

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

### 2.2.2 Multiple Selections

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

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

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

### 2.2.3 Transforming Multiple Selections into Lines

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

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

### 2.2.4 Column Selection

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

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

#### Using the Mouse

Windows

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

Linux

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

OS X

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

**Using the Keyboard**

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

### 2.2.5 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`)

### 2.2.6 Transposing Things

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

### 2.2.7 And much, much more...

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

## 2.3 Search and Replace

Sublime Text features two main types of searches:

### 2.3.1 Search and Replace – Single File

**Searching**

Keyboard shortcuts related to the search panel:

| Open search panel | Ctrl + F |
|---|---|
| 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 |

### Incremental Search

Keyboard shortcuts related to the incremental search panel:

| Open incremental search panel | Ctrl + I |
|---|---|
| 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 |

The only difference between this panel and the regular search panel lies in the behavior of the `Enter` key. In incremental searches, it will select the next match in the file and dismiss the search panel for you. Choosing between this panel or the regular search panel is a matter of preference.

### Replacing Text

Keyboard shortcuts related to the replace panel:

| Open replace panel | Ctrl + H |
|---|---|
| Replace next | Ctrl + Shift + H |
| Replace all | Ctrl + Alt + Enter |

### Tips

#### Other Ways of Searching in Files

*Goto Anything* provides the operator # to search in the active file.

#### Other Search-Related Key Bindings

These key bindings work when the search panel is hidden:

| Search forward using most recent pattern | F3 |
|---|---|
| Search backwards using most recent pattern | Shift + F3 |
| Select all matches using most recent pattern | Alt + F3 |

You can also perform searches based on the current selection:

| Search using current selection | Ctrl + E |
|---|---|
| Replace using current selection | Ctrl + Shift + E |

**Multiline Search**

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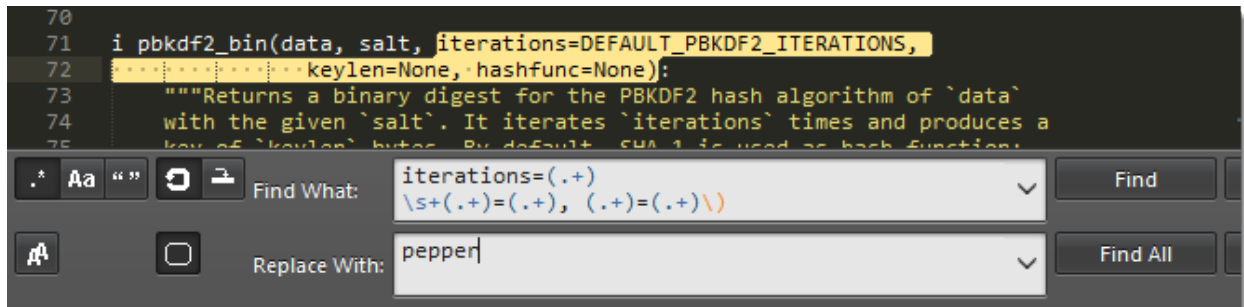


Fig. 2.1: A multiline pattern

Note that search panels are resizable too.

## 2.3.2 Search and Replace — Multiple Files

The search panel for searching multiple files is known as **Find in Files**.

**Searching**

Keyboard shortcuts related to Find in Files:

| Open Find in Files | Ctrl + Shift + F |
|---|---|
| Toggle regular expressions | Alt + R |
| Toggle case sensitivity | Alt + C |
| Toggle exact matches | Alt + W |
| Find next | Enter |

**Search Filters**

The **Where** field in Find in Files limits the search scope. You can define filters in several ways:

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

Relative paths in filters are interpreted to start at the root of the active project.

It is also possible to combine filters using commas. You can combine filters in any order.

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

**Results Format**

You can customize how results are displayed using buttons available in the Find in Files panel. These are the available options:
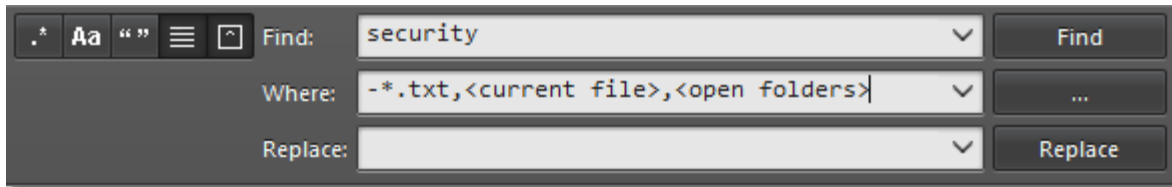
Fig. 2.2: Combining **Where** scopes in Find in Files
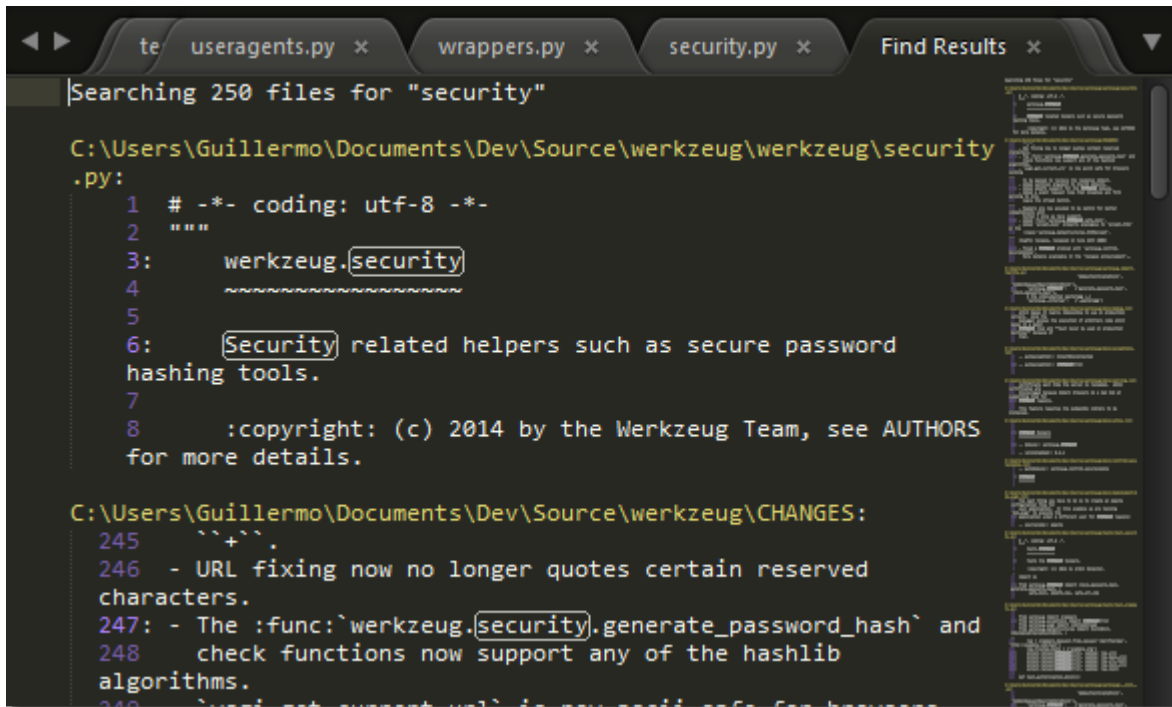
- Show in separate view (*Use Buffer*)

- Show context



Fig. 2.3: Find in Files results displayed in a view

### Navigating Results

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

| | |
|---|---|
| Next match | F4 |
| Previous match | Shift + F4 |

### 2.3.3 Regular Expressions

Search functions in Sublime Text support regular expressions, a powerful tool for searching and replacing text. Regular Expressions find complex *patterns* in text.

Regular expression patterns are composed of symbols and special constructs. To the non-initiated, regular expression patterns look undecipherable because common characters double as operators and cannot always be interpreted
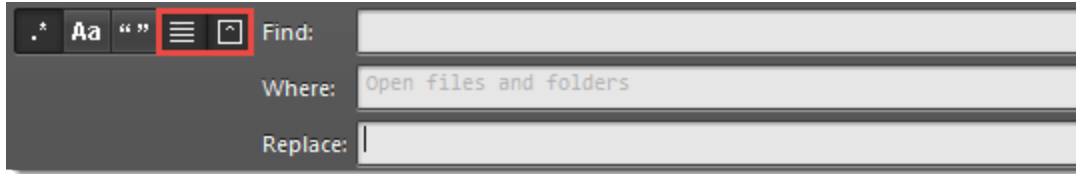
Fig. 2.4: Buttons for customizing the Find in Files results

literally.

This is how a regular expression might look:

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

The term *regular expression* is usually shortened to *regexp* or *regex*.

To take full advantage of the search and replace facilities in Sublime Text, you should at least learn the basics of regular expressions. This guide doesn't teach you how to use regular expressions.

The **Replace** box in a search and replace panel also supports special symbols called *format strings* that look similar to regular expressions. Format strings allow you to perform complex text transformations before inserting the result into the buffer.

Sublime Text uses the Perl Compatible Regular Expressions (PCRE) engine from the Boost library to power regular expressions in search panels.

### 2.3.4 Using Regular Expressions in Sublime Text

To use regular expressions in Sublime Text, first activate them in the corresponding search panel by clicking on the available buttons or using keyboard shortcuts.

If you don't activate regular expressions before performing a search, the search terms will be interpreted literally.

**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*.

## 2.4 Build Systems (Batch Processing)

**See also:**

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

> **Warning:** Build system selection is currently undergoing a rework in the dev channel. The following information may be outdated.
> See this forum thread for details.

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.
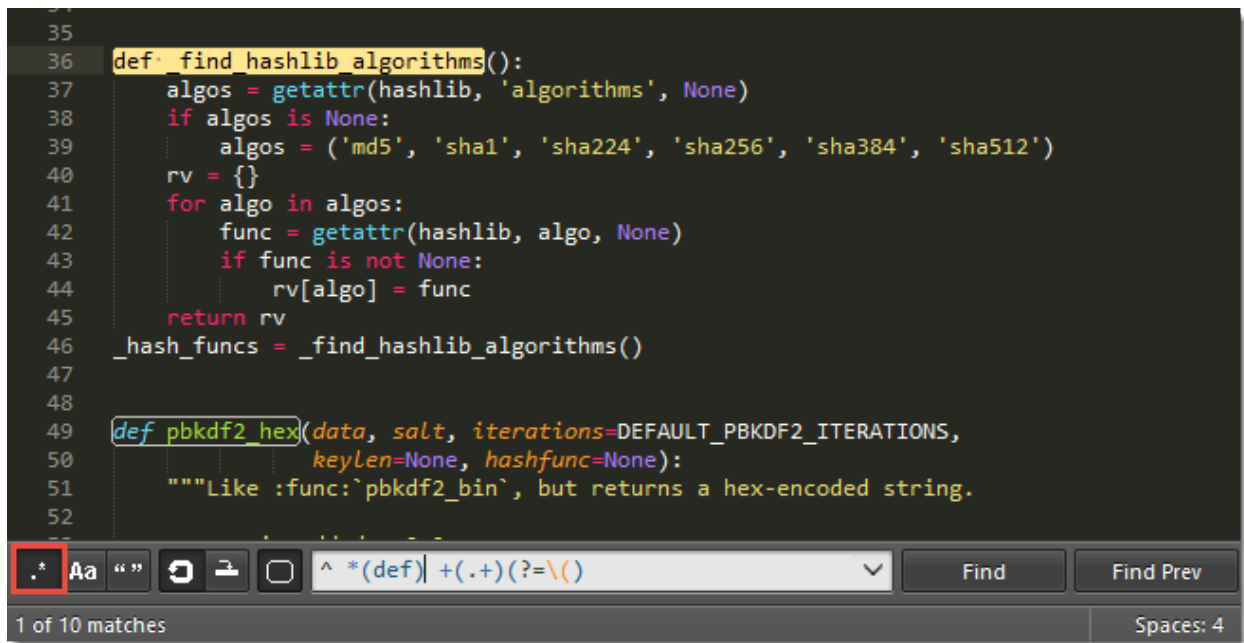
Fig. 2.5: A search panel with the regular expressions option enabled

### 2.4.1 File Format

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

#### Example

Here's an example of a build system:

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

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

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

**file_regex** A Perl-style regular expression to capture error information 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.

### 2.4.2 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.

---

### 2.4.3 Running Build Systems

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

## 2.5 File Navigation and File Management

Sublime Text includes a variety of features to help you keep your work organized and find your way around your projects.

### 2.5.1 Goto Anything

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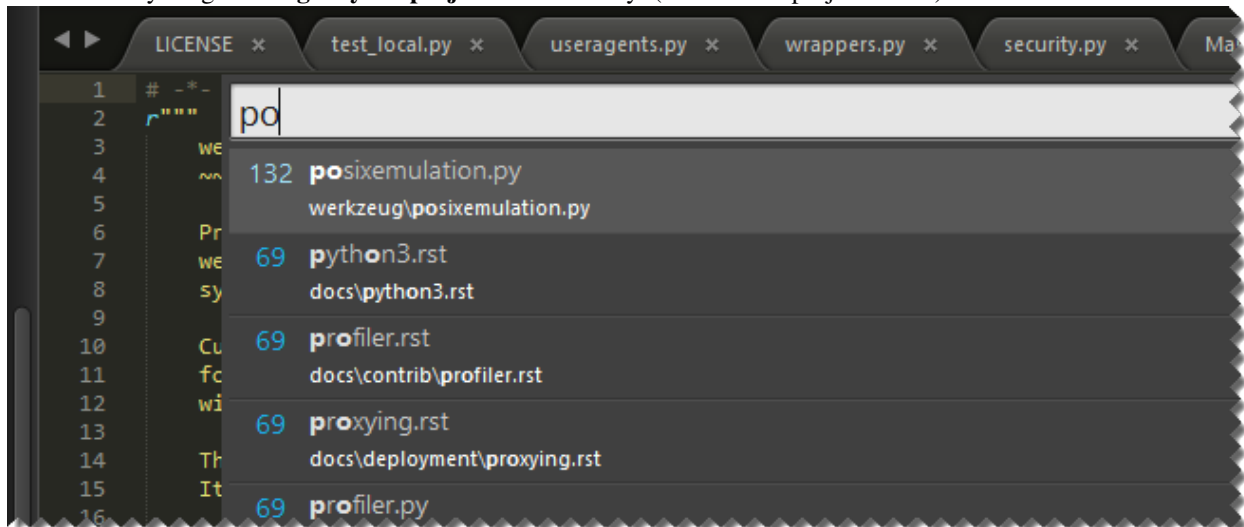


Keyboard shortcuts related to Goto Anything:

| Open Goto Anything | Ctrl + P |
|---|---|
| Pin current item and close Goto Anything | Enter |
| Pin current item | → |
| Close Goto Anything | Esc |

As you type into Goto Anything's input area, names of files in the current project will be searched, and a preview of the best match will be shown. This preview is *transient*; that is, it won't become the actual active view until you perform some operation on it. You will find transient views in other situations, for example, after clicking on a file in the sidebar.

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

### Goto Anything Operators

Goto Anything accepts several operators. All of them can be used on their own or after the search term.

Example:

```
models:100
```

This instructs Sublime Text to first search for a file whose path matches `models`, and then to go to line 100 in said file.

### Supported Operators

**@*symbol*** Searches the active file for the symbol named `symbol`.

---

**Note:** Symbols usually include class and function names.

Symbol searches will only yield results if the active file type has symbols defined for it. Symbols are defined in `.tmLanguage` files. For more information about symbols, see Symbols.

---

**#*term*** Performs a fuzzy search of the `term` search term and highlights all matches.

**:*line_number*** Goes to the specified `line_number`, or to the end of the file if `line_number` is larger that the file's line count.

The Goto Anything operators are bound to the following shortcuts:

| @ | Ctrl + R |
|---|----------|
| # | Ctrl + ; |
| : | Ctrl + G |

## 2.5.2 Sidebar

The sidebar provides an overview of the active project (more on projects later). Files and folders in the sidebar will be available in *Goto Anything* and project-wide actions like, for example, project-wide searches.

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

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

These are common keyboard shortcuts related to the side bar:

| **Toggle side bar** | Ctrl + K, Ctrl + B |
|---------------------|--------------------|
| Give the focus to the side bar | Ctrl + 0 |
| Return the focus to the view | Esc |
| Navigate side bar | Arrow keys |

Files opened from the sidebar create *semi-transient* views. Unlike transient views, semi-transient views show up as a new tab. The tab title of semi-transient views appears in italics. Before a new semi-transient view is opened, any other pre-existing semi-transient view in the same pane gets automatically closed.

Here's an example showing a normal view, a transient view, and a semi-transient view. Notice that the transient view has no tab:
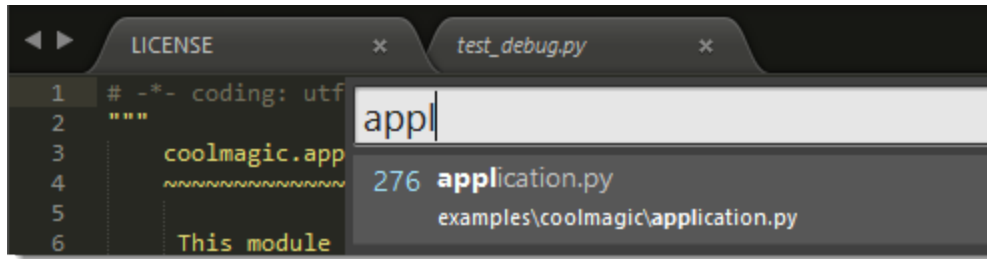
### 2.5.3 Projects

Projects group sets of files and folders to keep your work organized.

There is always an active project. If you haven't created one, an implicit one is created by Sublime Text.

Set up a project in a way that suits you by adding folders, and then save your new configuration. You can add and remove folders to/from a project using the **Project** menu or the side bar's context menu. If you drag a folder onto a Sublime Text window, it will be added to the project too.

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

Using the menu, you can switch projects by selecting **Projects** → **Recent Projects**.

Keyboard shortcuts related to projects:

| Switch project | Ctrl + Alt + P |
| --- | --- |

Project metadata is stored in JSON files with a `.sublime-project` extension. Wherever there's a `.sublime-project` file, you will find an ancillary `.sublime-workspace` file too. The second one is used by Sublime Text and you shouldn't edit it. (More on workspaces later.)

Projects can define settings applicable to that project only. See the official documentation for more information.

You can open a project from the **command line** by passing the `.sublime-project` file as an argument to the `subl` command line helper included with Sublime Text.

> **Warning:** Generally speaking, it's fine to commit `.sublime-project` files to a source code repository, but always be mindful of what you store in them.

#### The `.sublime-project` Format

Project metadata in `.sublime-project` files is split across three topmost sections: `folders`, for the included folders; `settings`, for project-specific settings; and `build_systems`, for project-specific build systems.

```
{
    "folders":
    [
        {
            "path": "src",
            "folder_exclude_patterns": ["backup"]
        },
        {
            "path": "docs",
            "name": "Documentation",
            "file_exclude_patterns": ["*.css"]
        }
    ],
```

```
    "settings":
    {
        "tab_size": 8
    },
    "build_systems":
    [
        {
            "name": "List",
            "cmd": ["ls"]
        }
    ]
}
```

**Folder Options**

**`path`** Required. The path may be relative to the project directory, or absolute.

**`name`** Optional. If present, it will appear in the side bar.

**`folder_exclude_patterns`** Optional. List of wildcards. Folders matching the wildcards will be excluded from the project.

**`folder_include_patterns`** Optional. List of wildcards. Folders matching the wildcards will be included in the project.

**`file_exclude_patterns`** Optional. List of wildcards. Files matching the wildcards will be excluded from the project.

**`file_include_patterns`** Optional. List of wildcards. Files matching the wildcards will be included in the project.

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

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

**See also:**

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

**Settings - Reference** Reference of available settings.

**Build Systems** You can define project-specific build systems in a `.sublime-project` file. A `name` must be specified for each one. Build systems included in a `.sublime-project` file will show up in the **Tools →  Build Systems** menu.

**See also:**

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

### 2.5.4 Other Settings Related to the Sidebar and Projects

**`binary_file_patterns`** A list of wildcards. Files matching these wildcards will show up in the side bar, but will be excluded from Goto Anything and Find in Files.

### 2.5.5 Workspaces

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

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

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

Workspaces metadata is stored in JSON files with the `.sublime-workspace` extension.

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

> **Warning:** Unlike `.sublime-project` files, `.sublime-workspace` files are not meant to be shared or edited manually. **You should never commit** `.sublime-workspace` **files into a source code repository.**

### 2.5.6 Panes

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

Main keyboard shortcuts related to panes:

| Create new pane | Ctrl+K, Ctrl+↑ |
|---|---|
| Close active pane | Ctrl+K, Ctrl+↓ |

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

## 2.6 Customizing Sublime Text

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

### 2.6.1 Settings

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

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

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

#### Format

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

#### Types of Settings

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

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

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

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

### How to Access and Edit Common Settings Files

Unless you need very fine-grained control over settings, you can access the main configuration files through the **Preferences | Settings - User** and **Preferences | Settings - More** menu items. Editing **Preferences | Settings - Default** is discouraged, because changes will be reverted with every update to the software. However, you can use that file for reference: it contains comments explaining the purpose of all available global and file type settings.

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

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

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

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

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

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

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

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

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

### Global Editor Settings and Global File Settings

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

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

**File Type Settings**

If you want to target a specific file type, name the *.sublime-settings* file after the file type's syntax definition. For example, if our syntax definition +was called `Python.tmLanguage`, we'd need to call our settings file *Python.sublime-settings*.

Settings files for specific file types usually live in packages, like +:file:*Packages/Python*, but there can be multiple settings files in separate locations for the same file type.

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

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

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

**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

**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`.

## 2.6.2 Indentation

**See also:**

**Indentation** Official Sublime Text Documentation.

### 2.6.3 Key Bindings

**See also:**

**Reference for key bindings** Complete documentation on key bindings.

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

#### File Format

Key bindings are defined in JSON and stored in `.sublime-keymap` files.

In the same package, separate keymap files for Linux, OSX and Windows may exist for better OS integration.

#### Example

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

#### Defining and Overriding Key Bindings

Sublime Text ships with default key bindings (for example, `Packages/Default/Default (Windows).sublime-keymap`). In order to override default key bindings or add new ones, use a separate keymap file with higher precedence: for example, `Packages/User/Default (Windows).sublime-keymap`.

See *Merging and Order of Precedence* for more information.

#### Advanced Key Bindings

Simple key bindings consist of a sequence of one or more keys mapped to a command. However, there are more complex syntaxes for passing arguments to commands and restricting key bindings to specific contexts.

#### Passing Arguments

Use the `args` key to specify arguments:

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

Here, `\n` is passed to the `insert` command whenever `Shift+Enter` is pressed.

#### Contexts

Contexts determine whether a given key binding is 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 snippet field available*. Thus, pressing ESC will only trigger this key binding if you are cycling through snippet fields.

The same key binding may be mapped to multiple contexts, so a single sequence of key presses may produce different results at different times.

### Key Chords

You can create key bindings composed of multiple keys.

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

Here, to trigger the command paste_from_history, you have to press Ctrl + K first, then release K, and finally press V.

Note: This example is a default key binding, so you can try it at any time.

## 2.6.4 Menus

**See also:**

**Reference for menus** Reference for menu creation.

Sublime Text provides several menus that can be modified, for example, by adding menu items.

### File Format

| Format | JSON (with comments) |
|---|---|
| Exten-sion | .sublime-menu |
| Name | One out of the list of available menus. See *Available Menus* for the complete name list and what menu they represent. |
| Loca-tion | Any under Packages |
| Content | A list of menu-item-objects |

### Example

The following is an excerpt from the default Main.sublime-menu file.

```
[
    {
        "caption": "Edit",
        "mnemonic": "E",
        "id": "edit",
        "children":
        [
            { "command": "undo", "mnemonic": "U" },
            { "command": "redo_or_repeat", "mnemonic": "R" },
            {
                "caption": "Undo Selection",
                "children":
                [
                    { "command": "soft_undo" },
```

---

```
                    { "command": "soft_redo" }
                ]
        },
        { "caption": "-", "id": "clipboard" },
        { "command": "copy", "mnemonic": "C" },
        { "command": "cut", "mnemonic": "t" },
        { "command": "paste", "mnemonic": "P" },
        { "command": "paste_and_indent", "mnemonic": "I" },
        { "command": "paste_from_history", "caption": "Paste from History" }
    ]
    }
]
```

**Images**
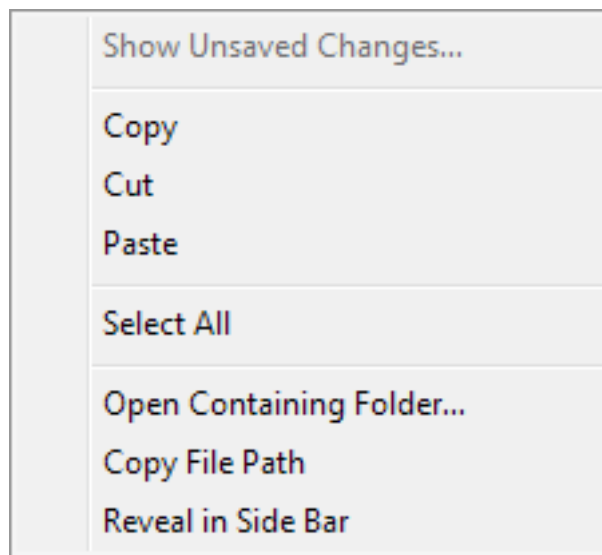


Fig. 2.6: The default context menu in the editing area.

**Available Menus**

The file name of a `.sublime-menu` file specifies the menu to be modified in the application.

The following menus are available:

| File/Menu Name | Description |
|---|---|
| **Main** | Main menu |
| **Context** | Context menu in the editing area |
| **Find in Files** | Appears when clicking the "…" button in the Find in Files panel. |
| **Side Bar** | Context menu for each node in the sidebar |
| **Side Bar Mount Point** | Additional context menu items for the top-level nodes in the sidebar |
| **Tab Context** | Context menu of the tab bar |
| **Widget Context** | Context menu of input fields in all kinds of widgets, including Command Palette, Goto Anything, the Find panels and panels opened by plugins |
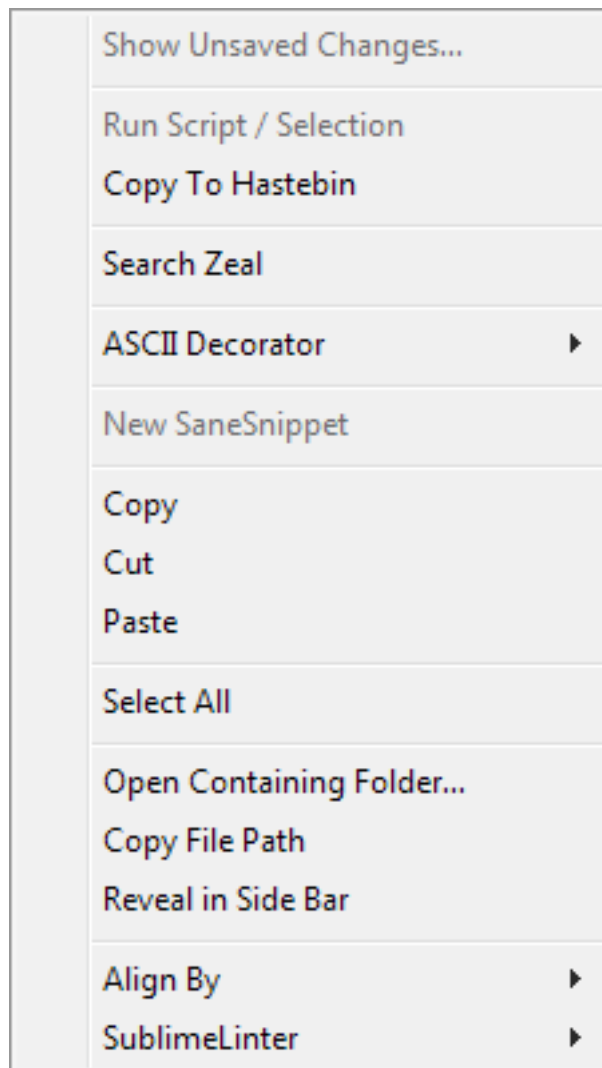
Fig. 2.7: A modified context menu in the editing area.

Additionally, the following four menus open when you click their respective section in the status bar:

- **Encoding**
- **Line Endings**
- **Indentation**
- **Syntax**

Fig. 2.8: Demonstration of a status bar menu.

### Menu Items

When selected, a menu item can either invoke a command (with arguments), or open a submenu.

The available properties are:

- a command name,
- arguments for the command,
- an ID,
- a caption,
- a mnemonic and
- a submenu.

In order to function properly, a menu item must provide at least:

- a command name,
- a caption and a submenu,
- just a caption, or
- an ID (see *below*).

When parsing a menu item, the following rules apply:

1. A menu item with a submenu cannot invoke a command. If the *separator caption* is used, it will be rendered as a literal hyphen.

2. If no caption is provided, a caption is inferred from the command's `description` method. If neither caption nor command are provided, the caption will be an empty string.

3. The character used for the mnemonic must be contained in the item's caption. Mnemonics are case-sensitive.

4. Menu that reference missing commands are disabled.

5. Menu items can be hidden or disabled by their referenced command.

### Separators

Separators are menu items with the caption – and no submenu. They are commonly used to group menu items with a similar purpose or that are otherwise related. Separators cannot invoke commands. The presence of a submenu causes the menu separator to be rendered as a regular item with a single hyphen as its caption.

Multiple consecutive separators are reduced to one, and separators at the beginning or the end of a menu are not displayed.

**Menu Merging**

`.sublime-menu` files are loaded in the same order as packages. Menu files with the same name are concatenated, unless IDs are specified (see *below*).

Menu files in the same package are loaded in lexicographical order starting at the root folder, and then traversing sub-folders in the same manner.

As a special case, menu items from the *User* package declared in the standard non-ID section of a menu are always inserted after any standard items from other packages.

**Item IDs**

When a menu item specifies an ID, a separate section within the menu is searched for and, if it does not exist, created at the end of the menu. This ID lookup is forward-going only.

The ID determines the section's name, and the menu item with the matching ID will be the first item in this section. All following items in the file will then be appended to the ID's section, until another item with an ID is found.

If two menu items from different `.sublime-menu` files reference the same item via ID, Sublime Text will override the item's previous parameters with the new parameters, if there are any. Child elements extend the submenu. All following items are then appended to the ID's section, until another item with an ID is found.

It is common practice to assign IDs to separators and items having a submenu, so that other packages or the user themselves can easily customize the menu. This allows appending items to sections introduced by separators and appending items to submenus.

---

**Note:** Due to the strict forward lookup, it is possible to have *multiple different items with the same ID* in one menu. Because of the potential confusion this may cause, it is discouraged.

Example: The following three IDs are defined in a menu, in this order: `test, test2, test`.

The item with the second "test" ID can then be targeted using the following ID combinations: `test, test2, test`; `test, test` or `test2, test`.

---

**Submenus**

Every menu item can have a submenu. Hovering the mouse pointer over a menu item with a submenu will reveal the items grouped under it. Submenus are independent menus with their own ID hierarchy.

In order to extend a submenu from a different menu file, an ID must be specified in both places to target the correct item.

**The Main Menu**

Unlike all other menus, the Main Menu's root menu represents the menu items in the menu bar (File, Help, etc.).

**Interface for Commands**

A menu item can be dynamically

- hidden,
- disabled,

- checked, or

- assigned a different caption.

For this, commands must implement the required methods in their class. Each implemented method will be called with the arguments specified in the corresponding menu item. If that call fails, the method will be immediately called again without arguments.

- `is_visible`

- `is_enabled`

- `is_checked`

- `description`

Some of these methods also have an influence on the Command Palette.

**See also:**

Official API Documentation on the Command interface

### Context Menus in the Side Bar

The **Side Bar** and **Side Bar Mount Point** menus are different to the other menus in that they provide **contextual information** regarding the selected item(s). The selected directories and files are passed as a list to the specified command in a `files` argument.

## 2.6.5 Color Schemes

Sublime Text uses color schemes to highlight source code and to define the colors of some items in the editing area: background, foreground, gutter, caret, selection...

Color schemes are fully customizable.

Let's look at a Python file as Sublime Text highlights it using the default color scheme:

```
54
55    class Revision(object):
56        """
57        Represents one revision of a page.
58        This is useful for editing particular revision of pages or creating
59        new revisions.  It's also used for the diff system and the revision
60        log.
61        """
62        query = session.query_property()
63
64        def __init__(self, page, text, change_note='', timestamp=None):
65            if isinstance(page, (int, Long)):
66                self.page_id = page
67            else:
68                self.page = page
69            self.text = text
70            self.change_note = change_note
71            self.timestamp = timestamp or datetime.utcnow()
72
73        def render(self):
74            """Render the page text into a genshi stream."""
75            return parse_creole(self.text)
76
77        def __repr__(self):
78            return '<%s %r:%r>' % (
79                self.__class__.__name__,
80                self.page_id,
81                self.revision_id
82            )
83
-- INSERT --, Line 73, Column 22                          Spaces: 4          Python
```

**See also:**

**Reference for color schemes** Complete reference on color scheme settings.

## 2.7 Extending Sublime Text

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

### 2.7.1 Commands

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

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

#### 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.

### Anatomy of a Command

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

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

**See also:**

**Reference for commands** Command reference.

## 2.7.2 Macros

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

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

### How to Record Macros

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

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

### How to Edit Macros

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

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

See the `/reference/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 `\`.

### Where to Store Macros

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

### Key Binding for Macros

Macro files can be bound to key combinations by passing the macro file path to the *run_macro_file* command like so:

```
{"keys": ["super+alt+l"], "command": "run_macro_file", "args": {"file": "res://Packages/User/Example
```

## 2.7.3 Snippets

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

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

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

### 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 look at each 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 `translate_tabs_to_spaces` is `true`.

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

- The `content` of your snippet must not contain `]]>` because this string of characters will prematurely close the `<![CDATA[...]]>` section, resulting in an XML error. To work around this pitfall, you can insert an undefined variable into the string like this: `]]$NOT_DEFINED>`. This modified string passes through the XML parser without closing the content element's `<![CDATA[...]]>` section, but Sublime Text will replace `$NOT_DEFINED` with an empty string before inserting the snippet into your file. In other words, `]]$NOT_DEFINED>` in your snippet file `content` will be written as `]]>` when you trigger the snippet.

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

A tab trigger is an implicit key binding.

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

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

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

---

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

---

### Snippet Features

### Environment Variables

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

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

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

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

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


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

### Fields

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

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

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

If you want to control where the exit point should be, use the `$0` mark. By default, this is the end of the snippet.

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

### Mirrored Fields

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

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

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

### Placeholders

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

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

Variables can be used as placeholders:

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

And you can nest placeholders within other placeholders too:

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

### Substitutions

In addition to the place holder syntax, tab stops can specify more complex operations with substitutions. Use substitutions to dynamically generate text based on a mirrored tab stop. Of course, the tab stop you want to use as variable has to be mirrored somewhere else in the 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: =========
```

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

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

# Output:

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

### 2.7.4 Completions

In the spirit of IDEs, Sublime Text suggests completions that aggregate code or content while writing by catching everything that you have written, like variable names.

There are however several ways to extend the list of completions (for example, depending on the current syntax).

This topic deals with how completions are used and where they come from.

#### 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.

Completions can be inserted in two ways:

- through the completions list :
    - `Ctrl + Spacebar` on Windows / OS X,

– `Alt + /` on Linux

- or by pressing `Tab`.

### The Completions List

To use the completions list:

1. Press `Ctrl + Spacebar` (`Alt + /` on Linux) or just type something.

2. Optionally, press the shortcut again to select the next entry or use *up* and *down* arrow keys.

3. Press `Enter` or `Tab` to validate selection (depending on the `auto_complete_commit_on_tab` setting)

4. Optionally, press `Tab` repeatedly to insert the next available completion.

---

**Note:** If the completions list was opened explicitly, the current selection in the completions list can also be validated with any punctuation sign that isn't itself bound to a snippet (e.g. `.`).

---

When the list of completion candidates can be narrowed down to one unambiguous choice given the current prefix, this one completion will be validated automatically the moment you trigger the completion list.

**Hints**   Additionally, you may see a trigger hint on the right side of a completion's trigger in the completions list. This can be used as a preview of the completion's content.
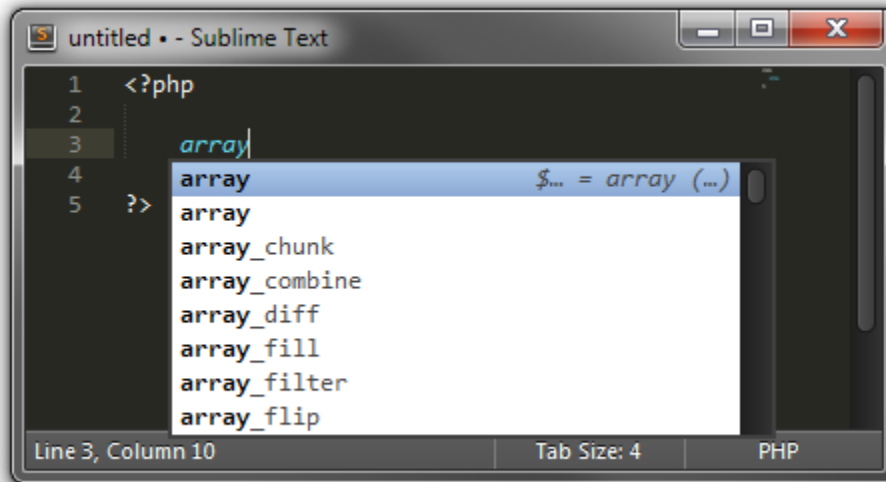


The above is in fact a snippet and expands to `$arrayName = array('' => , );`.

### Triggers and Contents

Completions not sourced from the text in the current file may provide a trigger that is different to the content they will insert if selected. This is commonly used for function completions where the content also includes the function's signature.

For example, completing `array_map` from the PHP completions will result in `array_map(callback, arr1)`:

You may notice in the animation that the cursor automatically selected `callback`. This is because completions support the same features as snippets with fields and placeholders. For more details, refer to *Snippet Features*.

---

### Completions with multiple cursors

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

Working example (`|` represents one cursor):

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

Not working example:

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

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

### `Tab`-Completed Completions

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

With `tab_completion` enabled, completion of items is always automatic. This means, unlike the case of the completions list, that Sublime Text will always make the decision for you. The rules for selecting the best completion are the same as described above, but in case of ambiguity, Sublime Text will insert the item it deems most suitable. You can press the `Tab` key multiple times to walk through other available options.

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

### Sources for Completions and their Priorities

These are the sources for completions the user can control, in the order they are prioritized:

1. Snippets

2. API-injected completions via *on_query_completions()*

3. `Completions files`

Additionally, the following completions are folded into the final list:

4. Words in the buffer

Snippets will always win when the current prefix matches their tab trigger *exactly*. For the rest of the completion sources, a fuzzy match is performed. Furthermore, snippets always lose in a fuzzy match.

When a list of completions is shown, snippets will still be listed alongside the other items, even if the prefix only partially matches the snippets' tab triggers.

---

**Note:** Completions sourced from words in the buffer can be suppressed explicitly from an `on_query_completions` event hook.

---

### 2.7.5 Command Palette

**See also:**

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

#### Overview

The *command palette* is an interactive list whose purpose is to execute commands. The command palette is fed entries by means of `.sublime-commands` files. Commands that don't warrant creating a key binding of their own are usually good candidates for inclusion in a `.sublime-commands` file.
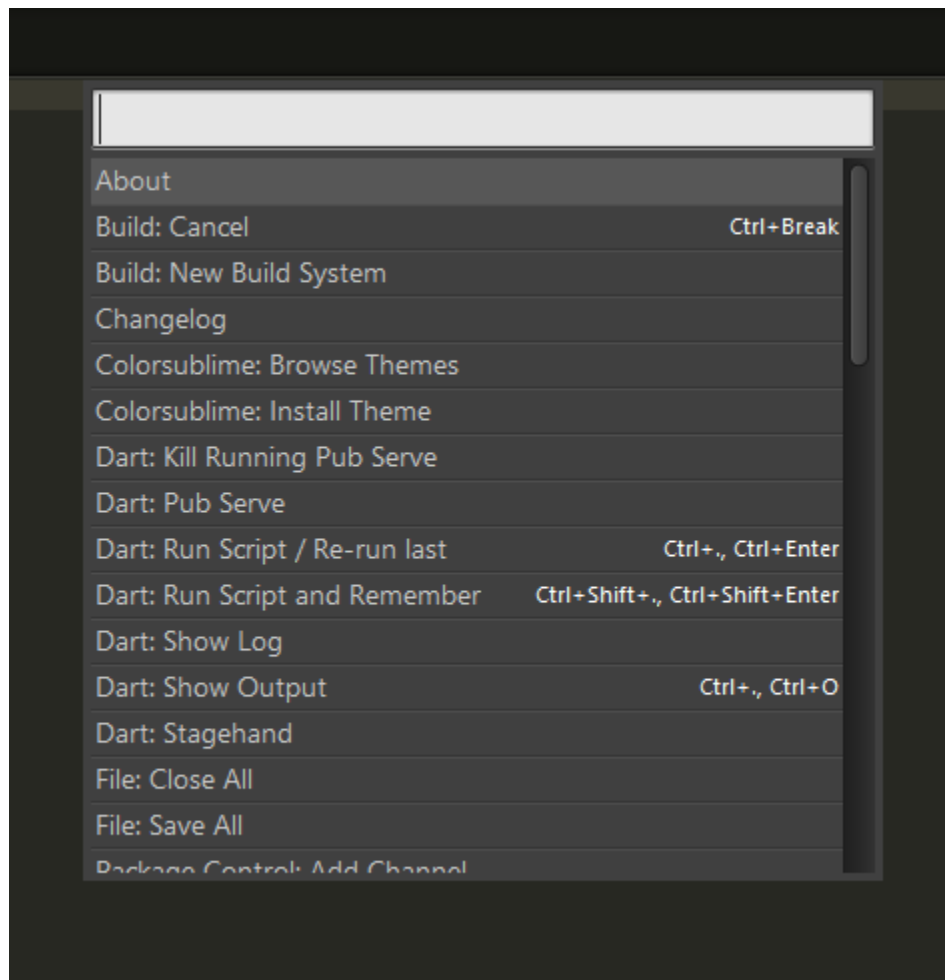


Fig. 2.9: The Command Palette

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

To use the command palette:

1. Press `Ctrl+Shift+P`

2. Select a command

---

3. Press Enter

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

### Example of a `.sublime-commands` File

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

## 2.7.6 Syntax Definitions

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

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

---

**Note:** As of Sublime Text Build 3084, a new syntax definition format has been added, with the .sublime-syntax extension.

Documentation is available here: http://www.sublimetext.com/docs/3/syntax.html

---

### Prerequisites

In order to follow this tutorial, you will need to install PackageDev, 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.

### 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 PackageDev package (mentioned above) comes in.

---

**Note:** If you experience unexpected errors during this tutorial, chances are PackageDev 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.

### 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. In particular, Color Schemes make extensive use of scopes to style every aspect of a language in the desired color.

### 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.

### 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 PackageDev 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 topmost 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.

### Analyzing Patterns

The `patterns` array can contain several types of element. 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. PackageDev 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 PackageDev knows what you want to convert your file to: It's specified in the first comment 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 PackageDev.

---

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 tell 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 PackageDev is backwards compatible.

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

---

**See also:**

**/reference/syntaxdefs** Reference for syntax definitions

### 2.7.7 Plugins

**See also:**

**API Reference** More information on the Python API.

**Plugins Reference** More information about plugins.

This section is intended for users with programming skills.

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

#### Prerequisites

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

#### Where to Store Plugins

Sublime Text will look for plugins only in these places:

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

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

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

### Your First Plugin

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

1. Select **Tools | New Plugin...** in the menu.

2. Save to `Packages/User/hello_world.py`.

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

1. Create a new buffer (`Ctrl+n`).

2. Open the Python console (`Ctrl+'`).

3. Type: `view.run_command("example")` and press enter.

You should see the text "Hello, World!" in the newly created buffer.

### 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; they are not part of the Python standard library.

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

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

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

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

### Conventions for Command Names

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

New commands should follow the same naming pattern.

## Types of Commands

You can create the following types of commands:

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

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

### Shared Traits of Commands

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

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

### Window Commands

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

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

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

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

### Text Commands

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

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

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

### Text Commands and the `edit` Object

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

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

### Responding to Events

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

**Another Plugin Example: Feeding the Completions List**

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

```python
import sublime, sublime_plugin

from xml.etree import ElementTree as ET
import urllib

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(
            urllib.request.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.

**Learning the API**

In order to create plugins, you need to get acquainted with the Sublime Text API and the available commands. Documentation on both is scarce at the time of this writing, but you can read existing code and learn from it.

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

## 2.7.8 Packages

### Overview

A package is a container for resources.

### Package Locations (and Abbreviations)

There are three locations for storing packages for different purposes.

- Packages can be **folders** under `Data`/`Packages` (short: `Packages`)

- or **zip archives** with the `.sublime-package` extension located under `Data`/`Installed Packages` (short: `Installed Packages`) or any of its subdirectories.

- Additionally, Sublime Text provides a set of default packages as **zip archives** in `Application`/`Packages` (short: `Shipped Packages`), where `Application` refers to the folder where the Sublime Text executable resides.

  This folder is not intended to be modified by the user.

---

**Note:** For simplicity, we will occasionally refer to all these directories simply as `Packages`, and to a package in any folder (`.sublime-package` or not) as `Packages`/PackageName. Consequently, a file inside a package may also be referred to as PackageName/a_file.extension.

---

#### `.sublime-package` Packages

Packages distributed as `.sublime-package` zip archives should be considered read-only containers of resources and never be modified manually. Since they are usually updated as a whole, any manual changes made to them will be lost in the process.

If you do want to modify files in these archives, see *Customizing or Overriding Packages*.

#### Interactions Between Packages with The Same Name

If two packages with the same name exist in both `Installed Packages` and `Shipped Packages`, the one in `Installed Packages` will be used and the one in `Shipped Packages` will be ignored.

Any file in `Packages/PackageName` takes precedence over an identically named file in `Installed Packages/PackageName`.sublime-package or *Shipped Packages/PackageName*.sublime-package.

See also *Customizing or Overriding Packages*.

### Package Contents

Typical resources found in packages include:

- build systems (`.sublime-build`)
- color schemes (`.tmTheme`)
- key maps (`.sublime-keymap`)
- macros (`.sublime-macro`)
- menus (`.sublime-menu`)
- metadata (`.tmPreferences`)
- mouse maps (`.sublime-mousemap`)
- plugins (`.py`)
- settings (`.sublime-settings`)
- snippets (`.sublime-snippet`)
- syntax definitions (`.tmLanguage`)
- themes (`.sublime-theme`)

Some packages may hold support files for other packages or for core features. For example, the spell checker uses `Installed Packages/`Language - English.sublime-package as a data store for English dictionaries.

### Types of Packages

In this guide, we categorize packages for clarity when discussing this topic, but Sublime Text doesn't use this terminology and you don't need to learn it.

**shipped packages, default packages**  A set of packages that Sublime Text ships with. Some of these packages are *core packages*, while others enhance Sublime Text to support common programming languages out of the box.

Examples: Default, Python, Java, C++, Markdown.

Located in *Shipped Packages*.

**core packages**  Sublime Text requires these packages in order to function properly.

Complete list: Default, Theme - Default, Color Scheme - Default, Text, Language - English.

They are part of the shipped packages and located in *Shipped Packages*.

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

Example: User.

Located in *Packages* and *Installed Packages*.

**installed packages**  A subtype of *user packages*.

Installed packages are `.sublime-package` archives and usually maintained by a package manager.

Located in *Installed Packages*.

---

**Note:** Due to the unfortunate name of this folder, talking about *installing* packages in Sublime Text is confusing.

---

Sometimes, in this guide, by *installing* we mean 'adding a user/third party package to Sublime Text' (in any form), and sometimes we use the term in its stricter sense of 'copying a `.sublime-package` archive to `Installed Packages`'.

**override packages**  A special type of *user packages*.

Override packages serve the purpose of customizing packages that are distributed as `.sublime-package` files. They are effectively injected into the original package and do not stand-alone.

See *Customizing or Overriding Packages* for details.

Located in `Packages`.

Note that by *third party* we also refer to users of other editors, notably Textmate, as Sublime Text and Textmate share some types of resource files that can be reused without modification.

## Managing Packages

### Installing Packages

**Note:**  Regular users rarely need to know how to install packages by hand, as automatic package managers are available.

The de facto package manager for Sublime Text is Package Control.

Packages can be installed in two main ways:

- by copying Sublime Text resources to a folder under `Packages`, or
- by copying a `.sublime-package` file to `Installed Packages`.

### Disabling Packages

To temporarily disable packages, you can add them to the `ignored_packages` list in your `Packages`/User/Preferences.sublime-settings file.  Packages will be loaded or unloaded as needed when the settings file is saved.

### Enabling Packages

To re-enable a package, remove the package's name from the `ignored_packages` list in your `Packages`/User/Preferences.sublime-settings file.

### Removing Packages

If you installed a package with a package manager, remove it using the method provided by the package manager.

If you installed a package manually, follow this procedure to safely remove a package:

1. *Disable* the package while Sublime Text is running.
2. Close Sublime Text.
3. Remove the package's resources from the disk.

4. Remove the package's name from the `ignored_packages` list.

In addition to the resources you have placed initially in a *Packages* folder or in *Installed Packages*, plugins may create configuration files (such as `.sublime-settings` files) or other files to store package-related data. Frequently, you will find them in the *User* package. Therefore, if you want to remove all traces of a package, you will need to find and remove all the additional files that it may have installed.

> **Warning:** Shipped packages are reinstated during every Sublime Text update, so you can't delete them forever. If you want to stop using a shipped package, *disable* it.

### Customizing or Overriding Packages

Since packages in `.sublime-package` zip archives *are read-only*, you cannot modify them directly. However, Sublime Text allows you to create an *override package* that will effectively inject files into the original archive without modifying the archive itself.

To create an override package, create a new folder under *Packages* and name it after the `.sublime-package` file you want to override, excluding the extension. Any file you create in this package will take precedence over any identically named file in the original package.

Python plugins in override packages are able to use relative imports for accessing other modules in the corresponding `.sublime-package` file as if they were part of it.

> **Warning:** Files in override packages override entire files. If the overriden file in the corresponding `.sublime-package` is updated, you will not be notified.

### Merging and Order of Precedence

Package precedence is important for merging certain resources, for example, `.sublime-keymap` and `.sublime-settings` files, and for loading plugins (`.py` files).

If an *override package* exists for a `.sublime-package` package, it will be loaded at the same time as the `.sublime-package` archive.

Sublime Text loads packages in this order:

1. *Packages*/Default;
2. *shipped packages* in lexicographical order;
3. *installed packages* in lexicographical order;
4. all remaining *user packages*, except for *Packages*/User, that did not override anything, in lexicographical order;
5. *Packages*/User

### Reverting Sublime Text to Its Default Configuration

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

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

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

## 2.8 Command Line Usage

Sublime Text includes a command-line helper called `subl`. Using the command-line helper, you can open files and folders and perform other actions from the command line.

Before using `subl`, make sure it is on your `PATH`. To put `subl` on your `PATH`, you may need to add directories to `PATH` or use symbolic links.

### 2.8.1 Invocations

| | |
|---|---|
| `subl [options] [files]` | Open given files. |
| `subl [options] [directories]` | Open given directories. |
| `subl [options] -` | Edit standard input. |

Filenames may be given a `:line` or `:line:column` suffix to open at a specific location. The `line` and `colunmn` specifiers are 1-based offsets.

Reading from standard input only works in OS X.

### 2.8.2 Options

The `subl` command-line helper accepts a few options. For details, see below the next summary.

| | |
|---|---|
| `--project <project>` | Load the given project. |
| `--command <command>` | Execute the given command. |
| `--new-window, -n` | Open a new window. |
| `--add, -a` | Add folders to the current window. |
| `--wait, -w` | Wait for the files to be closed before returning. |
| `--background, -b` | Don't activate the application. |
| `--stay, -s` | Keep the application activated after closing the file. |
| `--help, -h` | Display help. |
| `--version, -v` | Show version information. |

**`--project <project>`** Load the given project.

> The **project** argument designates the `.sublime-project` or `.sublime-workspace` file to be loaded.

**`--command <command>`** Execute the given command.

> The **command** argument designates the command to be run.
>
> If Sublime Text isn't running already, only `ApplicationCommands` will work when invoked from the command line. If Sublime Text is already running, `WindowCommands` will work as well when invoked from the command line.
>
> You can also pass arguments to the command. The arguments must be separated from the command name by a space and represented as a JSON object. As usual, you have to escape quotation marks and other characters as required by your shell. For example, this syntax may work in bash as well as PowerShell: `subl --command 'echo {\"foo\": 100 }'`.

**`--new-window (-n)`** Open a new window.

> This option should be used when an instance of Sublime Text is already running.

---

**--add (-a)** Add folders to the current window.

> Add folders to the current window instead of opening a new window.

**--wait (-w)** Wait for the files to be closed before returning.

> This is useful, for example, to use Sublime Text as an editor with version control systems like git. Implied if reading from standard input.

**--background (-b)** Don't activate the application.

**--stay (-s)** Keep the application activated after closing the file.

> Only works in combination with --wait.

**--help (-h)** Display help.

**--version (-v)** Show version information.

## 2.9

Sublime Text   Sublime Text

### 2.9.1 Settings – Reference

**See also:**

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

#### Global Settings

Global settings can only be modified in `Preferences.sublime-settings` and `Preferences (<platform>).sublime-settings` (where `<platform>` can be any of `Linux`, `OSX` or `Windows`) and, where indicated, also in `.sublime-project` files.

`theme`

| Type | str |
|---|---|
| Default | `"Default.sublime-theme"` |

> Theme to be used. Accepts a base name for a `.sublime-theme` file.

`scroll_speed`

| Type | float |
|---|---|
| Default | `1.0` |

> Controls the smooth scrolling feature.
>
> - A value of `0` disables smooth scrolling.
> - A value between `0` and `1` makes scrolling slower.
> - A value of `1` is the default scrolling speed.
> - A value larger than `1` makes scrolling faster.

`hot_exit`

| Type | bool |
|---|---|
| **Default** | `true` |

If `true` and you exit the application or active window, Sublime Text will close the application or window without prompting, even if there are unsaved files. Unsaved files and the state of the active project will be restored the next time Sublime Text starts.

`remember_open_files`

| Type | bool |
|---|---|
| **Default** | `false` |

If `true`, every time you start Sublime Text it will reopen the files that were open when the application was closed the last time.

`open_files_in_new_window`

| Type | bool |
|---|---|
| **Default** | `true` |

**OS X only.** If `true`, a new window is created when files are opened from Finder or by dragging them onto the dock icon.

`close_windows_when_empty`

| Type | bool |
|---|---|
| **Default (All)** | `false` |
| **Default (OS X)** | `true` |

If `true` and no folder is open in the active window, the window will be closed when the last file is closed.

`show_full_path`

| Type | bool |
|---|---|
| **Default (All)** | `true` |
| **Default (OS X)** | `false` |

If `true`, show the active file's full path in the title bar.

`preview_on_click`

| Type | bool |
|---|---|
| **Default** | `true` |

If `true`, preview file contents when clicking on a file in the side bar. Files in preview will be closed automatically as soon as they lose focus (for example, if you press `Esc`). Double-clicking or editing a file in preview will open the file in a tab.

`folder_exclude_patterns`

| Type | [str,] |
|---|---|
| **Default** | `[".svn", ".git", ".hg", "CVS"]` |

Accepts wildcards. Excludes the matching folders in the project from the side bar, Goto Anything and any project-wide action.

This setting must be used in a global `Preferences.sublime-settings` file, or in folder items in a `.sublime-project` file.

`file_exclude_patterns`

| Type | [str,] |
|------|--------|
| **De-fault** | ["*.pyc", "*.pyo", "*.exe", "*.dll", "*.obj", "*.o", "*.a", "*.lib", "*.so", "*.dylib", "*.ncb", "*.sdf", "*.suo", "*.pdb", "*.idb", ".DS_Store", "*.class", "*.psd", "*.db", "*.sublime-workspace"] |

Accepts wildcards. Excludes the matching files in the project from the side bar, Goto Anything and any project-wide action.

This setting must be used in a global `Preferences.sublime-settings` file, or in folder items in a `.sublime-project` file.

`binary_file_patterns`

| Type | [str,] |
|------|--------|
| **De-fault** | ["*.jpg", "*.jpeg", "*.png", "*.gif", "*.ttf", "*.tga", "*.dds", "*.ico", "*.eot", "*.pdf", "*.swf", "*.jar", "*.zip"] |

Accepts wildcards. Excludes the matching files in the project from Goto Anything and any project-wide action, but not the side bar.

This setting must be used in a global `Preferences.sublime-settings` file, or in folder items in a `.sublime-project` file.

`show_tab_close_buttons`

| Type | bool |
|------|------|
| **Default** | `true` |

If `false`, hides close buttons in the tab bar until you hover the mouse over the tab bar.

`mouse_wheel_switches_tabs`

| Type | bool |
|------|------|
| **Default** | `false` |

If `true` and the cursor is in the tab bar, scrolling the mouse wheel will switch tabs.

`ignored_packages`

| Type | [str,] |
|------|--------|
| **Default** | `["Vintage"]` |

A list of packages that Sublime Text will ignore. Packages in this list will be disabled until you remove them from the list.

Some packages may not respond gracefully to being disabled/enabled via this setting. Therefore, after editing this setting, you should restart Sublime Text.

### File Settings

#### Whitespace and Indentation

`auto_indent`

| Type | bool |
|------|------|
| **Default** | `true` |

Toggles automatic indentation. Automatic indentation will try to calculate the correct indentation when you press :kbd:´Enter´.

`smart_indent`

| Type | bool |
|------|------|
| **Default** | `true` |
| **Depends on** | `auto_indent` |

Toggles smart indentation. Smart indentation tries additional heuristics to calculate the correct indentation. when you press :kbd:´Enter´.

`indent_to_bracket`

| Type | bool |
|------|------|
| **Default** | `false` |
| **Depends on** | `auto_indent` |

If `true`, adds whitespace up to the first open bracket when indenting.

`tab_size`

| Type | int |
|------|------|
| **Default** | 4 |

Size of a tab in spaces.

`translate_tabs_to_spaces`

| Type | bool |
|------|------|
| **Default** | `false` |

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

`use_tab_stops`

| Type | bool |
|------|------|
| **Default** | `true` |

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

`trim_automatic_white_space`

| Type | bool |
|------|------|
| **Default** | `true` |

Toggles deletion of whitespace added by `auto_indent`.

`detect_indentation`

| Type | bool |
|------|------|
| **Default** | `true` |

If `false`, disables detection of tabs vs. spaces whenever a buffer is loaded. If `true`, it will automatically modify `translate_tabs_to_spaces` and `tab_size`.

`draw_white_space`

| Type | enum : str |
|------|------|
| **Default** | `"selection"` |

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

`trim_trailing_white_space_on_save`

| Type | bool |
|------|------|
| **Default** | `false` |

If `true`, Sublime Text will remove whitespace from the active file before saving.

`tab_completion`

| Type | bool |
|---------|------|
| Default | `true` |

If `true`, pressing `Tab` will insert the best matching completion.

If `false`, `Tab` will only trigger snippets or insert a tab.

`Tab` can still be used to insert an explicit tab when `tab_completion` is `true`.

`auto_complete`

| Type | bool |
|---------|------|
| Default | `true` |

If `true`, completions will be shown automatically while typing.

`auto_complete_size_limit`

| Type | int |
|---------|------|
| Default | `4194304` |

In files whose size is larger than this value, auto complete will stop being automatically triggered.

`auto_complete_delay`

| Type | int |
|---------|------|
| Default | `50` |

Delay in milliseconds before the auto complete window is shown after typing.

`auto_complete_selector`

| Type | str |
|---------|------|
| Default | `"meta.tag - punctuation.definition.tag.begin, source - comment - string.quoted.double.block - string.quoted.single.block - string.unquoted.heredoc"` |

Selector denoting scopes where auto complete will be active.

`auto_complete_triggers`

| Type | [{ (enum : str) : str }, ] |
|---------|------|
| Default | `[{"characters": "<", "selector": "text.html"}]` |

Additional situations to trigger auto complete.

`auto_complete_commit_on_tab`

| Type | bool |
|---------|------|
| Default | `false` |

By default, auto complete will commit the current completion on `Enter`.

If `true`, this setting will allow you to complete on `Tab` instead.

Completing on `Tab` is generally a superior option, as it removes the ambiguity between committing the completion and inserting a newline.

`auto_complete_with_fields`

| Type | bool |
|------------|------|
| Default | `false` |
| Depends on | `auto_complete_commit_on_tab` |

If `true`, auto complete will be also shown when snippet fields are active.

`auto_complete_cycle`

---

| Type | bool |
|---|---|
| Default | `false` |

If `true`, pressing `Up` on the first item in the auto complete window will select the last item.

If `false`, the auto complete window will be closed in the same situation.

Likewise for the `Down` key when the last item is selected.

auto_close_tags

| Type | bool |
|---|---|
| Default | `true` |

If `true`, entering `</` will automatically close HTML and XML tags.

shift_tab_unindent

| Type | bool |
|---|---|
| Default | `false` |

By default, `Shift+Tab` will only unindent if the selection spans multiple lines. When pressing `Shift+Tab` at other times, it will insert a tab character. This way, tabs can be inserted when `tab_completion` is enabled.

If `true`, `Shift+Tab` will always unindent instead of inserting tabs.

copy_with_empty_selection

| Type | bool |
|---|---|
| Default | `true` |

If `true`, the `copy` and `cut` commands will operate on the current line when the selection is empty instead of doing nothing.

find_selected_text

| Type | bool |
|---|---|
| Default (All) | `true` |
| Default (OS X) | `false` |

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

auto_find_in_selection

| Type | bool |
|---|---|
| Default | `false` |

If `true`, the **Find in Selection** flag will be enabled automatically when multiple lines are selected.

drag_text

| Type | bool |
|---|---|
| Default | `true` |

If `true`, clicking on selected text will begin a drag-drop operation.

Not currently implemented under Linux.

### Visual Settings

always_show_minimap_viewport

| Type | bool |
|---|---|
| Default | `false` |

If `true`, shows a rectangle on the minimap highlighting the file's visible area. If `false`, only shows the rectangle when you hover the cursor over the minimap.

`color_scheme`

| Type | str |
|---|---|
| Default | `"Packages/Color Scheme - Default/Monokai.tmTheme"` |

Sets the colors used for text highlighting. Accepts a path relative to the `Data` directory (for example: `Packages/Color Scheme - Default/Monokai Bright.tmTheme`).

`font_face`

| Type | str |
|---|---|
| Default (Linux) | `"Monospace"` |
| Default (OS X) | `"Menlo-Regular"` |
| Default (Windows) | `"Consolas"` |

Font face to be used for editable text.

`font_size`

| Type | int |
|---|---|
| Default (All) | `10` |
| Default (OS X) | `12` |

Size of the font for editable text.

`font_options`

| Type | enum : str |
|---|---|
| Default | `[]` |

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

`gutter`

| Type | bool |
|---|---|
| Default | `true` |

Toggles display of gutter.

`line_numbers`

| Type | bool |
|---|---|
| Default | `true` |

If `true`, displays line numbers in the gutter.

`margin`

| Type | int |
|---|---|
| Default | `4` |

Spacing between the gutter and the text.

`fold_buttons`

| Type | bool |
|---|---|
| Default | `true` |

If `true`, you will see triangles next to lines that can be folded.

`fade_fold_buttons`

| Type | bool |
|---------|------|
| Default | `true` |

If `true`, the fold buttons will be hidden until you hover the mouse over the gutter.

`rulers`

| Type | float | [float,] |
|---------|------------|
| Default | `[]` |

Columns at which to display vertical lines. Rulers help to visually indicate the length of a line. Accepts a list of numeric values (such as `[79, 89, 99]`) or a single numeric value (for example, `79`).

`draw_minimap_border`

| Type | bool |
|---------|-------|
| Default | `false` |

If `true`, draws a border around the minimap's region corresponding to the the view's currently visible text. The `minimapBorder` key of the active color scheme controls the border's color.

`highlight_line`

| Type | bool |
|---------|-------|
| Default | `false` |

If `true`, highlights lines where a caret is present.

`line_padding_top`

| Type | int |
|---------|-----|
| Default | `0` |

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

`line_padding_bottom`

| Type | int |
|---------|-----|
| Default | `0` |

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

`caret_style`

| Type | enum : str |
|---------|--------------|
| Default | `"smooth"` |

Determines the style of the caret (text insertion point).

Valid values: `smooth`, `phase`, `blink`, `solid`.

`caret_extra_top`

| Type | int |
|---------|-----|
| Default | `0` |

Increases the size of the caret.

`caret_extra_bottom`

| Type | int |
|---------|-----|
| Default | `0` |

Increases the size of the caret.

`caret_extra_width`

| Type | int |
|---------|-----|
| Default | 0 |

Increases the width of the caret.

`scroll_past_end`

| Type | bool |
|------------------|-------|
| Default (All) | true |
| Default (OS X) | false |

If `true`, Sublime Text will leave a wide, empty margin between the last line and the bottom of the window.

`word_wrap`

| Type | bool \| auto |
|---------|------------|
| Default | "auto" |

If `true`, disables horizontal scrolling. If set to `auto`, will be disabled for source code and enabled otherwise.

`wrap_width`

| Type | int |
|------------|-----------|
| Default | 0 |
| Depends on | word_wrap |

If greater than 0, wraps long lines at the specified column as opposed to the window width.

`indent_subsequent_lines`

| Type | bool |
|------------|-----------|
| Default | true |
| Depends on | word_wrap |

If `false`, wrapped lines will not be indented. Only has effect if `word_wrap` is set to `true`.

`draw_centered`

| Type | bool |
|---------|-------|
| Default | false |

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

`match_brackets`

| Type | bool |
|---------|------|
| Default | true |

If `false`, disables bracket highlighting for brackets enclosing the caret.

`match_brackets_content`

| Type | bool |
|---------|------|
| Default | true |

If `true`, the nearest bracket pair surrounding the caret will be highlighted. Otherwise, the caret has to be next to a bracket for highlighting to occur.

`match_brackets_square`

| Type | bool |
|------------|----------------|
| Default | true |
| Depends on | match_brackets |

If `false`, stops highlighting square brackets. Only has effect if `match_brackets` is `true`.

`match_brackets_braces`

| Type | bool |
|---|---|
| Default | `true` |
| Depends on | `match_brackets` |

If `false`, stops highlighting curly brackets. Only has effect if `match_brackets` is `true`.

`match_brackets_angle`

| Type | bool |
|---|---|
| Default | `false` |
| Depends on | `match_brackets` |

If `false`, stops highlighting angle brackets. Only has effect if `match_brackets` is `true`.

`match_tags`

| Type | bool |
|---|---|
| Default | `true` |

If `true`, enables visualization of the matching tag in HTML and XML content.

`match_selection`

| Type | bool |
|---|---|
| Default | `true` |

If `true`, highlights other occurrences of the selected text.

`draw_indent_guides`

| Type | bool |
|---|---|
| Default | `true` |

If `true`, draws lines at every indentation level.

The color of the indentation guides is controlled via the *.tmTheme* settings: `guide`, `activeGuide` and `stackGuide`.

`indent_guide_options`

| Type | enum : str |
|---|---|
| Default | `["draw_normal"]` |
| Depends on | `draw_indent_guides` |

Valid options: `draw_normal`, `draw_active`.

The `draw_active` option will cause the indent guide containing the caret to be shown in a different color.

`show_definitions`

| Type | bool |
|---|---|
| Depends on | `index_files` |

If `true`, hovering over a word will show a popup listing all possible locations for the symbol.

`tree_animation_enabled`

| Type | bool |
|---|---|
| Default | `true` |

If `true`, animations will be shown in the sidebar when expanding or collapsing folders.

`animation_enabled`

---

| Type | bool |
|---|---|
| Default | `true` |

If `true`, animations will be shown throughout the application.

`hightlight_modified_tabs`

| Type | bool |
|---|---|

If `true`, makes tabs with unsaved changes more visible.

`bold_folder_labels`

| Type | bool |
|---|---|
| Default | `false` |

If `true`, folder names in the side bar will be bold.

`use_simple_full_screen`

| Type | bool |
|---|---|
| Default | `false` |

**OS X only.**

If `true`, disables Lion-style full-screen support.

Sublime Text must be restarted after changing this setting for it to take effect.

`gpu_window_buffer`

| Type | enum : (bool \| str) |
|---|---|
| Default | `"auto"` |

**OS X only.**

Valid values: `true`, `false`, `auto`.

If `auto`, it will enable the setting when running on a screen 2560 pixels or wider (for example, a Retina display).

If `true`, OpenGL is used to accelerate drawing.

Sublime Text must be restarted after changing this setting for it take effect.

`overlay_scroll_bars`

| Type | enum : str |
|---|---|
| Default | `"system"` |

Valid values: `system`, `enabled`, `disabled`.

`enable_tab_scrolling`

| Type | bool |
|---|---|
| Default | `true` |

If `true`, tabs will scroll left and right instead of simply shrinking when there are too many to show on the tab bar.

`show_encoding`

| Type | bool |
|---|---|
| Default | `false` |

If `true`, shows the active file's encoding in the status bar.

`show_line_endings`

---

| Type | bool |
|---|---|
| Default | false |

If `true`, shows the active file's type of line endings in the status bar.

`remember_full_screen`

| Type | bool |
|---|---|
| Default | false |

If `true`, allows Sublime Text to start in full-screen mode if it was exited in full-screen mode.

If `false`, Sublime Text will never start in full-screen mode.

`always_prompt_for_file_reload`

| Type | bool |
|---|---|
| Default | false |

If `true`, always prompt before reloading a file.

By default, prompting will only occur if a file has unsaved changes.

`create_window_at_startup`

| Type | bool |
|---|---|
| Default | true |

**OS X only.**

If `true`, an empty window will be created at startup.

`show_panel_on_build`

| Type | bool |
|---|---|
| Default | true |

If `true`, shows the Build Results panel when building.

If `false`, the Build Results panel can be shown via the `Tools --> Build Results` menu item.

`index_files`

| Type | bool |
|---|---|
| Default | true |

If `true`, file indexing parses all files in the side bar and builds an index of their symbols.

Required for Goto Definition to work.

Symbols are only available for syntaxes that define them via `.tmPreferences` files.

`index_workers`

| Type | int |
|---|---|
| Default | 0 |
| Depends on | index_files |

Sets the number of threads used for indexing.

If `0`, Sublime Text will use all available cores.

To disable indexing completely, use `index_files`.

`index_exclude_patterns`

---

| Type | [str,] |
|---|---|
| Default | `["*.log"]` |
| Depends on | `index_files` |

Indicates which files will not be indexed.

Patterns in this list accept wildcards.

## Automatic Behavior

`auto_match_enabled`

| Type | bool |
|---|---|
| Default | `true` |

Toggles automatic pairing of quotes, brackets, etc.

`save_on_focus_lost`

| Type | bool |
|---|---|
| Default | `false` |

If `true`, saves files automatically when switching to a different file or application.

`find_selected_text`

| Type | bool |
|---|---|
| Default (All) | `true` |
| Default (OS X) | `false` |

If `true`, the selected text will be copied into the find panel when the panel is opened.

`word_separators`

| Type | str |
|---|---|
| Default | `"./\\()\"'-:,.;<>~!@#$%^&*|+=[]{}`~?"` |

Characters considered to divide words for actions like advancing the cursor, etc. Not used for every context where a notion of a word separator is useful (for example, word wrapping). In some contexts, the text might be tokenized based on other criteria (for example, the syntax definition rules).

`ensure_newline_at_eof_on_save`

| Type | bool |
|---|---|
| Default | `false` |

If `true`, adds a new line at the end of the active file before saving if no new line is present.

## System and Miscellaneous Settings

`is_widget`

| Type | bool |
|---|---|
| Default | `null` |

Is set to `true` by Sublime Text if the buffer is an input field in a dialog, as opposed to a regular buffer.

`spell_check`

| Type | bool |
|---|---|
| Default | `false` |

Toggles the spell checker.

`dictionary`

| Type | bool |
|---------|------|
| Default | `"Packages/Language - English/en_US.dic"` |

Word list to be used by the spell checker. Accepts a path relative to the `Data` directory (such as `Packages/Language - English/en_US.dic`). You can add more dictionaries.

`spelling_selector`

| Type | str |
|---------------|------|
| De-<br>fault | `"markup.raw, source string.quoted - punctuation -`<br>`meta.preprocessor.c.include, source comment - source`<br>`comment.block.preprocessor, -(source, constant, keyword,`<br>`storage, support, variable, markup.underline.link, meta.tag)"` |
| De-<br>pends<br>on | `spell_check` |

Scope selector to determine which scopes will be spell checked.

`fallback_encoding`

| Type | bool |
|---------|------|
| Default | `"Western (Windows 1252)"` |

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

`default_encoding`

| Type | str |
|---------|------|
| Default | `"UTF-8"` |

Encoding used when saving new files, and files opened with an undefined encoding (for example, plain ascii files).

If a file is opened with a specific encoding (either detected or given explicitly), this setting will be ignored and the file's encoding will be used to save the file.

`enable_hexadecimal_encoding`

| Type | bool |
|---------|------|
| Default | `true` |

If `true`, files containing null bytes will be opened as hexadecimal by default.

`default_line_ending`

| Type | bool |
|---------|------|
| Default | `"system"` |

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

`tab_completion`

| Type | bool |
|---------|------|
| Default | `true` |

Determines whether pressing `Tab` will insert completions.

### Build and Error Navigation Settings

`result_file_regex`

| Type | str |
|---|---|
| Default | `null` |

Regular expression used to extract file names from build system output printed to a view or output panel. Follows the rules for error capturing in build systems.

`result_line_regex`

| Type | str |
|---|---|
| Default | `null` |

Regular expression used to extract line information from build system output printed to a view or output panel. Follows the rules for error capturing in build systems.

`result_base_dir`

| Type | str |
|---|---|
| Default | `null` |

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

`build_env`

| Type | bool \| {str : str} |
|---|---|
| Default | `false` |

List of paths to add to build systems by default.

### File and Directory Settings

`default_dir`

| Type | str |
|---|---|
| Default | `null` |

Sets the default save folder for the view. If the value of this setting points to an existing folder, you will be prompted to save the file to that folder when you try to save the file.

`atomic_save`

| Type | bool |
|---|---|
| Default | `false` |

If `true`, Sublime Text will save by writing to an alternate file and then renaming it over the original file.

Atomic save attempts to reduce data loss by only removing the orginal file when the new file has been successfully written. Because the a file is first created and then renamed instead of being updated in place, many file listeners and virtual file systems will get confused. For this reason, it's disabled by default.

### Input Settings

`command_mode`

| Type | bool |
|---|---|
| Default | `false` |

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

```
move_to_limit_on_up_down
```

| Type | bool |
|---|---|
| **Default (All)** | `false` |
| **Default (OS X)** | `true` |

If `true` and the caret is on the first line, pressing the up arrow will move the caret to the beginning of the line. If the caret is on the last line, the down arrow will move the caret to the end of the last line.

### 2.9.2 Symbols

#### Overview

Sublime Text provides basic support for *symbol navigation* (jumping to class and function definitions, etc.). Symbol navigation can be enabled for any type of file.

The symbol navigation framework in Sublime Text is strictly text-based. No lexical or syntactical analysis is performed.

#### Format

Symbols are defined using metadata files. Because symbol definition files are commonly required by packages, they are discussed separately in this page for convenience.

Just as regular metadata files, symbol definition files have the `.tmPreferences` extension and use the Property List format. The file name is ignored by Sublime Text.

**See also:**

**metadata** Detailed documentation on metadata files.

#### Defining Symbols

Sublime Text features two types of symbol list: a local symbol list (active file) and a global symbol list (project-wide). Using symbol definition files, you can target both individually.

Symbol definition files use scope selectors to capture symbols in source code files.

Several symbol definition files can coexist in the same package. For example, two symbol definition files could work in tandem: one would define all symbols, and a second one could selectively hide some of them if they were uninteresting for users.

Let's see an example of a symbol definition file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList
<plist version="1.0">
<dict>
   <key>name</key>
   <string>Symbol List</string>
   <key>scope</key>
   <string>source.python meta.function.python, source.python meta.class.python</string>
   <key>settings</key>
   <dict>
      <key>showInSymbolList</key>
      <integer>1</integer>
   </dict>
```

```
</dict>
</plist>
```

Using the file above, Sublime Text would scan source code files for scope names `source.python meta.function.python` and `source.python meta.class.python`, and text within would be indexed as symbols. The `showInSymbolList` setting tells Sublime Text to use the local symbol list.

### Text Transformations

It is possible to apply transformations to symbols before they are displayed to the user. Symbol transformations consist of text substitutions defined as regular expressions using the Oniguruma syntax.

This is an example of a text substitution:

```
s/class\s+([A-Za-z_][A-Za-z0-9_]*.+?\)?)(\:|$)/$1/g;
```

In this case, a captured symbol such as `class FooBar(object)` would show up as `FooBar(object)` in the symbol list.

Let's expand our previous example to use a symbol transformation:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyLis
<plist version="1.0">
<dict>
   <key>name</key>
   <string>Symbol List</string>
   <key>scope</key>
   <string>source.python meta.function.python, source.python meta.class.python</string>
   <key>settings</key>
   <dict>
      <key>showInSymbolList</key>
      <integer>1</integer>
      <key>symbolTransformation</key>
      <string>
         s/class\s+([A-Za-z_][A-Za-z0-9_]*.+?\)?)(\:|$)/$1/g;
         s/def\s+([A-Za-z_][A-Za-z0-9_]*\()(?:(.{0,40}?\))|((.{40}).+?\)))(\:)/$1(?2:$2)(?3:$4...\))/
      </string>
   </dict>
</dict>
</plist>
```

### Structure of a Symbol Definition File

All metadata files share the same topmost structure, which is inherited from the Property List format.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/Propert
<plist version="1.0">
<dict>
   ...
</dict>
</plist>
```

These are all the valid elements in a symbol definition file:

**name** Optional. Name of the symbol definition. Ignored by Sublime Text.

```
<key>name</key>
<string>Some arbitrary name goes here</string>
```

**scope** Comma-separated list of scope names that Sublime Text will use to capture symbols in files.

```
<key>scope</key>
<string>source.python meta.function.python, source.python meta.class.python</string>
```

**settings** Required. A container for settings.

```
<key>settings</key>
<dict>
    ...
</dict>
```

**uuid** Optional. A unique identifier for the file. Ignored by Sublime Text.

```
<key>uuid</key>
<string>BC062860-3346-4D3B-8421-C5543F83D11F</string>
```

### `settings` Subelements

**showInSymbolList** Optional. Links symbols to the local symbol list. Valid values are `0` or `1`. If `0`, the corresponding symbols will not be displayed.

```
<key>showInSymbolList</key>
<integer>1</integer>
```

**showInIndexedSymbolList** Optional. Links symbols to the global symbol list. Valid values are `0` or `1`. If `0`, the corresponding symbols will not be displayed.

```
<key>showInIndexedSymbolList</key>
<integer>1</integer>
```

**symbolTransformation** Optional. Targets the local symbol list. Semicolon-separated list of text substitutions expressed as regular expressions using the Oniguruma syntax. Whitespace between substitution instructions is ignored.

```
<key>symbolTransformation</key>
<string>
    s/class\s+([A-Za-z_][A-Za-z0-9_]*.+?\)?)(\:|$)/$1/g;
    s/def\s+([A-Za-z_][A-Za-z0-9_]*\()(?:(.{0,40}?\))|((.{40}).+?\)))(\:)/$1(?2:$2)(?3:$4...\))/g
</string>
```

**symbolIndexTransformation** Optional. Targets the global symbol list. Semicolon-separated list of text substitutions expressed as regular expressions using the Oniguruma syntax. Whitespace between substitution instructions is ignored.

```
<key>symbolIndexTransformation</key>
<string>
    s/class\s+([A-Za-z_][A-Za-z0-9_]*.+?\)?)(\:|$)/$1/g;
    s/def\s+([A-Za-z_][A-Za-z0-9_]*\()(?:(.{0,40}?\))|((.{40}).+?\)))(\:)/$1(?2:$2)(?3:$4...\))/g
</string>
```

### Navigating Symbols

Once symbols are defined, you can navigate them using standard key bindings:

| F12 | Go to definition |
|------|------|
| Ctrl+R | Show local symbol list |
| Ctrl+Shift+R | Show global symbol list |

**See also:**

*Goto Anything* Browsing symbols using Goto Anything.

### 2.9.3 Command Palette

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

#### File Format of `.sublime-commands` Files

| Format | JSON (with comments) |
|------|------|
| **Extension** | `.sublime-commands` |
| **Name** | Any |
| **Location** | Any under `Packages` |
| **Content** | Command palette item |

#### Example

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"}
]
```

#### Command Palette Item

These are the elements that can be included in a `.sublime-commands` item:

**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.

#### 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.

### 2.9.4 Python API

**See also:**

Official Documentation  API documentation.


**Missing in the official docs**

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


**Index**

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 tile-based panel layout of view groups.

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

      **Returns** None

      Expects a dictionary like this:

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

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

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

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

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

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

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

---

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

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

---

**Examples**:

```python
# 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]]
})
```

```python
# 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]]
})
```

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

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

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

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

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

> **Parameters**
>
> - **point** (*int*) – Point in the view whose scope the selector should be matched against.
> - **selector** (*str*) – A scope selector.
>
> **Returns bool** Whether the selector matches or not.

Equivalent to:

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

---

**sublime_plugin module**

class sublime_plugin.**EventListener**

> **on_query_completions**(*view*, *prefix*, *locations*)
>> Called whenever the completion list is requested.
>>
>> This accounts for all views and all windows, so in order to provide syntax-specific completions you should test the current scope of locations with *match_selector()*.
>>
>> **view**  A *View* instance for which the completions should be made.
>>
>> **prefix**  The text entered so far. This is only until the next word separator.
>>
>> **locations**  Array of points in view where the completion should be inserted. This can be interpreted as the current selection.
>>
>>> If you want to handle completions that depend on word separator characters you need to test each location individually. See *Completions with multiple cursors* on how Sublime Text handles completions with multiple cursors.
>>
>> *Return value*  Expects two (three) formats for return values:
>>
>>> 1. [[*trigger*, *contents*], ...]
>>>
>>>> A **list** of completions similar to completions-trigger-based 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. This consists of any word in the current document that is longer than 3 characters.
>>>>
>>>> **sublime.INHIBIT_EXPLICIT_COMPLETIONS**  Prevents Sublime Text from suggesting entries from .sublime-completions files. Therefore, with this flag set, it will only show completions that are returned by plugins from their on_query_completions methods (along with word completions unless the above flag is also set.)
>>>>
>>>> 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

### 2.9.5 Keyboard Shortcuts - Windows/Linux

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

---

#### Editing

| Keypress | Command |
| --- | --- |
| Ctrl + X | Cut 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 + + K | Delete Line |
| 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 |
| Alt + + W | Wrap Selection in html tag |
| Alt + . | Close current html tag |

#### Windows

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

**Linux**

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

### 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 |

### General

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

### Find/Replace

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

### Tabs

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

### Split window

| Keypress | Command |
|---|---|
| Alt + + 1 | Revert view to single column |
| Alt + + 2 | Split view into two columns |
| Alt + + 3 | Split view into three columns |
| Alt + + 4 | Split view into four columns |
| Alt + + 5 | Set view to grid (4 groups) |
| Alt + + 8 | Split view into two rows |
| Ctrl + [NUM] | Jump to group where num is 1-4 |
| Ctrl + + [NUM] | Move file to specified group where num is 1-4 |

**Bookmarks**

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

**Text manipulation**

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

## 2.9.6 Keyboard Shortcuts - OSX

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

**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 to select next occurrence |
| + + G | Select all occurrences of current selection |
| + + ↑ | Extra cursor on the line above |
| + + ↓ | Extra cursor on the line below |
| + M | Jump to closing parentheses Repeat to jump to opening parentheses |
| + + M | Select all contents of the current parentheses |
| + A, + Left | Move to beginning of line |
| + E, + Right | Move to end of line |
| + 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 |
| | Continued on next page |

Table  2.1 – continued from previous page

| Keypress | Command |
|----------|---------|
| + + Up | Column selection up |
| + + Down | Column selection down |
| + + W | Wrap Selection in html tag |
| + + K | Delete current line of cursor |

### Navigation/Goto Anywhere

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

### General

| Keypress | Command |
|----------|---------|
| + + P | Command Palette |
| + ' | Python Console |
| + + F | Toggle fullscreen mode |
| + + + F | Toggle distraction-free mode |
| + K,  + B | Toggle side bar |
| + + P | Show scope in status bar |

### Find/Replace

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

### Scrolling

| Keypress | Command |
|----------|---------|
| + V | Scroll down one page |
| + L | Center current line vertically in page |
| + Down | Scroll to end of file |
| + Up | Scroll to start of file |

**Tabs**

| Keypress | Command |
|---|---|
| + + t | Open last closed tab |
| + [NUM] | Jump to tab in current group where num is 1-9 |
| + 0 | Jump to 10th tab in current group |
| + + [ | Cycle left through tabs |
| + + ] | Cycle right through tabs |
| ^ + Tab | Cycle up through recent tabs |
| + ^ + Tab | Cycle down through recent tabs |
| | Find in files |

**Split window**

| Keypress | Command |
|---|---|
| + + 1 | Revert view to single column |
| + + 2 | Split view into two columns |
| + + 3 | Split view into three columns |
| + + 4 | Split view into four columns |
| + + 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 |

**Bookmarks**

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

**Text manipulation**

| Keypress | Command |
|---|---|
| + K, + U | Transform to Uppercase |
| + K, + L | Transform to Lowercase |
| + + up, + + down | Clip text upwards / downwards |

# 2.10

## S

## C

core packages, **49**

## D

default packages, **49**

## E

EventListener (class in sublime_plugin), 74

## I

installed packages, **49**

## M

match_selector() (sublime.View method), 73

## O

on_query_completions()   (sublime_plugin.EventListener
            method), 74
override packages, **50**

## S

set_layout() (sublime.Window method), 72
shipped packages, **49**
sublime (module), 72
sublime_plugin (module), 73

## U

user packages, **49**

## V

View (class in sublime), 73

## W

Window (class in sublime), 72