PYTHON

How to Create a Sublime Text 2 Plugin

by Will Bond 14 Nov 2011 49 Comments

9 5 49

Sublime Text 2 is a highly customizable text editor that has been increasingly capturing the attention of coders looking for a tool that is powerful, fast and modern. Today, we're going to recreate my popular Sublime plugin that sends CSS through the Nettuts+ Prefixr API for easy cross-browser CSS.

When finished, you'll have a solid understanding of how the Sublime Prefixr plugin is written, and be equipped to start writing your own plugins for the editor!

Preface: Terminology and Reference Material

The extension model for Sublime Text 2 is fairly full-featured.

The extension model for Sublime Text 2 is fairly full-featured. There are ways to change the syntax highlighting, the actual chrome of the editor and all of the menus. Additionally, it is possible to create new build systems, auto-completions, language definitions, snippets, macros, key bindings, mouse bindings and plugins. All of these different types of modifications are implemented via files which are organized into packages.

A package is a folder that is stored in your `Packages` directory. You can access your Packages directory by clicking on the *Preferences > Browse Packages…* menu entry. It is also possible to bundle a package into a single file by creating a zip file and changing the extension to `.sublime-package`. We'll discuss packaging a bit more further on in this tutorial.

Sublime comes bundled with quite a number of different packages. Most of the bundled packages are language specific. These contain language definitions, auto-completions and build systems. In addition to the language packages, there are two other packages: `Default` and `User`. The `Default` package contains all of the standard key bindings, menu definitions, file settings and a whole bunch of plugins written in Python. The `User` package is special in that it is always loaded last. This allows users to override defaults by customizing files in their `User` package.

> *During the process of writing a plugin, the Sublime Text 2 API referencewill be essential.*

During the process of writing a plugin, the Sublime Text 2 API referencewill be essential. In addition, the `Default` package acts as a good reference for figuring out how to do things and what is possible. Much of the functionality of the editor is exposed via *commands*. Any operation other than typing characters is accomplished via commands. By viewing the *Preferences > Key Bindings - Default*menu entry, it is possible to find a treasure trove of built-in functionality.

Now that the distinction between a plugin and package is clear, let’s begin writing our plugin.

Step 1 - Starting a Plugin

Sublime comes with functionality that generates a skeleton of Python code needed to write a simple plugin. Select the *Tools > NewPlugin…*menu entry, and a new buffer will be opened with this boilerplate.

```python
import sublime, sublime_plugin

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

Here you can see the two Sublime Python modules are imported to allow for use of the API and a new command class is created. Before editing this and starting to create our own plugin, let’s save the file and trigger the built in functionality.

When we save the file we are going to create a new package to store it in. Press **ctrl+s** (Windows/Linux) or **cmd+s** (OS X) to save the file. The save dialog will open to the `User` package. Don't save the file there, but instead browse up a folder and create a new folder named `Prefixr`.

```
Packages/
…
- OCaml/
- Perl/
- PHP/
- Prefixr/
- Python/
- R/
- Rails/
…
```

Now save the file inside of the Prefixr folder as `Prefixr.py`. It doesn't actually matter what the filename is, just that it ends in `.py`. However, by convention we will use the name of the plugin for the filename.

Now that the plugin is saved, let's try it out. Open the Sublime console by pressing **ctrl+`**. This is a Python console that has access to theAPI. Enter the following Python to test out the new plugin:

```
view.run_command('example')
```

You should see `Hello World` inserted into the beginning of the plugin file. Be sure to undo this change before we continue.

Step 2 - Command Types and Naming

For plugins, Sublime provides three different types of commands.

- *Text commands* provide access to the contents of the selected file/buffer via a `View` object
- *Windowcommands* provide references to the current window via a `Window` object
- *Application commands* do not have a reference to any specific window or file/buffer and are more rarely used

Since we will be manipulating the content of a CSS file/buffer with this plugin, we are going to use the `sublime_plugin.TextCommand` class as the basis of our custom Prefixr command. This brings us to the topic of naming command classes.

In the plugin skeleton provided by Sublime, you'll notice the class:

```
class ExampleCommand(sublime_plugin.TextCommand):
```

When we wanted to run the command, we executed the following code in the console:

```
view.run_command('example')
```

Sublime will take any class that extends one of the `sublime_plugin` classes (`TextCommand`, `WindowCommand` or `ApplicationCommand`), remove the suffix `Command` and then convert the `CamelCase` into `underscore_notation` for the command name.

Thus, to create a command with the name `prefixr`, the class needs to be `PrefixrCommand`.

```
class PrefixrCommand(sublime_plugin.TextCommand):
```

Step 3 - Selecting Text

One of the most useful features of Sublime is the ability to have multiple selections.

Now that we have our plugin named properly, we can begin the process of grabbing CSS from the current buffer and sending it to the Prefixr API. One of the most useful features of Sublime is the ability to have multiple selections. As we are grabbing the selected text, we need to write our plug into handle not just the first selection, but all of them.

Since we are writing a text command, we have access to the current view via `self.view`. The `sel()` method of the `View` object returns an iterable `RegionSet` of the current selections. We start by scanning through these for curly braces. If curly braces are not present we can expand the selection to the surrounding braces to ensure the whole block is prefixed. Whether or not our selection included curly braces will also be useful later to know if we can tweak the whitespace and formatting on the result we getback from the Prefixr API.

```
braces = False
sels = self.view.sel()
for sel in sels:
    if self.view.substr(sel).find('{') != -1:
        braces = True
```

This code replaces the content of the skeleton `run()` method.

If we did not find any curly braces we loop through each selection and adjust the selections to the closest closing curly brace. Next, we use the built-in command `expand_selection` with the `to` arg set to `brackets` to ensure we have the complete contents of each CSS block selected.

```
if not braces:
    new_sels = []
    for sel in sels:
        new_sels.append(self.view.find('\}', sel.end()))
    sels.clear()
    for sel in new_sels:
        sels.add(sel)
    self.view.run_command("expand_selection", {"to": "brackets"})
```

If you would like to double check your work so far, please compare the source to the file `Prefixr-1.py` in the source code zip file.

Step 4 - Threading

To prevent a poor connection from interrupting other work, we need to make sure that the Prefixr API calls are happening in the background.

At this point, the selections have been expanded to grab the full contents of each CSS block. Now, we need to send them to the Prefixr API. This is a simple HTTP request, which we are going to use the `urllib` and `urllib2` modules for. However, before we start firing off web requests, we need to think about how a potentially laggy web request could affect the performance of the editor. If, for some reason, the user is on a high-latency, or slow connection, the requests to the Prefixr

API could easily take a couple of seconds or more.

To prevent a poor connection from interrupting other work, we need to make sure that the Prefixr API calls are happening in the background. If you don't know anything about threading, a very basic explanation is that threads are a way for a program to schedule multiple sets of code to run seemingly at the same time. It is essential in our case because it lets the code that is sending data to, and waiting for a response from, the Prefixr API from preventing the rest of the Sublime user interface from freezing.

Step 5 - Creating Threads

We will be using the Python `threading` module to create threads. To use the threading module, we create a new class that extends `threading.Thread` called `PrefixrApiCall`. Classes that extend `threading.Thread` include a `run()` method that contains all code to be executed in the thread.

```
class PrefixrApiCall(threading.Thread):
    def __init__(self, sel, string, timeout):
        self.sel = sel
        self.original = string
        self.timeout = timeout
        self.result = None
        threading.Thread.__init__(self)

    def run(self):
        try:
            data = urllib.urlencode({'css': self.original})
            request = urllib2.Request('http://prefixr.com/api/index.php', dat
                headers={"User-Agent": "Sublime Prefixr"})
            http_file = urllib2.urlopen(request, timeout=self.timeout)
            self.result = http_file.read()
            return

        except (urllib2.HTTPError) as (e):
            err = '%s: HTTP error %s contacting API' % (__name__, str(e.code))
        except (urllib2.URLError) as (e):
            err = '%s: URL error %s contacting API' % (__name__, str(e.reason)

        sublime.error_message(err)
        self.result = False
```

Here we use the thread `__init__()` method to set all of the values that will be needed during the web request. The `run()` method contains the code toset up and execute the HTTP request for the Prefixr API. Since threads operate concurrently with other code, it is not possible to directly return values. Instead we set `self.result` to the result of the call.

Since we just started using some more modules in our plugin, we must add them to the import statements at the top of the script.

```
import urllib
import urllib2
import threading
```

Now that we have a threaded class to perform the HTTP calls, we need to create a thread for each selection. To do this we jump back into the `run()` method of our `PrefixrCommand` class and use the following loop:

```
threads = []
for sel in sels:
    string = self.view.substr(sel)
    thread = PrefixrApiCall(sel, string, 5)
    threads.append(thread)
    thread.start()
```

We keep track of each thread we create and then call the `start()` method to start each.

If you would like to double check your work so far, please compare the source to the file `Prefixr-2.py` in the source code zip file.

Step 6 - Preparing for Results

Now that we've begun the actual Prefixr API requests we need toset up a few last details before handling the responses.

First, we clear all of the selections because we modified them earlier. Later we will set them back to a reasonable state.

```
self.view.sel().clear()
```

In addition we start a new `Edit` object. This groups operations for undo and redo. We specify that we are creating a group for the `prefixr` command.

```
edit = self.view.begin_edit('prefixr')
```

As the final step, we call a method we will write next that will handle the result of the API requests.

```
self.handle_threads(edit, threads, braces)
```

Step 7 - Handling Threads

At this point our threads are running, or possibly even completed. Next, we need to implement the `handle_threads()` method we just referenced. This method is going to loop through the list of threads and look for threads that are no longer running.

```
def handle_threads(self, edit, threads, braces, offset=0, i=0, dir=1):
    next_threads = []
    for thread in threads:
        if thread.is_alive():
            next_threads.append(thread)
            continue
        if thread.result == False:
            continue
        offset = self.replace(edit, thread, braces, offset)
    threads = next_threads
```

If a thread is still alive, we add it to the list of threads to check again later. If the result was a failure, we ignore it, however for good results we call a new `replace()` method that we'll be writing soon.

If there are any threads that are still alive, we need to check those again shortly. In addition, it is a nice user interface enhancement to provide an activity indicator to show that our plugin is still running.

```
if len(threads):
        # This animates a little activity indicator in the status area
        before = i % 8
        after = (7) - before
        if not after:
            dir = -1
        if not before:
            dir = 1
        i += dir
        self.view.set_status('prefixr', 'Prefixr [%s=%s]' % \
            (' ' * before, ' ' * after))

        sublime.set_timeout(lambda: self.handle_threads(edit, threads,
            braces, offset, i, dir), 100)
        return
```

The first section of code uses a simple integer value stored in the variable `i` to move an `=` back and forth between two brackets. The last part is the most important though. This tells Sublime to run the `handle_threads()` method again, with new values, in another 100 milliseconds. This is just like the `setTimeout()` function in JavaScript.

> *The `lambda` keyword is a feature of Python that allows us to create a new unnamed, or anonymous, function.*

The `sublime.set_timeout()` method requires a function or method and the number of milliseconds until it should be executed. Without `lambda` we could tell it we wanted to run `handle_threads()`, but we would not be able to specify the parameters.

If all of the threads have completed, we don't need to set another timeout, but instead we finish our undo group and update the user interface to let the user know everything is done.

```
self.view.end_edit(edit)

self.view.erase_status('prefixr')
selections = len(self.view.sel())
sublime.status_message('Prefixr successfully run on %s selection%s' %
    (selections, '' if selections == 1 else 's'))
```

If you would like to double check your work so far, please compare the source to the file `Prefixr-3.py` in the source code zip file.

Step 8 - Performing Replacements

With our threads handled, we now just need to write the code that replaces the original CSS with the result from the Prefixr API. As we referenced earlier, we are going to write a method called `replace()`.

This method accepts a number of parameters, including the `Edit` object for undo, the thread that grabbed the result from the Prefixr API, if the original selection included braces, and finally the selection offset.

```
def replace(self, edit, thread, braces, offset):
    sel = thread.sel
    original = thread.original
    result = thread.result

    # Here we adjust each selection for any text we have already inserted
    if offset:
        sel = sublime.Region(sel.begin() + offset,
            sel.end() + offset)
```

The offset is necessary when dealing with multiple selections. When we replace a block of CSS with the prefixed CSS, the length of that block will increase. The offset ensures we are replacing the correct content for subsequent selections since the text positions all shift upon each replacement.

The next step is to prepare the result from the Prefixr API to be dropped in as replacement CSS. This includes converting line endings and indentation to match the current document and original selection.

```
result = self.normalize_line_endings(result)
(prefix, main, suffix) = self.fix_whitespace(original, result, sel,
    braces)
self.view.replace(edit, sel, prefix + main + suffix)
```

As a final step we set the user’s selection to include the end of the last line of the new CSS we inserted, and then return the adjusted offset to use for any further selections.

```
end_point = sel.begin() + len(prefix) + len(main)
```

```
self.view.sel().add(sublime.Region(end_point, end_point))

return offset + len(prefix + main + suffix) - len(original)
```

If you would like to double check your work so far, please compare the source to the file `Prefixr-4.py` in the source code zip file.

Step 9 - Whitespace Manipulation

We used two custom methods during the replacement process to prepare the new CSS for the document. These methods take the result of Prefixr and modify it to match the current document.

`normalize_line_endings()` takes the string and makes sure it matches the line endings of the current file. We use the `Settings` class from the Sublime API to get the proper line endings.

```
def normalize_line_endings(self, string):
    string = string.replace('\r\n', '\n').replace('\r', '\n')
    line_endings = self.view.settings().get('default_line_ending')
    if line_endings == 'windows':
        string = string.replace('\n', '\r\n')
    elif line_endings == 'mac':
        string = string.replace('\n', '\r')
    return string
```

The `fix_whitespace()` method is a little more complicated, but does the same kind of manipulation, just for the indentation and whitespace in the CSS block. This manipulation only really works with a single block of CSS, so we exit if one or more braces was included in the original selection.

```
def fix_whitespace(self, original, prefixed, sel, braces):
    # If braces are present we can do all of the whitespace magic
    if braces:
        return ('', prefixed, '')
```

Otherwise, we start by determining the indent level of the original CSS. This is done by searching for whitespace at the beginning of the selection.

```
(row, col) = self.view.rowcol(sel.begin())
indent_region = self.view.find('^\s+', self.view.text_point(row, 0))
if self.view.rowcol(indent_region.begin())[0] == row:
    indent = self.view.substr(indent_region)
else:
    indent = ''
```

Next we trim the whitespace from the prefixed CSS and use the current view settings to indent the trimmed CSS to the original level using either tabs or spaces depending on the current editor settings.

```
prefixed = prefixed.strip()
prefixed = re.sub(re.compile('^\s+', re.M), '', prefixed)

settings = self.view.settings()
use_spaces = settings.get('translate_tabs_to_spaces')
tab_size = int(settings.get('tab_size', 8))
indent_characters = '\t'
if use_spaces:
    indent_characters = ' ' * tab_size
prefixed = prefixed.replace('\n', '\n' + indent + indent_characters)
```

We finish the method up by using the original beginning and trailing white space to ensure the new prefixed CSS fits exactly in place of the original.

```
match = re.search('^(\s*)', original)
prefix = match.groups()[0]
match = re.search('(\s*)\Z', original)
suffix = match.groups()[0]

return (prefix, prefixed, suffix)
```

With the `fix_whitespace()` method we used the Python regular expression (re)module, so we need to add it to the list of imports at the top of the script.

```
import re
```

And with this, we've completed the process of writing the `prefixr` command.The next step it to make the command easy to run by providing a keyboard shortcut and a menu entry.

Step 10 - Key Bindings

Most of the settings and modifications that can be made to Sublime are done via JSON files, and this is true for key bindings. Key bindings are usually OS-specific, which means that three key bindings files will need to be created for your plugin. The files should be named `Default (Windows).sublime-keymap`, `Default (Linux).sublime-keymap` and `Default (OSX).sublime-keymap`.

```
Prefixr/
...
- Default (Linux).sublime-keymap
- Default (OSX).sublime-keymap
- Default (Windows).sublime-keymap
- Prefixr.py
```

The `.sublime-keymap` files contain a JSON array that contains JSON objects to specify the key bindings. The JSON objects must contain a `keys` and `command` key, and may also contain a `args` key if the command requires arguments. The hardest part about picking a key binding is to ensure the key binding is not already used. This can be done by going to the *Preferences > Key Bindings – Default* menu entry and searching for the keybinding you wish to use. Once you've found a suitably unused binding, add it to your `.sublime-keymap` files.

```
[
    {
        "keys": ["ctrl+alt+x"], "command": "prefixr"
    }
]
```

Normally the Linux and Windows key bindings are the same. The cmd key on OS Xis specified by the string `super` in the `.sublime-keymap` files. When porting a key binding across OSes, it is common for the `ctrl` key onWindows and Linux to be swapped out for `super` on OS X. This may not, however, always be the most natural hand movement, so if possible try and test your keybindings out on a real keyboard.

Step 11 - Menu Entries

One of the cooler things about extending Sublime is that it is possible to add items to the menu structure by creating `.sublime-menu` files. Menufiles must be named specific names to indicate what menu they affect:

- `Main.sublime-menu` controls the main program menu
- `Side Bar.sublime-menu` controls the right-click menu on a file or folder in the sidebar
- `Context.sublime-menu` controls the right-click menu on a file being edited

There are a whole handful of other menu files that affect various other menus throughout the interface. Browsing through the *Default* package is the easiest way to learn about all of these.

For Prefixr we want to add a menu item to the *Edit* menu and some entries to the *Preferences* menu for settings. The following example is the JSON structure for the *Edit* menu entry. I've omitted the entries for the *Preferences* menu since they are fairly verbose being nested a few levels deep.

```
[
{
    "id": "edit",
    "children":
    [
        {"id": "wrap"},
        { "command": "prefixr" }
    ]
}
]
```

The one piece to pay attention to is the `id` keys. By specifying the `id` of an existing menu entry, it is possible to append an entry without redefining the existing structure. If you open the `Main.sublime-menu` file from the `Default` package and browse around, you can determine what `id` you want to add your entry to.

At this point your Prefixr package should look almost identical to the official version on GitHub.

Advertisement

Step 12 - Distributing Your Package

Now that you’ve taken the time to write a useful Sublime plugin, it is time to get into the hand of other users.

> *Sublime supports distributing a zip file of a package directory as a simple way to share packages. Simply zip your package folder and change the extension to `.sublime-package`. Other users may now place this into their Installed Packages directory and restart Sublime to install the package.*
>
> *Along with easy availability to lots of users, having your package available via Package Control ensures users get upgraded automatically to your latest updates.*

While this can certainly work, there is also a package manager forSublime called Package Controlthat supports a master list of packages and automatic upgrades. To get your package added to the default channel, simply host it on GitHubor BitBucket and then fork the channel file (on GitHub, or BitBucket), add your repository and send a pull request. Once the pull request is accepted, your package will be available to thousands of users using Sublime. Along with easy availability to lots of users, having your package available via Package Control ensures users get upgraded automatically to your latest updates.

If you don't want to host on GitHub or BitBucket, there is a customJSON channel/repository system that can be used to host anywhere, while still providing the package to all users. It also provides advanced functionality like specifying the availability of packages by OS. See the PackageControl page for more details.

Go Write Some Plugins!

Now that we've covered the steps to write a Sublime plugin, it is time for you to dive in! The Sublime plugin community is creating and publishing new functionality almost every day. With each release, Sublime becomes more and more powerful and versatile. The Sublime Text Forum is a great place to get help and talk with others about what you are building.

leaseweb
reliable hosting
Taste the power of Bare Metal.
No wimpy processors allowed. Our dedicated servers will rock your world.
Learn more

Advertisement

Difficulty:
Intermediate

Length:
Short

Categories:

Python | Web Development | Sublime Text

Translations Available:

Tuts+ tutorials are translated by our community members. If you'd like to translate this post into another language, let us know!

Download Attachment

View Online Demo

About Will Bond

I'm Will Bond. I built the open source Flourish PHP library and I work full time as the director of engineering at iMarc. In my spare time I hack on Noted and various open source and commercial software.

Related Courses

Perfect Workflow in Sublime Text 2

Jeffrey Way

Foundational Flask: Creating Your Own Static Blog Generator

Christopher Roach

Django Unchained

Christopher Roach

AdChoices
THANKS TO A QUICK-THINKING PMI-ACP® SHE WON'T BE LEFT HANGING.
READ THE STORY
PMI

Advertisement

Related Tutorials

Rapid Website Deployment With Django, Heroku & New Relic

Code

How to Build a Tweet Controlled RGB LCD

Computer Skills

Alternatives to Prefixr

Code

Check Out Atom, GitHub's New Development Editor

Code

Jobs

Tuts+ Copy & QA Editor

at Envato in Melbourne VIC, Australia

Backend Rails Developer

at Mitoo in San Francisco, CA, USA

Android Engineer

at Mitoo in San Francisco, CA, USA

Envato Marketplace Item

Go - Responsive Pricing & Compare Tables for WP

Category: miscellaneous

$19.00

49 Comments Nettuts+ Login

Sort by Best Share Favorite

Join the discussion…

greg • 2 years ago

It would be great if this tutorial gets an update for ST3. I guess that Will Bond is busy with package control and his other packages but after things settle it would be great to see what changes particularly in terms of threatening etc.

11 • Reply • Share ›

chadananda • a year ago

Worst tutorial ever. "See how simple plugins are, let's write a simple one to do search and replace. Ok, let's not forget to cover multi-threading." Good lord.

16 • Reply • Share ›

Andy Matthews • 2 years ago

It feels like you left parts out of your first section.

1) Using Sublime Text 2 1.4 (Windows), the plugin Boilerplate looks like this:
import sublime, sublimeplugin
class SampleCommand(sublimeplugin.TextCommand):
def run(self, view, args):
view.insert(view.size(), "Hello, World!\n")

2) It's not 100% clear quite where the package folder should be created. Should it appear alongside "Actionscript, ASP, CSS, Erlang, etc." or should it appear within the User directory? A disclaimer...when trying to save the newly created plugin file my save path was:
C:\Users\\AppData\Roaming\Sublime Text\Packages\Default\Options

So going up a single directory simply took me to the Default folder.

3) Hitting CTRL+` indeed brought up the console but when entering "view.run_command('example')" I receive an error:

>>> view.run_command('example')

see more

1 • Reply • Share ›

Carl Giesberts ➦ Andy Matthews • 2 years ago

It was a bit hard to find, but this page has the answer: http://docs.sublimetext.info/e...

ExampleCommand becomes 'example' as the string argument to run_command
AnythingElseCommand likewise becomes 'anything_else'

1 • Reply • Share ›

Keith David Bershatsky ➦ Carl Giesberts • a year ago

I recently made a similar error by using an older syntax that has since been replaced -- i.e., no longer "view.runCommand('example')]", but instead should be "view.run_command('example')".

1 • Reply • Share ›

Deven • 3 years ago

Sublime is growing and this tutorial really help me out to understand the sublime editor.

1 • Reply • Share ›

currybill • 3 years ago

Hi, I am working my way through the Tuts 'Learn to write HTML in 30 days'. On it the tutor recommends Sublime Text 2. So I downloaded it. I haven't a clue how to use it. As a front end graphic designer of some senior years I am baffled. How for instance can I view my html in my browser? It would be brilliant if you were to produce a user guide for idiots. I've just thought - maybe there's a Sublime Text for Dummies?
I did ask the forum for help but the instruction didn't work for me or I just didn't understand. I'm sure all you experts know exactly what ST2 is about but getting under the bonnet is bewildering - I used to be able to fix my old Ford but have you seen what's under the bonnet of a modern car? Best regards

.

1 • Reply • Share ›

BrettJay → currybill • 3 years ago

Start writing your markup, and save the html file and preview it in a web browser such as Firefox or Chrome. Typically working with an editor like Sublime Text (or other editors like TextMate) involves making changes in the text editor and then switching to and refreshing the page in your browser to see those changes. Tools like Firebug (in Firefox), the Element Inspector in Webkit browsers, and Dragonfly (in Opera) are useful for debugging css layouts and Javascript, and should also help you learn HTML and CSS.

2 • Reply • Share ›

Alex Hunt • a month ago

Fantastic tutorial that really got me started on my first plugin :)
https://github.com/huntie/supe...
http://hunt.ghost.io/improving...

• Reply • Share ›

HuaHero • a year ago

Thanks for this good tutorial,especially for us who have a long way to make progress in programing

• Reply • Share ›

Shawn McCool • a year ago

Ok, but this ignores all of the basics. How to analyse data structures, methods for debugging and other such basics.

Let's say you're looping through selections.. Well.. what are the available fields / methods? I don't find them on the sublime text API either.

Entry seems to be a pain in the ass due to lack of documentation.

• Reply • Share ›

Vaisagh Viswanathan • a year ago

Hi, I was trying to create my own plugin and I'm kind of getting stuck in the part where I want to add a key binding and a menu item. When I add the key binding to the default OSX keybinding file the key binding works. However, when I simply create a Default (OSX).sublime-keymap in the folder this key binding is no longer detected by Google. Am I missing some step?

• Reply • Share ›

Amir Masoud • 2 years ago

How can I get all pressed key with API?

• Reply • Share ›

Chris Bumgardner → Amir Masoud • 2 years ago

Check out my plugin I started the other day to do just that: https://github.com/cbumgard/Su...

1 • Reply • Share ›

Pablo • 2 years ago

Great tutorial, very usefull

• Reply • Share ›

Ilya • 2 years ago

how to make localization?

• Reply • Share ›

Kjell • 2 years ago

JSON is also verbose, but better thant XML

http://www.w3schools.com/json/...

```
var JSONObject = {
"name":"John Johnson",
"street":"Oslo West 16",
"age":33,
"phone":"555 1234567"};
```

Syntaxically this will also work when one creates a special DSL

```
SomeObjectName {
name John Johnson
address {
street Oslo West 16
}
```

}
age 33
phone 555 1234567
}

The simplification with a special DSL vs. JSON is large.

• Reply • Share ›

Yongning Liang • 2 years ago

COOL, love will's sublime plugins, and this tutorial!

• Reply • Share ›

Gemma W. • 2 years ago

Thanks for the tutorial. However, I'm still looking for a tutorial on how to create your own syntax highlighting colour scheme using ST2.

• Reply • Share ›

Cindy • 3 years ago

I don't know if it's relevant to this post but how do I get the Mac terminal output in the built in Console? Actually what I really wonder is how to get the CoronaSDK terminal output in the ST2 Console?

• Reply • Share ›

Dan LaManna • 3 years ago

Great tutorial! Glad Sublime is so open to plugin developers, however I did have a question regarding keybindings. I've read the tutorial and the API reference a good amount and I'm having a hard time understanding how multiple commands would work. I could only call prefixr for the run method, but no other methods within the class? In which case, each keybinding would have to refer to a class, is there a way to extend every class upon a centralized class for the plugin?

Thanks!

• Reply • Share ›

Joey • 3 years ago

" To prevent a poor connection from interrupting other work, we need to make sure that the Prefixr API calls are happening in the background. "

If you really wanna to go to a whole lot of trouble to maintain a nice experience, you'd be better off using the `view.add_regions` API (et al) than managing offsets yourself.

If you aren't familiar with `view.add_regions`, check it out, it's really useful for keeping track of regions when you know the buffer will, or may, be modified.

"Yeah, cool, I can make edits while I wait then huh? What if I actually remove one of the initial selections? I hear you say. (Leaving aside the fact that the grouped undo is by design)

There's an undocumented 3rd parameter (usually only discovered by the curious) to the `sublime.Region` constructor. It's the meta property available via `region.meta()`. It's just a humble integer, but it's the only real way to deterministically identify any region you put into `view.add_regions` when you retrieve it via `view.get_regions`.

eg. You feed in a length 4 list of regions indexed via [0, 1, 2, 3]) and you get back a length 3 list: [?, ?, ?] !? You would use the region.meta) property rather than the indexes)

see more

• Reply • Share ›

Emmanuel • 3 years ago

Excellent tutorial !

It's going far beyond that what we usually see. I think that this kind of tutorial is what some readers (average - advanced users) want ! This is not the typical tutorial which scratch the surface. It's going on the way i think what should be tutorials and that's what some users need (maybe not everyone but there's not only noobs on the www - no offense that's the reality...).

Excellent initiative !

All my respect for the job done.

PS: sorry for my english, i'm french ;)

• Reply • Share ›

Gabriel Mateu's • 3 years ago

What Alexander Hultner said describes what I feel about this awesome editor:

"I heard about Sublime from Jeff's post and never looked back."

Thanks!

• Reply • Share ›

pit • 3 years ago

hi

thank for the information

• Reply • Share ›

Liam • 3 years ago

Thanks Will for this great tutorial. I wish I had this two days again when I was making my first plugin (*sigh*) but nonetheless I persevered... (with the help of sublimetext.info and the forum's plugin development section).

I hope more people catch on and write some tutorials on how to use some other core ST2 features that are not found in the API docs, such as the autocomplete pop-up or the quick panel.

Now I just need to work on my python skills.

Thanks again.

• Reply • Share ›

Tresor Paris • 3 years ago

awesome! thanks for sharing!

• Reply • Share ›

dkbose • 3 years ago

I heard about Sublime from Jeff's post and never looked back. The best editor - thanks for tutorial on how to create a plugin. This will definitely rock it more.

• Reply • Share ›

John • 3 years ago

My friend just told me about Sublime Text. This is awesome, thank you.

• Reply • Share ›

Alexander Hultner • 3 years ago

This is a great tutorial. I've been considering testing out sublime 2 so I'll really have use of this then. Thanks a lot for the guide.

• Reply • Share ›

Johan • 3 years ago

Brilliant! This is probably one of the coolest tutorials I've seen on net.tuts ever. Extremely appreciated and useful. Any chance to follow it up with a tutorial demonstrating of how to cook your own syntax highlight files for languages not available in Sublime out of the box?

Keep it up! Cheers!

• Reply • Share ›

Fed03 → Johan • 3 years ago

ty for the awesome tutorial it will be extremely usefull^^

i look forward to see an explanatory tutorial on highlight code too.

Just to understand how to color <?php and ?> tooks me days XD

• Reply • Share ›

adumpaul • 3 years ago

Great tutorial. Thank you for sharing.

Reply • Share ›

Akkis • 3 years ago

Thank you for this tut. This is really helpful. I start using Sublime these days, but i use also Notepad++ because i want to make "All Open Documents" replaces. Does Sublime support it?

Reply • Share ›

Will Bond → Akkis • 3 years ago

Yes, as of build 2139 Sublime supports replacing across open files, folders, or combinations filtered by filename.

Reply • Share ›

Alex John • 3 years ago

I just only knew that there is "Sublime Text" editor and when i was used this editor. I feel great and made my life easy. thanks to that editor. i love it.

Reply • Share ›

Brian Temecula • 3 years ago

I wish Sublime Text 2 was a little more fine tuned for PHP. I purchased a license, and then afterwards realized there was a very irritating indentation of parenthesis. The indentation isn't part of javascript, only PHP. Why? It drives me nuts. Anyways, nice to see an article about the plugin creation.

Reply • Share ›

Will Bond → Brian Temecula • 3 years ago

There have been a few situations where the built-in indentation rules don't exactly match my preferences. If you are comfortable with editing regular expressions, you can go to Preferences > Browse Packages… and open the PHP folder. Inside there you'll find a file named "Indentation Rules.tmPreferences". This controls how indentation is increased and decreased.

Reply • Share ›

Jeffrey Way → Brian Temecula • 3 years ago

Remember that it's beta software.

Reply • Share ›

Brian Temecula → Jeffrey Way • 3 years ago

Oh, I know. I do think it's quite awesome, and have a sense of excitement every time and new update is available. Gotta love the new code folding sidebar buttons in the last one...

1 • Reply • Share ›

Sergey • 3 years ago

Great article! And great approach by taking a real world plugin as an example!

• Reply • Share ›

Jake • 3 years ago

Good tutorial, but I think that if you are trying to reach an audience of people who haven't made a ST plugin before and maybe even people who don't know Python very well, it would have been a much better to idea for the plugin to be MUCH more simple. Like 20 lines simple.

The best way to expose people to a plugin API like ST's is to have them create several small (20 lines or less) plugins that all do a variety of different (and maybe useless) things. Getting people comfortable with an API and where to look for what is the goal.

Half of the source code is dedicated to threading and http requests... So in the end, half of your audience is probably spending a bunch of time trying to understand the threading and extra Python stuff instead of learning about the API and what you can do with it, which is the purpose of the tutorial in the first place.

• Reply • Share ›

Will Bond Jake • 3 years ago

Thanks for great the feedback!

I can see that there may be some desire for a more basic tutorial on writing a plugin, but I did intend to write a more intermediate-level tutorial. While basic tutorials are great in that they make it really easy to get started, and are often appropriate, they tend to run out of useful material very quickly for those that like to jump in and try to build something themselves.

The Sublime API reference does a pretty good job of covering what functionality is present, and includes a list of bundled plugins that cover some basic examples of of extending the editor. What is missing is the big picture of putting all of the pieces together. You certainly can spend the time and explore the Default package and look at all of the different types of files to see what is going on, but with this tutorial I wanted to create a survey of what is frequently used.

From my experience in writing a number of plugins I've found that most of them end up using threading or subprocesses, so I felt it was important to at least provide an example of using threads and explain why they are necessary. This decision was made partially because of various comments and questions I have seen in the Plugin Development board on the Sublime Forum (http://www.sublimetext.com/for....

All that said, I do think that more basic, and advanced, tutorials could augment this and the other material that is currently available online. It would also be great to see some more tutorials on creating build systems, syntax definitions and auto-completions. I do need to call out that one member of the community, guillermooo, has spent a good deal of time writing additional documentation that can be read at http://sublimetext.info/.

1 • Reply • Share ›

Charles Roper → Will Bond • 3 years ago

I really liked this tutorial too - thanks Will. I also love the idea of creating a suite of several other example tutorials to demonstrate other aspects of plugin creation, syntax definition (including indenting) and so on.

• Reply • Share ›

Arturo → Will Bond • 3 years ago

This tutorial was excellent for me.

I don't know python but I am an experienced programmer in other languages so I already understand Threads and HTTP Request. I consider this tutorial great since it gave me a nice overview at how ST handles plugins and how to work with threads and http request in python. I love how this tutorial is actually useful, and not just another "draw pink elephants".

Although I agree with Jake that a more basic tut will reach a more basic audience, unfortunately there's no "One Tutorial to rule them all (devs)" :)

• Reply • Share ›

Rian Ariona • 3 years ago

awesome! thanks for sharing! :)

• Reply • Share ›

Muhammad Adnan • 3 years ago

Awesome, liked.

Advertisement

Teaching skills to millions worldwide.

- Tutorials
- Courses
- eBooks
- Jobs
- Blog

Follow Us

- Subscribe to Blog
- Follow us on Twitter
- Be a fan on Facebook
- Circle us on Google+
- Tutorials
- Courses
- eBooks

Add more features to your website such as user profiles, payment gateways, image galleries and more.

Browse WordPress Plugins

OPEN
New Services!