Writing Sublime Plugins

JOSH EARL

Writing Sublime Plugins

Josh Earl

This book is for sale at http://leanpub.com/writing-sublime-plugins

This version was published on 2014-03-28

Contents

Contact Me i

Learn more about your favorite editor ii

Introduction iii

I What is a Sublime plugin? 1

1 What is the difference between a package and a plugin? 2

2 What can a Sublime plugin do? 3

3 What restrictions does Sublime place on plugins? 15

4 Hello World 18

II Event listeners 22

5 Introducing the KeyBindingInspector plugin 23

6 Setup 24

7 Implementation overview 27

8 What is an event listener? 28

9 Hello World, event listener edition 30

10 Dead end 36

Contact Me

Writing Sublime Plugins is a work in progress. If you'd like to let me know about an error (I'm sure there are a few), or if you just have a question or want to offer some constructive feedback, email me at josh@sublimeproductivity.com.

Learn more about your favorite editor

Want a regular dose of Sublime Text tips to improve your productivity and streamline your workflow?

Head on over to http://sublimetexttips.com/newsletter[1] and drop your email address in the signup form.

[1]http://sublimetexttips.com/newsletter

Introduction

Who is this book for?

If you've been kicking around an idea for a Sublime Text plugin, but you're not sure how to get started, this book is for you. So if you're a Sublime plugin newbie looking to get started quickly, you've come to the right place.

On the other hand, if you're already an experienced plugin developer, you might not find much new information here. If you decide to read on, I'm hoping you'll at least glean a few bits of knowledge that will help you down the road.

What level of knowledge is assumed?

I'm aiming for an audience at a similar skill level to me. I have a several years of experience programming with C#, JavaScript and various scripting languages. Before I started working on this project, I hadn't written a Sublime plugin before. If you're a programmer with some solid experience with programming applications, you shouldn't have any trouble following along.

But if your programming experience is limited to basic JavaScript or jQuery, you might have a tough time keeping up.

Sublime plugins are written in Python, and familiarity with Python helps but is no means required. I hadn't worked with Python before tackling my first plugin, and I found it intuitive and easy to learn. It quickly became one of my favorite languages to work with.

What will this book cover?

This book is divided into four main sections.

In part one, we'll get a high-level overview of what a Sublime plugin is, how they differ from packages, how to install them, and the process for creating your own plugin.

Part two is a peek inside the Sublime plugin developer's toolbox. We'll look at what plugins are capable of and explore a few limitations. You'll learn what user interface components you have at your disposal, what you can do to interact with the user's machine, and see some of the hooks Sublime gives you to interact with the editing environment.

In part three, we get down to business and start writing plugins. After a quick "Hello, World" walkthrough, we'll build a plugin that illustrates many of the main concepts you need to know to write your own. We'll build the plugin step by step, and you'll see how each new concept fits into the bigger picture. You'll see how to troubleshoot common issues encountered with plugin development and how to debug your code. We'll also see how to give your plugin a finished, professional feel by supporting multiple versions of Sublime, providing keyboard shortcuts and integrating it into the Sublime application menu.

By the time we're finished, you'll understand the entire process of creating a plugin, from the first lines of code to publishing your plugin on Package Control.

This book isn't meant to serve as the central store of all plugin development knowledge. The Sublime API is too expansive and flexible to cover adequately in an introductory level book. Part four provides some help if you want to learn more, including an overview of what code is available in your Sublime installation for you to review and where to go for more information.

Which version of Sublime does this book target?

Deciding which version of Sublime to target was a significant decision. As of this writing, Sublime Text 3 is still in public beta. While many users have already upgraded, Sublime Text 2 is still the official released version.

Because of that, and because many plugins are yet to be upgraded to Sublime Text 2, and because most of the material and documentation available online was written for Sublime Text 2, I decided to start by developing the plugin for Sublime Text 2. This would simplify my learning curve, as I'd be less likely to get hung up trying to port a Sublime Text 2 example to Sublime Text 3. I'd also be less likely to encounter bugs or incomplete features that are just part of the Sublime Text 3 beta.

Then at the end, we'll "upgrade" our new plugin to support Sublime Text 3, which will make it easier as we're just concentrating on learning the differences. There were several changes to the Sublime APIs for version 3, but for plugin developers, the biggest consideration by far is that Sublime Text 3 upgrades Python from 2.6 to 3.3, which includes many breaking changes.

How can I follow along with the tutorials?

As we work through the process of building the plugin, we'll following the same steps that I took when writing the plugin myself.

In each section, I'll indicate code that we're adding by using boldface type:

```
def SayHello():
    print "Hello!"
```

And the code that you'll need to delete is highlighted using strike-through:

```
def SayHello():
~~    print "Hello!"~~
```

To save space, I’ve replaced the bodies of methods and classes that aren’t changing in a given section with ellipses:

```
def SayHello():
    ...
```

I’ll give instructions with the idea that you’re following along exactly. Of course, you’re free to name things differently than I do, or organize your files differently. But using the same steps I use will mean fewer mental gymnastics for you as we progress.

How will this tutorial be structured?

I’m going to walk you chronologically through the same steps and process of discovery that I took as I was writing the example. Along the way we’ll pause periodically to dig in to a topic in more depth. Then I’ll show you how the concept relates to the example we’re building. So expect some back and forth between step-by-step instructions and abstract theory.

Because a lot of the code we’ll be using in this plugin is standard Python code, I’m not going to go line by line through every code example. Instead, I’ll call out the most important elements in each example.

Ready to get started?

I What is a Sublime plugin?

1 What is the difference between a package and a plugin?

Before we take a look at what Sublime plugins can do, it's helpful to see how plugins relate to another Sublime concept: packages. The terms are often used interchangeably, and the distinction isn't immediately apparent.

A *package* is simply a collection of files that extends Sublime when properly installed. Packages may include editor skins and color themes, macros, code snippets, keyboard shortcuts, and programming language definitions to provide syntax highlighting. Popular third-party packages include the Flatland theme[1], which applies a flat UI look to Sublime's side bar and tabs, and the railsdev-sublime-snippets[2], which adds a collection of code snippets of use to Ruby on Rails developers.

Plugins are a specific category of Sublime package with several characteristics that set them apart from other packages. While regular packages generally consist of JSON and XML files, plugins are Python scripts. And while packages make use of predefined Sublime commands and events, plugins create new commands by extending one of the Sublime command classes or adding handlers for Sublime's built-in events.

Plugins are the most powerful of Sublime's extensibility mechanisms. They can combine existing commands and macros to provide new functionality, define new commands, manipulate elements of the Sublime environment such as tabs and text, and display user interface components to define new workflows.

One example that illustrates the power of plugins is SubliminalCollaborator[3], which allows two Sublime users to link their Sublime instances over IRC and pair program remotely.

[1]https://github.com/thinkpixellab/flatland

[2]https://github.com/j10io/railsdev-sublime-snippets

[3]https://github.com/nlloyd/SubliminalCollaborator

2 What can a Sublime plugin do?

Sublime gives you, the plugin creator, a lot of power through its APIs. Before we start trying to harness that power, it's useful to get an overview of what, exactly, you can expect to accomplish by creating a plugin, and what the limitations are.

Let's start by looking at what a Sublime plugin can do.

Reusing existing features and commands

Because the API Sublime exposes for plugins is so powerful, many of the "features" that ship with Sublime, which you may consider to be built-in functions of the editor, area actually just plugins written with the same APIs that you have access to. As a result, you can reuse many of these functions and commands in your own plugins, combining them in new ways to create brand new features. You can invoke commands defined by these default plugins from your own commands, combining them in new ways.

For example, Sublime's Goto Line feature is really a plugin defined in a file named `goto_line.py` in the `Default` package. This plugin defines the `GotoLineCommand`, which you could reuse if your plugin allowed users to navigate to a specific point in a file.

Importing Python modules

Since Sublime plugins are scripted in Python, you can import most modules that are included in the Python standard libraries. This puts the power of Python at your disposal and helps make your code portable across platforms.

And if there's an open source Python library you'd like to use, you can bundle it with your plugin and load it with your plugin, opening up a wide world of code reuse.

Defining your own commands

In addition to reusing Sublime's predefined commands, you can define your own commands and call them from within your plugin.

Your commands integrate alongside the default ones, and you can integrate your commands into Sublime's command history to support undo operations.

Processing text

Sublime provides a broad set of APIs for processing text. Most text processing operations fall into one of two categories. The first group of API calls allow you to query the text on the screen and return a result set, similar to the way jQuery allows you to interact with an HTML document.

The second set provides a set of actions that you use to manipulate a piece of text once you've selected it.

Querying text

Sublime provides several helper methods that allow you to select regions of text in the buffer by running queries and retrieving a result set.

For example, you can query by language type. Since Sublime applies syntax highlighting to code files, it has a limited understanding of the structure of a language. When you open a file containing multiple languages, Sublime parses the file and applies syntax highlighting selection scopes to various sections of the document. These scopes work like CSS selectors, allowing you to specify what colors and fonts to use for each one. Similar to how jQuery allows you to query by CSS selectors, you could query an HTML file using scopes to get a list of embedded JavaScript blocks.

When Sublime opens a code file for a supported language, it parses the file and compiles a list of *symbols*. If you're dealing with JavaScript, for example, the symbols list would include function definitions. You can get a list of these symbols easily.

Similarly, Sublime provides convenient access to its list of known word completions. When Sublime opens a file, it parses out a list of words to use as suggestions for its auto-complete feature. You can access this list.

Another way to select text is by specifying a set of coordinates that represent the starting and ending points of the selection.

Or you can select based on positional values such as the word's start or end, or the start or end points of the current line.

Manipulating text

Once you have a text selection or set of selections, you'll need to perform an editing operation on the text. Sublime includes methods that allow you to delete regions change the syntax highlighting applied to them. You can also split a region of text into individual lines or fold or unfold the region. Or you can choose to run another query to break the region up further.

Managing the editing session

Sublime provides a lot of access to the current editing session, including the windows and tabs that the user is using. These hooks allow you to learn about the environment your plugin is operating in and the state of the user interface.

Accessing environment variables

Sublime defines several environment variables that allow you to determine what version of Sublime is installed, where packages are installed, what operating system the user's computer is running, and what type of CPU is in the machine.

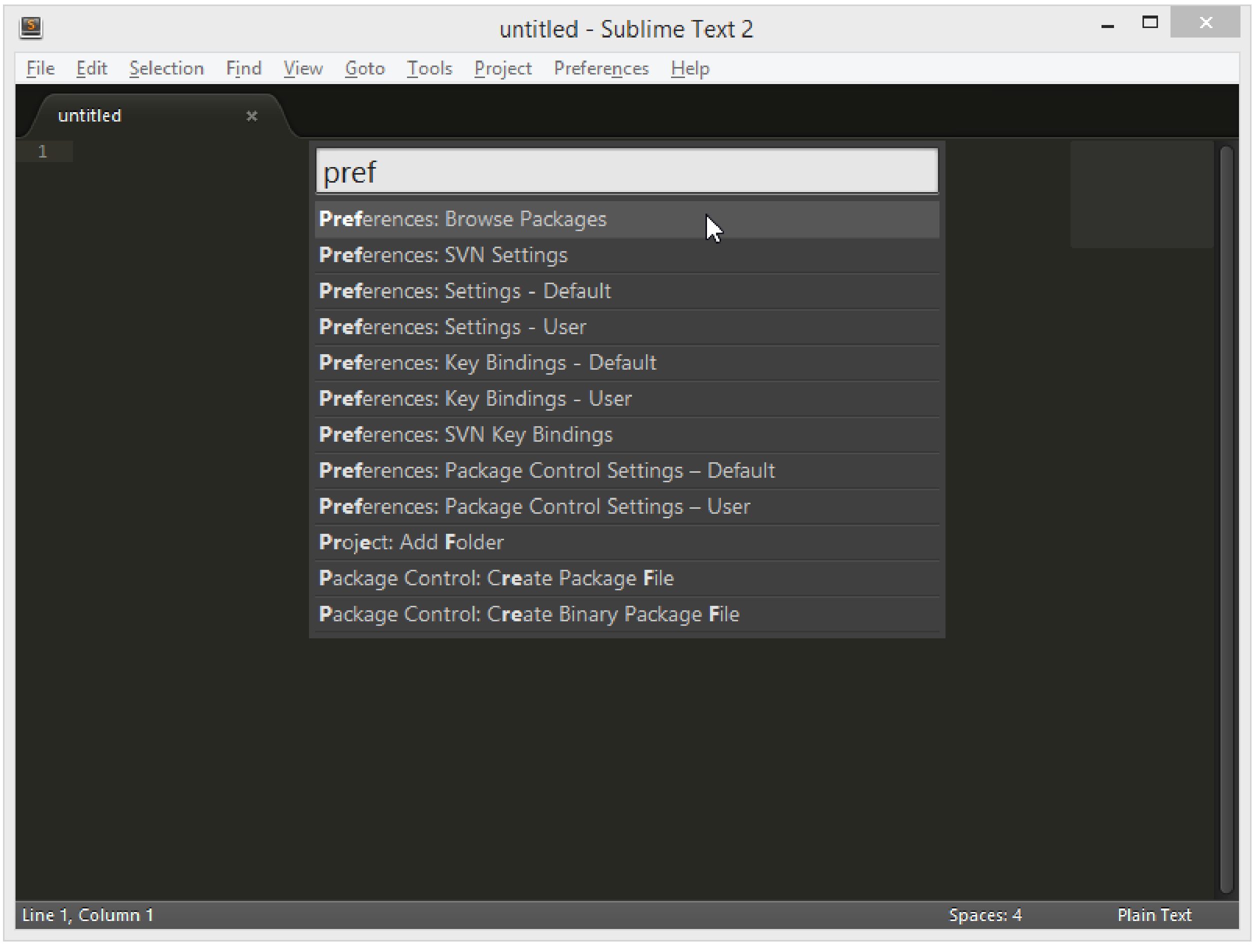
Checking and changing settings

You're able to check the current values of the settings defined in the default and user preferences files. You can also change them programmatically, although you'd need to exercise caution when doing so. You can also listen for changes to preferences and respond accordingly.

Creating user interface elements

Sublime provides API hooks that allow you to create and interact with most of the user interface components you're familiar with.

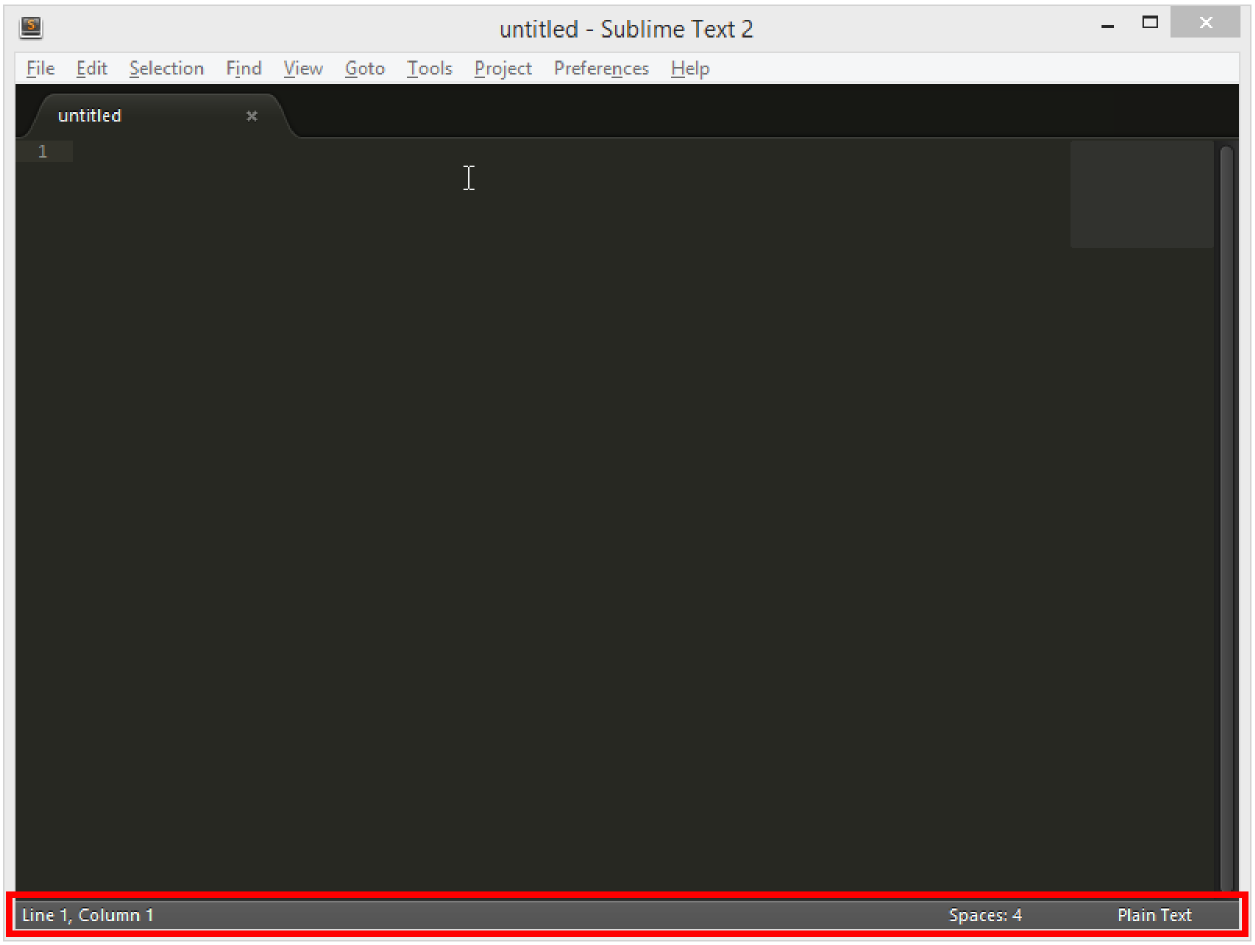
Quick panel


untitled - Sublime Text 2
File Edit Selection Find View Goto Tools Project Preferences Help
untitled
pref
Preferences: Browse Packages
Preferences: SVN Settings
Preferences: Settings - Default
Preferences: Settings - User
Preferences: Key Bindings - Default
Preferences: Key Bindings - User
Preferences: SVN Key Bindings
Preferences: Package Control Settings – Default
Preferences: Package Control Settings – User
Project: Add Folder
Package Control: Create Package File
Package Control: Create Binary Package File
Line 1, Column 1
Spaces: 4
Plain Text

The quick panel displays a filterable list of items.

You're familiar with the quick panel from using the command palette. It allows you to display a filterable list of items. The user can filter the items by typing. Fuzzy matching is built in.

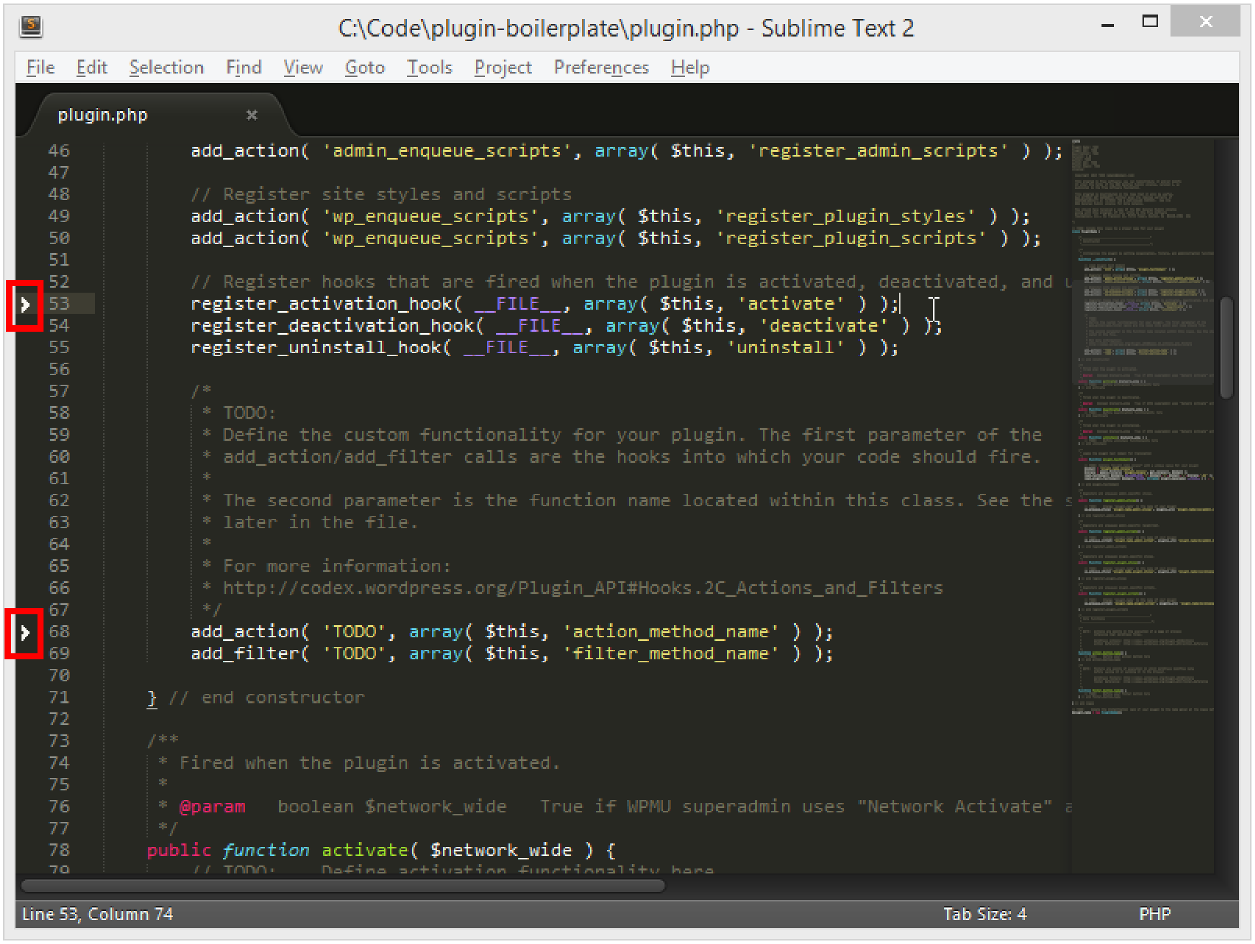
Status bar


untitled - Sublime Text 2
File Edit Selection Find View Goto Tools Project Preferences Help
untitled
Line 1, Column 1
Spaces: 4
Plain Text

The status bar displays short messages.

The status bar appears at the bottom of the screen, although the user can choose to hide it. You can use the status bar to display text messages to the user or for basic ASCII art style animations like a simple progress bar.

Gutter icons


C:\Code\plugin-boilerplate\plugin.php - Sublime Text 2
File Edit Selection Find View Goto Tools Project Preferences Help
plugin.php
Line 53, Column 74
Tab Size: 4
PHP

```
        add_action( 'admin_enqueue_scripts', array( $this, 'register_admin_scripts' ) );

        // Register site styles and scripts
        add_action( 'wp_enqueue_scripts', array( $this, 'register_plugin_styles' ) );
        add_action( 'wp_enqueue_scripts', array( $this, 'register_plugin_scripts' ) );

        // Register hooks that are fired when the plugin is activated, deactivated, and u
        register_activation_hook( __FILE__, array( $this, 'activate' ) );
        register_deactivation_hook( __FILE__, array( $this, 'deactivate' ) );
        register_uninstall_hook( __FILE__, array( $this, 'uninstall' ) );

        /*
         * TODO:
         * Define the custom functionality for your plugin. The first parameter of the
         * add_action/add_filter calls are the hooks into which your code should fire.
         *
         * The second parameter is the function name located within this class. See the s
         * later in the file.
         *
         * For more information:
         * http://codex.wordpress.org/Plugin_API#Hooks.2C_Actions_and_Filters
         */
        add_action( 'TODO', array( $this, 'action_method_name' ) );
        add_filter( 'TODO', array( $this, 'filter_method_name' ) );

    } // end constructor

    /**
     * Fired when the plugin is activated.
     *
     * @param    boolean $network_wide   True if WPMU superadmin uses "Network Activate" a
     */
    public function activate( $network_wide ) {
        // TODO:    Define activation functionality here
```

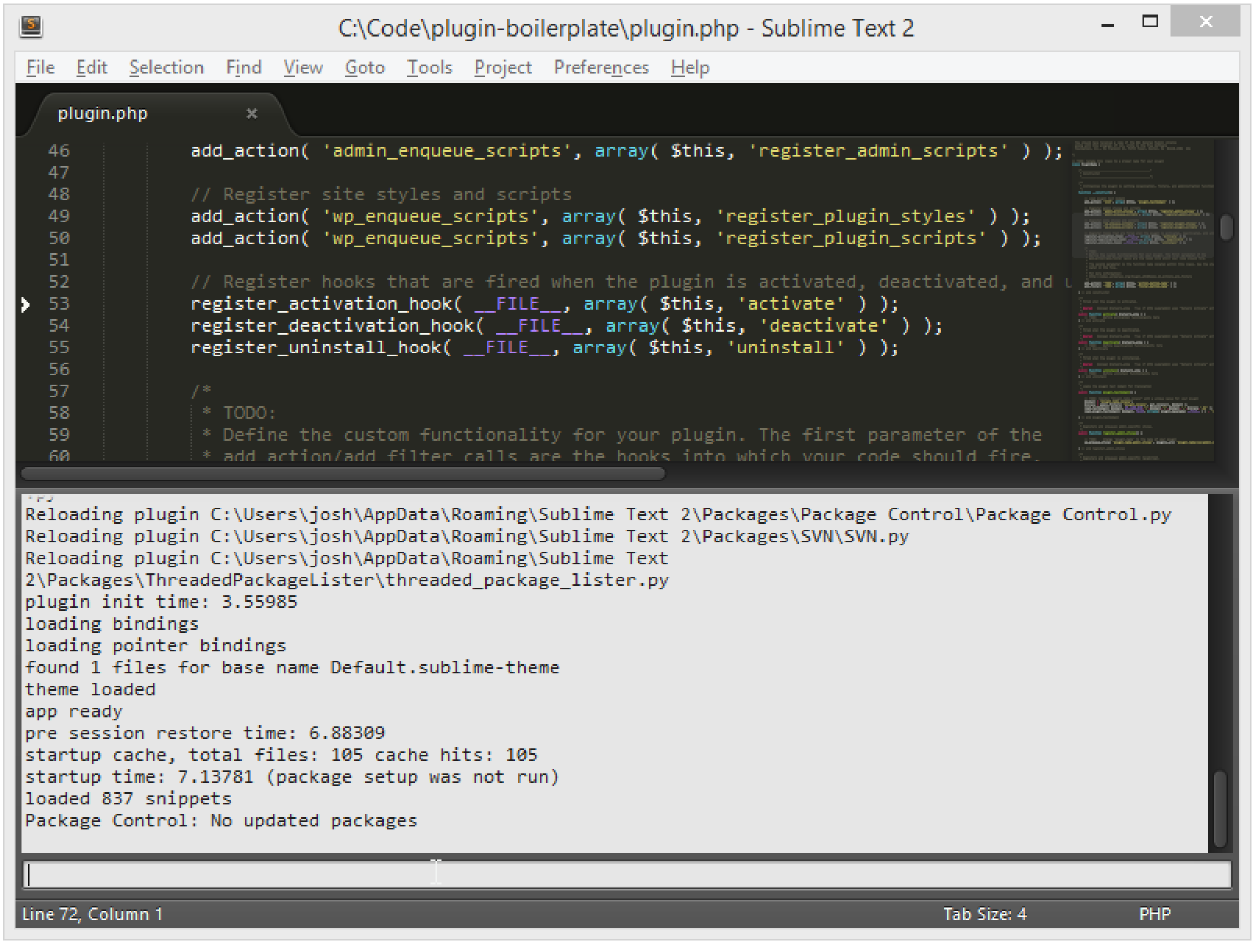
Gutter icons are useful line markers.

If you've used Sublime's bookmarks feature, you've seen gutter icons in action. You can programmatically set and remove icons, and you can use the default icons that ship with the theme or you can package your own in your plugin.

Dialogs

You can create dialog and confirmation boxes and display them to the user, as well as respond to the user's choice. Starting with Sublime Text 3, dialogs look like native operating system dialogs.

Console


C:\Code\plugin-boilerplate\plugin.php - Sublime Text 2
File Edit Selection Find View Goto Tools Project Preferences Help
plugin.php
Line 72, Column 1
Tab Size: 4
PHP

```
        add_action( 'admin_enqueue_scripts', array( $this, 'register_admin_scripts' ) );

        // Register site styles and scripts
        add_action( 'wp_enqueue_scripts', array( $this, 'register_plugin_styles' ) );
        add_action( 'wp_enqueue_scripts', array( $this, 'register_plugin_scripts' ) );

        // Register hooks that are fired when the plugin is activated, deactivated, and u
        register_activation_hook( __FILE__, array( $this, 'activate' ) );
        register_deactivation_hook( __FILE__, array( $this, 'deactivate' ) );
        register_uninstall_hook( __FILE__, array( $this, 'uninstall' ) );

        /*
         * TODO:
         * Define the custom functionality for your plugin. The first parameter of the
         * add_action/add_filter calls are the hooks into which your code should fire.
```

```
Reloading plugin C:\Users\josh\AppData\Roaming\Sublime Text 2\Packages\Package Control\Package Control.py
Reloading plugin C:\Users\josh\AppData\Roaming\Sublime Text 2\Packages\SVN\SVN.py
Reloading plugin C:\Users\josh\AppData\Roaming\Sublime Text
2\Packages\ThreadedPackageLister\threaded_package_lister.py
plugin init time: 3.55985
loading bindings
loading pointer bindings
found 1 files for base name Default.sublime-theme
theme loaded
app ready
pre session restore time: 6.88309
startup cache, total files: 105 cache hits: 105
startup time: 7.13781 (package setup was not run)
loaded 837 snippets
Package Control: No updated packages
```

Users can run plugin commands directly from the console.

The console is primarily useful during plugin development, when you can use it to try out snippets of Python code and see error messages.

But users can also use it to invoke your plugin and manipulate plugin settings directly. You can also use it to output debugging messages to help users troubleshoot issues they might encounter with your plugin.

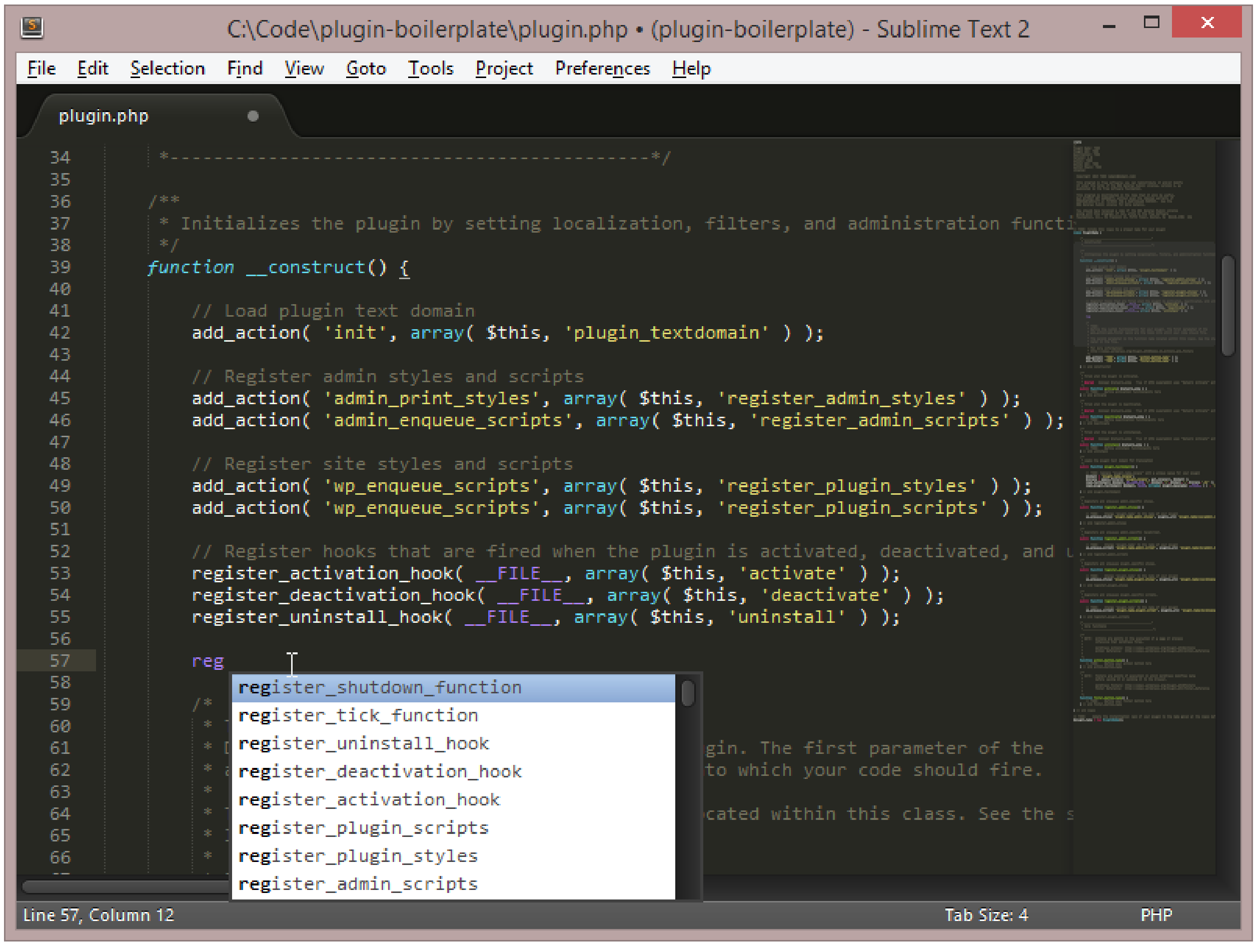
Menus

Most of Sublime's menus are open for extension. You can add entries that open files, run plugin-related commands, and even run external programs.

The menus you can modify include the main application menu and the context-specific menus that display when the user right-clicks a Sublime interface element such as files and folders in the side bar or tabs in the tab well.

Popout menu

The most familiar use of a pop-out menu is Sublime's completion feature:


C:\Code\plugin-boilerplate\plugin.php • (plugin-boilerplate) - Sublime Text 2
File Edit Selection Find View Goto Tools Project Preferences Help
plugin.php
Line 57, Column 12
Tab Size: 4
PHP

```
    *-----------------------------------------*/

    /**
     * Initializes the plugin by setting localization, filters, and administration functi
     */
    function __construct() {

        // Load plugin text domain
        add_action( 'init', array( $this, 'plugin_textdomain' ) );

        // Register admin styles and scripts
        add_action( 'admin_print_styles', array( $this, 'register_admin_styles' ) );
        add_action( 'admin_enqueue_scripts', array( $this, 'register_admin_scripts' ) );

        // Register site styles and scripts
        add_action( 'wp_enqueue_scripts', array( $this, 'register_plugin_styles' ) );
        add_action( 'wp_enqueue_scripts', array( $this, 'register_plugin_scripts' ) );

        // Register hooks that are fired when the plugin is activated, deactivated, and u
        register_activation_hook( __FILE__, array( $this, 'activate' ) );
        register_deactivation_hook( __FILE__, array( $this, 'deactivate' ) );
        register_uninstall_hook( __FILE__, array( $this, 'uninstall' ) );

        reg
            register_shutdown_function
            register_tick_function
            register_uninstall_hook
            register_deactivation_hook
            register_activation_hook
            register_plugin_scripts
            register_plugin_styles
            register_admin_scripts
```

Popout menus are used for autocomplete lists.

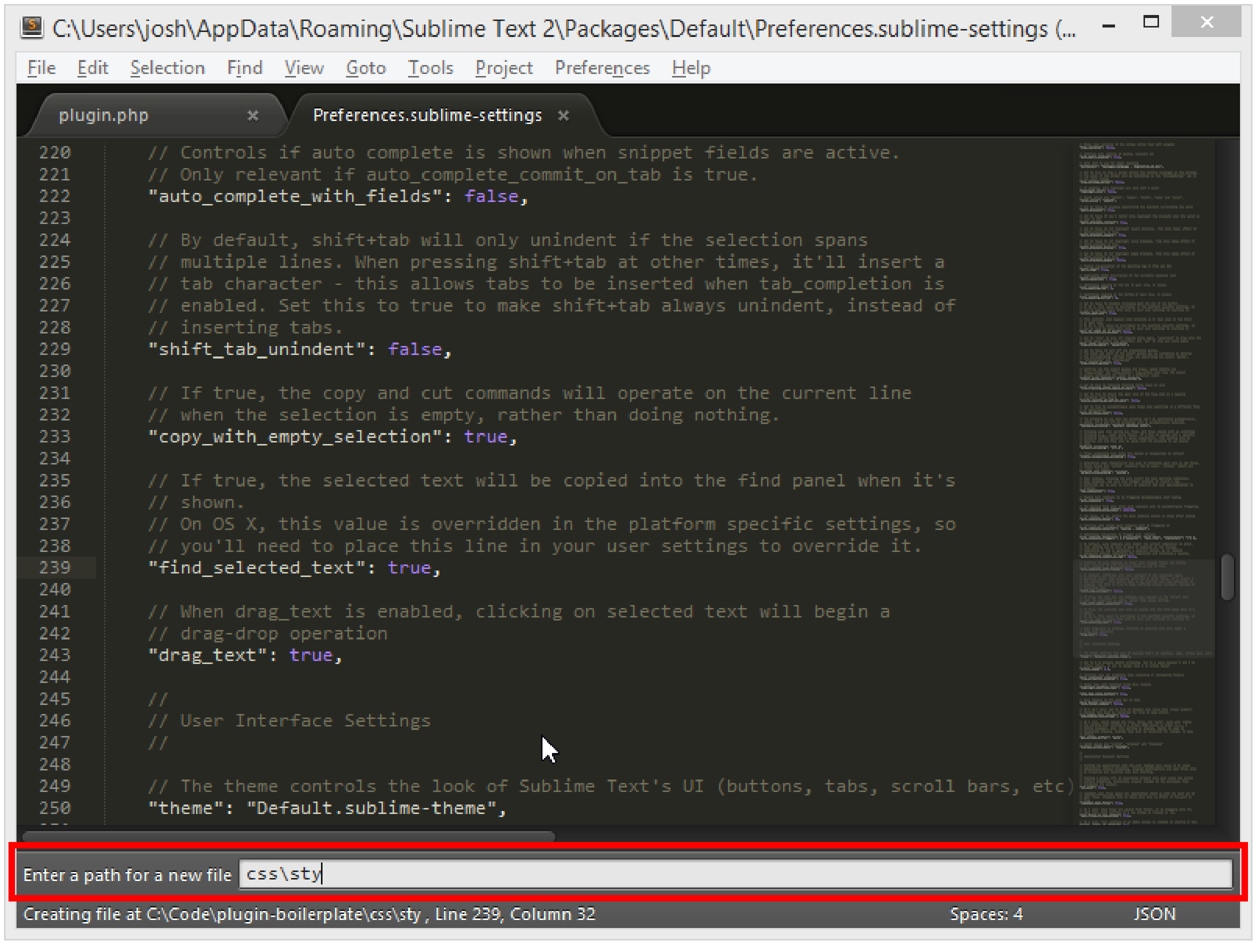
You can create pop-outs programmatically to support inline selection from a list of options.

Output panel

The output panel is a temporary buffer that allows you display output from a command. Sublime uses it to display the output from build commands. It appears at the bottom of the screen and it disappears when it loses focus.

Input panel

Plugins can prompt for user input using an input panel. An input panel appears at the bottom of the Sublime window and usually consists of a long text input field:


C:\Users\josh\AppData\Roaming\Sublime Text 2\Packages\Default\Preferences.sublime-settings (...
File Edit Selection Find View Goto Tools Project Preferences Help
plugin.php
Preferences.sublime-settings

```
    // Controls if auto complete is shown when snippet fields are active.
    // Only relevant if auto_complete_commit_on_tab is true.
    "auto_complete_with_fields": false,

    // By default, shift+tab will only unindent if the selection spans
    // multiple lines. When pressing shift+tab at other times, it'll insert a
    // tab character - this allows tabs to be inserted when tab_completion is
    // enabled. Set this to true to make shift+tab always unindent, instead of
    // inserting tabs.
    "shift_tab_unindent": false,

    // If true, the copy and cut commands will operate on the current line
    // when the selection is empty, rather than doing nothing.
    "copy_with_empty_selection": true,

    // If true, the selected text will be copied into the find panel when it's
    // shown.
    // On OS X, this value is overridden in the platform specific settings, so
    // you'll need to place this line in your user settings to override it.
    "find_selected_text": true,

    // When drag_text is enabled, clicking on selected text will begin a
    // drag-drop operation
    "drag_text": true,

    //
    // User Interface Settings
    //

    // The theme controls the look of Sublime Text's UI (buttons, tabs, scroll bars, etc)
    "theme": "Default.sublime-theme",
```

Enter a path for a new file: css\sty

Creating file at C:\Code\plugin-boilerplate\css\sty , Line 239, Column 32 Spaces: 4 JSON

The input panel allows a plugin to request text entry.

The AdvancedNewFile[1] plugin uses an input panel to allow the user to specify a file path and name for a new file, then creates the file and any parent folders that don't already exist.

Keyboard shortcuts

Good keyboard shortcuts are a critical user interface element, and Sublime allows plugins to define their own shortcuts. Shortcuts are created declaratively using JSON-formatted file.

User settings

If your plugin makes assumptions or choices for the user, you can allow the user to override your decisions by creating user settings and making your preferences the defaults. User preferences are created declaratively via a JSON-formatted file.

[1]https://github.com/skuroda/Sublime-AdvancedNewFile

The Git plugin[2] includes a setting that allows users to specify the path to their preferred Git executable.

Another use for user settings is to allow you to include additional the user to enable a mode in your plugin that you want to default to off. Many plugins include a debug setting that prints verbose output to the console.

Background threads

Long-running operations like calling web services or accessing the file system can cause Sublime's user interface to lock up until they complete, so Sublime allows plugins to shift these tasks to background threads. Using background threads allows you to display status updates to the user and allow the UI to continue to respond to user interactions.

Events

Sometimes you'd like your plugin to run automatically in the background, rather than requiring the user to invoke it explicitly. Sublime provides a number of event hooks that allow you to listen for important occurrences and act accordingly.

Sublime provides events that fire immediately before and after a file is saved, when text is modified, when text is selected, or when the window loses focus.

The SublimeOnSaveBuild[3] plugin uses an event handler approach to run the currently selected build task every time the user saves a file.

Windows

An instance of Sublime can have multiple windows, each with its own set of tabs or its own project open. Plugins are able to detect which Sublime window is active, access a list of all windows, and even close windows.

Projects

You can access the contents of the current project file, which is a JSON file. You can also make changes to the project file and save them.

This capability would enable you to create a plugin that automatically adds a custom build task to project files.

[2]https://github.com/kemayo/sublime-text-git
[3]https://github.com/alexnj/SublimeOnSaveBuild

Tabs

Sublime allows plugins to manage tabs, including open and close them, and moving them between panes, and changing the currently focused tab group. You're also able to check whether a tab contains unsaved changes.

The Origami plugin[4] uses the tab management API calls to allow users to easily navigate between panes and shuffle files around in a multi-pane layout.

Editing area

Sublime supports a coordinate system that allows plugins to determine the visible area of the screen. This system also allows you to scroll up and down to offscreen areas of a file or center a specific point on the screen.

Clipboard

You have ready access to read the contents of the clipboard, and you can write to the clipboard as well.

The Markdown2Clipboard plugin[5] uses this capability to let users convert Markdown to HTML with a simple copy and paste.

System resources

Plugins can access the host machine's resources by importing relevant Python modules. This provides a lot of power and flexibility.

The `os` module, for example, allows your code to work with the file system and makes it easy to create, open, read, save or delete files.

External programs

Sublime plugins can also invoke command line programs, which opens up a whole world of potential integrations with Ruby and its myriad gems, Node.js modules, shells like bash and PowerShell, and traditional command line tools such as git and rsync.

The JSHint plugin[6] uses Node.js to run the contents of the current file through the JSHint syntax checker, which you can easily install through the Node.js package management tool.

[4]https://github.com/SublimeText/Origami
[5]https://github.com/fanzheng/Sublime.Markdown2Clipboard
[6]https://github.com/uipoet/sublime-jshint

Plugins can also invoke GUI tools. The SideBarEnhancements[7] plugin adds a command to the side bar that allows the user to open a `.psd` file in PhotoShop just by right-clicking the file name.

And plugins can also call web services to perform tasks. Package Control is an excellent example of this. The Gist plugin[8] uses the GitHub APIs to allow users to create and edit gists directly in Sublime.

[7]https://github.com/titoBouzout/SideBarEnhancements
[8]https://github.com/condemil/Gist

3 What restrictions does Sublime place on plugins?

As we've seen, Sublime puts a huge selection of tools at the disposal of plugin authors. But all of this power comes at a price–there are some restrictions you'll quickly encounter when you set out to build your first plugin.

The most significant fact that plugins are essentially second class citizens. That's not to say that Sublime is overly restrictive of plugins. No, Sublime's creators have gone out of their way to expose as much of Sublime's power as possible. Plugins aren't crippled, but they are limited to what the Sublime API supports. Sublime's core is written in native C++ code and is thus inaccessible.

Within your own code you have near-total freedom and can do just about anything that Python will support. But when it comes to interacting with Sublime itself, you're limited to what the existing APIs will support.

This divide between native code and scripted plugins is frustrating at times in that it limits your ability to see and understand what Sublime is doing with your code. This can be especially irksome when you're troubleshooting a problem, such as trying to figure out why your event listener isn't firing, or why your plugin isn't getting called in the order you'd expect. Sublime either runs your code or it doesn't, and trying to figure out why can be exasperating.

We'll see an example of this in a bit when we delve into the example plugin. My initial idea for implementing the plugin was to intercept keyboard shortcuts when the user triggered them, then look up what commands Sublime planned to execute when the key binding was invoked. But I was fighting Sublime's API limitations every step of the way. I couldn't figure out how to intercept keyboard shortcuts. I had trouble figuring out why events weren't triggering as expected. And I couldn't access Sublime's list of bound commands, which it must maintain internally.

"Missing" APIs

In addition to the restrictions imposed by the fact that Sublime's core is native code, there are a number of API hooks that plugin developers have clamored for but just don't exist yet. These are probably not deliberate omissions, and they seem likely to appear in future versions of Sublime. The following is a sampling of the most frequently requested features from plugin developers–and the Sublime users who want plugins based on these features.

At the top of this wish lists is more API calls to manage the side bar and mini map. Plugin developers have asked for more access to theme the side bar, as well as apply custom icons to files and folders. Support for drag-and-drop file management is also missing.

Likewise, there's very little you can do with the mini map with it, and developers have asked for more control over its appearance.

In the editing window itself, there's no way to set custom background colors behind individual pieces of text. And Sublime doesn't support a concept of "tool tips," bits of text that appear when you hover over a portion of code, which are frequently used to add documentation and hints in other editors.

Many developers also want the ability to allow user input from the Sublime console. Users can run Python commands from the console, and developers can display program output there, but there's no way to run a plugin interactively via the console, accepting user input before continuing execution.

Plugin loading

The way Sublime loads plugins also imposes some restrictions.

For starters, the order in which your plugin is loaded is outside of your control. Sublime compiles a list of installed plugins and performs some sorting internally to determine when each plugin should load. This can cause cause unpredictable conflicts. If another plugin loads after yours and happens to remap one of your keyboard shortcuts, the other plugin wins.

Sublime Text 3 takes some steps to reduce the chance of conflicts by introducing function namespacing. In Sublime Text 2, you could easily declare global functions that had the potential for naming collisions.

Another restriction is that you shouldn't attempt to call the Sublime API while your plugin is loading. Most of the time, your plugin will be loading when the user is launching Sublime, and to keep the load times minimal you shouldn't try to make API calls. The application is also launching, so you can't depend on the application state.

There are a few exceptions to this: It's permissible to use a few API methods that allow you to check which version of Sublime is running, which allows you to support multiple versions.

Sublime Text 3 actively enforces this and ignores calls to the Sublime API that occur while the plugin is loading.

Command sandboxing

As we'll see shortly, *commands* are at the core of most Sublime plugins, and Sublime places several restrictions on commands that effectively sandbox them from each other.

While commands can invoke other commands, you're limited in what you can pass between commands. Only Sublime can pass live Python objects to a command. Any data that you need to send to another command must be in the form of a JSON-formatted string.

“Missing” Python modules

Experienced Python developers might be surprised to find that some of the modules in the Python standard library are absent in the version of Python that ships with Sublime. Among the most notable omissions are the `tkinter` GUI library, the `multiprocessing` threading library and the `sqlite3` database module.

Because of this, it pays to do some research and experimentation before you commit to an implementation path that leans heavily on Python standard library modules.

4 Hello World

Before we dive into writing a full-featured plugin, let's first look at a simple example so we can see the basic structure of a Sublime plugin. You know it's inevitable, so let's get right into Hello World.

What will it do?

There are several ways we could implement Hello World as a Sublime plugin, but for this example we'll stick with one that's built in to Sublime.

This plugin allows you to execute a command in the Sublime console to insert `Hello World` at the top of the current tab. If there are no open tabs, the plugin will create one, then insert the message.

Starting the plugin

Sublime ships with a command that allows you to easily generate the boilerplate code for a new plugin.

To get started, click **Tools | New Plugin** ....

Sublime creates a new, unsaved tab and inserts the following code:

```
import sublime, sublime_plugin

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

This is a fully functional plugin, but before we try it out, let's rename the command class to `HelloWorldCommand`:

```
import sublime, sublime_plugin

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

Now we're ready to take the plugin for a spin, but first we need to install it.

Installing the plugin

Installing the plugin involves saving the file in a subdirectory of your `Packages` directory, which is where Sublime expects to find plugins. This path differs by operating system and Sublime version, but it's usually located in your home directory.

You don't need to worry about the exact location just yet. Instead, click **File | Save** and Sublime opens the system file saving dialog with the `Packages` directory already selected.

Create a new folder under `Packages` called `HelloWorld`. Open your new folder and save your file as `hello_world.py`.

Because Sublime watches the `Packages` folder for changes, it will automatically load your new plugin as soon as you save the file.

Always create a new directory for plugins

Never save plugin files directly in the `Packages` folder or in `Packages/User`. Sublime sorts plugins to ensure that they load in the correct order, and it expects each plugin to live in its own subdirectory. Failing to put plugin scripts in a folder can cause them to load out of order, causing hard-to-troubleshoot bugs.

Understanding the code

This simple code snippet introduces some key concepts, so it's worth digging into it a bit.

First, we import the `sublime` and `sublime_plugins` modules:

```python
import sublime, sublime_plugin
```

These modules define the Python base classes that plugins derive from, as well as the methods that make up the Sublime API. (In Python, importing a module makes its functions available for use in the current file, similar to the way you'd link in a JavaScript library on a web page or use a `require` in Ruby.)

Next, we define the actual plugin:

```python
class HelloWorldCommand(sublime_plugin.TextCommand):
```

This line includes several elements worth pointing out. First, note that the plugin extends the class `sublime_plugin.TextCommand` by passing the name of the base class in the class definition. In Python, extending a class is similar to inheriting from a class in other languages. Sublime defines several base classes for plugins to use, and plugins that manipulate text generally extend `TextCommand`.

Next, take a look at how the plugin class is named. The name uses camel casing (first letter of each word capitalized) and ends in `Command`. When Sublime loads plugins, it examines all files with a `.py` extension, looking for classes that end in the word `Command` and that extend one of Sublime's `Command` classes. As it loads each plugin, it converts each command name by dropping the word `Command` from the end and converting the remainder from camel case to snake case (all lowercase, with underscores to separate words). In our example, `HelloWorldCommand` becomes `hello_world`.

In addition to extending a `Command` class and respecting Sublime's naming conventions, a plugin must also define a `run` method:

```
  def run(self, edit):
```

The `run` method is the jumping-off point for your code. Sublime calls this method when the command is invoked–after that, you're free to do just about anything, including calling other Python modules and classes and using the Sublime API to make your plugin go.

The `run` method takes two parameters, both Python objects, that are passed in by Sublime. In Python, the first parameter of any method is `self`, a reference to the class's current instance. The second parameter, `edit`, is an instance of the Edit class. This class allows you to bundle multiple text changes together so they can be rolled back as a group if necessary.

That's it for the boilerplate plugin code. The final line is where we get down to business and add our plugin's "features":

```
    self.view.insert(edit, 0, "Hello, World!")
```

To actually say "Hello, World!" we need to insert some text into a tab.

Since all `TextCommands` allow you to manipulate text in the editing window, each `TextCommand` has an attribute that points to the currently active tab, `self.view`. Here we're accessing the `view` to call its `insert` method, which takes three parameters: the `edit` object for the text command, an index number, and the string to insert. The index indicates what point to insert the string, and a value of `0` means to insert the text at the beginning of the document.

Executing the command

Enough with dissecting the code–let's run our new plugin.

At this point, the plugin is just a simple command with no user interface. But you can execute any Sublime command manually by invoking it from the console.

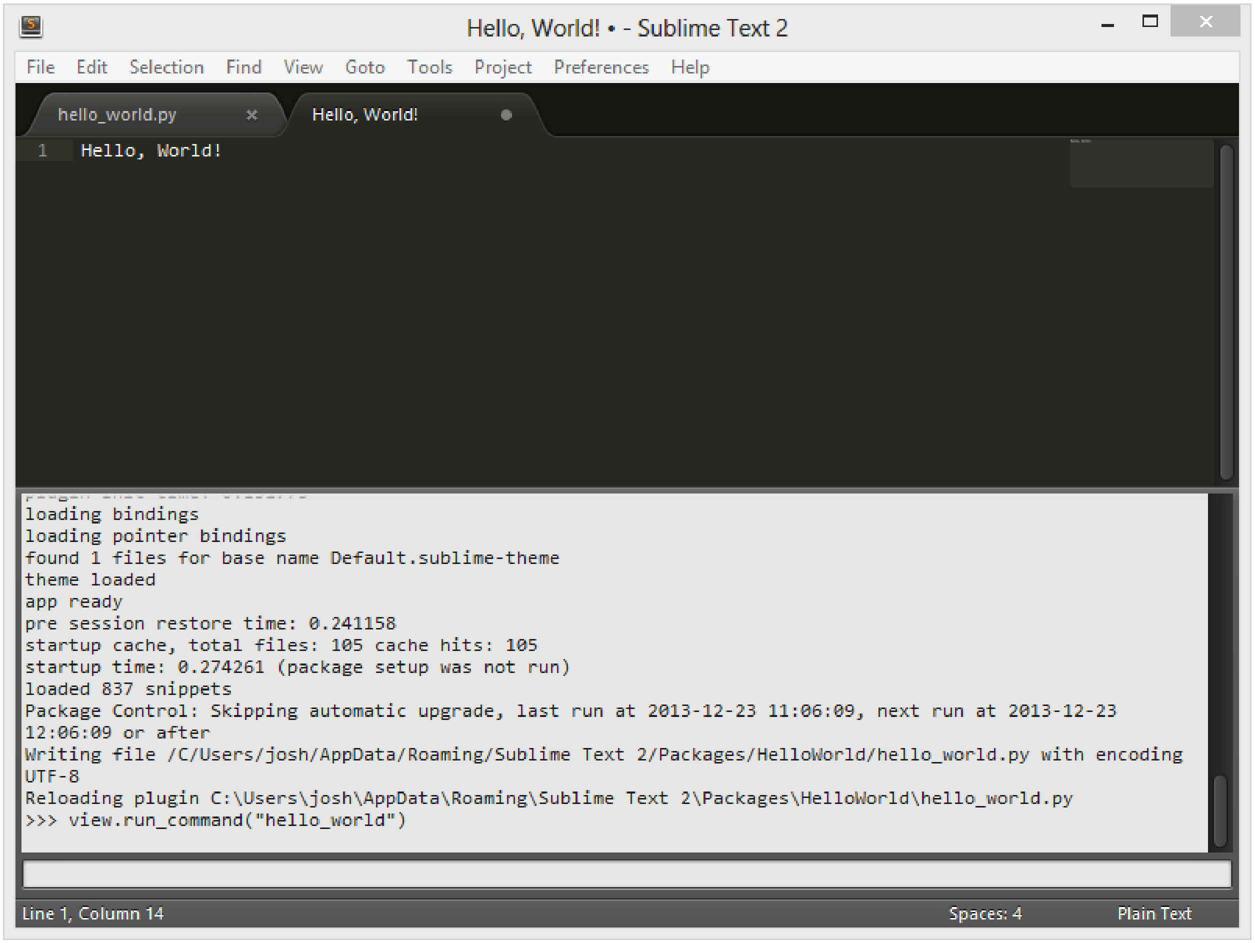
Before you run the command, though, open a fresh tab to give our `TextCommand` a clean `view` to work with.

Then click **View | Show Console** and enter the following:

```
view.run_command("hello_world")
```

We're again referencing the current `view` and calling the `run_command` method, passing the name of the command to execute as a string.

If the plugin is properly installed, you'll see our greeting at the top of your tab:


Hello, World! • - Sublime Text 2
File Edit Selection Find View Goto Tools Project Preferences Help
hello_world.py
Hello, World!
Line 1, Column 14
Spaces: 4
Plain Text

```
Hello, World!
```

```
loading bindings
loading pointer bindings
found 1 files for base name Default.sublime-theme
theme loaded
app ready
pre session restore time: 0.241158
startup cache, total files: 105 cache hits: 105
startup time: 0.274261 (package setup was not run)
loaded 837 snippets
Package Control: Skipping automatic upgrade, last run at 2013-12-23 11:06:09, next run at 2013-12-23
12:06:09 or after
Writing file /C/Users/josh/AppData/Roaming/Sublime Text 2/Packages/HelloWorld/hello_world.py with encoding
UTF-8
Reloading plugin C:\Users\josh\AppData\Roaming\Sublime Text 2\Packages\HelloWorld\hello_world.py
>>> view.run_command("hello_world")
```

The plugin inserts a greeting into a new tab.

Incidentally, a `TextCommand` always needs a `view` to work with, so if we'd called `run_command` when no tabs were open, Sublime will create a new tab.

II Event listeners

5 Introducing the KeyBindingInspector plugin

Now that you understand the basics of what makes up a Sublime plugin, it's time to dig in and start writing something useful.

The inspiration for the plugin that we're going to build came from a Twitter follower who wanted to know if there was a way to tell what any given keyboard shortcut was supposed to do. While it seems like an easy question to answer—just press it!—it's often not that simple. Many keyboard shortcuts only fire under certain conditions, or behave differently depending on the type of file you're editing. For example, the SmartMarkdown plugin[1] adds keyboard shortcuts that are only active when you're editing a Markdown file. When you're editing a blog post with SmartMarkdown installed, the Tab key will fold a section when the cursor is positioned on a headline; otherwise it just indents the text.

The KeyBindingInspector plugin is the result of my attempts to answer this question. I say "attempts"—as you'll see, it took me two tries to arrive at a workable solution. But I decided to include the false start as well because it illustrates how to listen for and handle events from the editor rather than responding to user instructions. It also exposes some limits in Sublime's APIs, and that knowledge might prove useful as you write your own plugin.

How will it work?

When I start building a new piece of software, I always begin by thinking about how I'd like the user to interact with what I'll be building.

Using that approach, I decided that the KeyBindingInspector plugin should respond whenever the user presses a keyboard shortcut by displaying the name of the command the shortcut was bound to.

On its face, it seemed like the most straightforward way to tackle the problem. I'd set up my plugin to eavesdrop on the user's keystrokes. Each time a keyboard shortcut was invoked, I'd intercept the shortcut, then check with Sublime to see what command this specific shortcut would invoke. I'd output the name of the command in the status bar, then invoke the original command, making my plugin transparent to the user. To avoid causing a performance hit, I'd require the user to enable my plugin before it would start monitoring keystrokes.

[1]https://github.com/demon386/SmartMarkdown

6 Setup

To kick things off, let's create, install and test a basic Hello World plugin. It's always a good idea to test that you've got the simplest possible case working before you get too far into a new project so you don't waste time later "debugging" code that's not working, only to find that you saved your plugin in the wrong folder or something.

Click **Tools | New Plugin**…. Sublime opens a new tab with the plugin boilerplate.

Click **File | Save** to open the file save dialog. Sublime preselects your `Packages` directory. Create a new folder named `KeyBindingInspector`, then save the plugin file as `KeyBindingListener.py`.

Pop open a new tab, then open the Sublime console by clicking **View | Show console**. Test your new plugin with the following command:

```
view.run_command("example")
```

Your plugin should display a cheerful greeting in the current tab.

Plugins and symlinking

Already I can see some of you thinking, "Wait, work directly in my `Packages` folder? No way! I'd rather put my code in a different folder, then just symlink the folder into the `Packages` directory.

That was my first thought as well, especially since I switch machines a lot. I figured I'd keep my in-progress plugin in Dropbox so I could work on it anywhere.

But Sublime didn't like my strategy. Sublime normally reloads a plugin when it detects that the plugin file has been changed, but the symlink confused it. Sometimes it would reload the file and other times it wouldn't.

I decided that the uncertainty wasn't worth the convenience, so I dropped the symlinking idea.

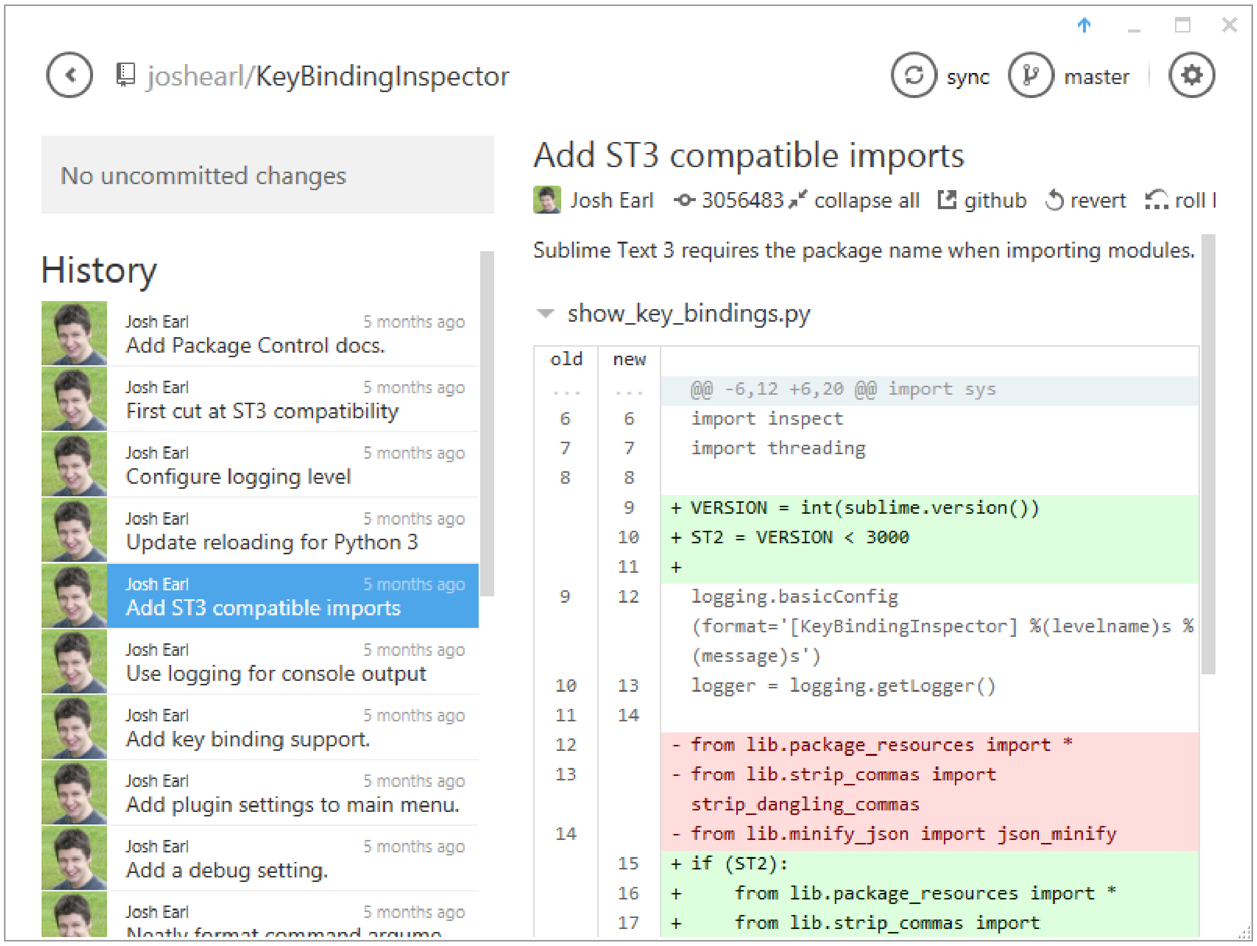
Creating the git repository

This step is optional but highly recommended. Using a version control tool like Git will free you up to experiment fearlessly with the code, and it'll probably spare you some debugging headaches at some point.

It certainly helped me on more than one occasion. When I was halfway through writing KeyBindingInspector, I thought I noticed that the plugin was taking *forever* to run. *What did I break?* I started to wonder.

Since I was using version control, I just reverted to an older version of the plugin and tested the performance. As it turned out, my computer was just running slowly that day–the old version took just as long to run as the latest code. I was able to set my performance concerns aside and get back to work.

A how-to on Git or another version control tool is out of scope for this book. If you're looking for an easy way to get started, check out GitHub for Mac[1] and GitHub for Windows[2]. They provide a nice graphical user interface that lets you perform basic Git commands and easily view your changes.



No uncommitted changes

History

Josh Earl 5 months ago
Add Package Control docs.

Josh Earl 5 months ago
First cut at ST3 compatibility

Josh Earl 5 months ago
Configure logging level

Josh Earl 5 months ago
Update reloading for Python 3

Josh Earl 5 months ago
Add ST3 compatible imports

Josh Earl 5 months ago
Use logging for console output

Josh Earl 5 months ago
Add key binding support.

Josh Earl 5 months ago
Add plugin settings to main menu.

Josh Earl 5 months ago
Add a debug setting.

Josh Earl 5 months ago

Add ST3 compatible imports

Josh Earl 3056483

Sublime Text 3 requires the package name when importing modules.

show_key_bindings.py

```
@@ -6,12 +6,20 @@ import sys
  import inspect
  import threading

+ VERSION = int(sublime.version())
+ ST2 = VERSION < 3000
+
  logging.basicConfig
  (format='[KeyBindingInspector] %(levelname)s %
  (message)s')
  logger = logging.getLogger()

- from lib.package_resources import *
- from lib.strip_commas import
  strip_dangling_commas
- from lib.minify_json import json_minify
+ if (ST2):
+     from lib.package_resources import *
+     from lib.strip_commas import
```

GitHub for Windows supports basic revision control commands.

And of course, there are several excellent Sublime plugins that allow you to interact with git without leaving Sublime. The Git plugin[3] is a favorite as of this writing.

[1]http://mac.github.com/
[2]http://windows.github.com/
[3]https://github.com/kemayo/sublime-text-git

Regardless of which tool you select, you'll need to initialize a new git repository inside the `KeyBindingInspector` folder, then add and commit your new plugin file.

7 Implementation overview

My initial idea for this plugin involved printing the command name to Sublime's status bar each time the user pressed a keyboard shortcut. Let's get a quick overview of what's required to make this approach work.

First, we need a way to react when a keyboard shortcut is triggered. Sublime allows plugins to respond to events by creating an *event listener*. To meet the plugin's requirements, we need to be notified every single time a keyboard shortcut is triggered, regardless of whether the command it's bound to will actually execute.

Once our event listener gets called, we need to somehow retrieve the name of the command that's going to execute in response. I was a little fuzzy about how this might be doable; I hoped that Sublime kept a global map somewhere of keyboard shortcuts and command names somewhere that I could access.

Another avenue I considered was piggybacking off of Sublime's command logging feature, which prints the name of each command to the console as it executes. Perhaps I could scrape the console history and get the name of the most recent command from that. This felt like a hack, though, and one that was likely to be fragile.

A slightly better option might be to grab the last entry from Sublime's command history and display that. That still seemed fragile, though, and I also didn't like that relying on history means my plugin is forever stuck in reactionary mode, responding to commands after they happened and trying to guess which one was triggered by the most recent key binding.

Once we get the command name, the rest is easy. We'll do some simple formatting to convert the command name from snake case to something more aesthetically pleasing, then insert a message into the status bar.

Finally, we'll add a setting to allow the user to toggle the plugin on and off so it wouldn't be a constant drag on Sublime's performance.

8 What is an event listener?

The core of our plugin is an *event listener*. Sublime supports two main plugin models. We've already seen a command in action and learned that a command is a Python class that extends one of Sublime's base `Command` classes and follows a set of naming conventions.

Similarly, event listeners are Python classes that extend the `sublime_plugin.EventListener` class. Unlike commands, the name of the event listener class doesn't matter–there's no need to end it with `EventListener`.

In addition to extending `EventListener`, an event listener also defines one or more methods that are named after the event the class wants to listen for, such as `on_new` or `on_close`.

How does an event-driven plugin work?

Just like with commands, Sublime looks for event listeners in the `Packages` directory. When it finds one, it loads it, and it inspects the event listener class for methods that match the names of supported events, like `on_modified`. When it finds a match, it registers that method in an internal list, along with any other plugins that indicate they want to be notified of the `on_modified` event.

Each time the user makes a change to a file, the `on_modified` event is triggered, and Sublime loops through its list of `on_modified` methods, executing each one as a callback. (You can see why the `on_modified` event can be a problem. If several plugins register for this event, Sublime executes each callback after every keystroke. Yikes.)

When a event callback is triggered, Sublime passes whatever parameters are defined for this event. You don't have control over these–you only get what Sublime supplies. When `on_new` is called, for example, you get a reference to the new `view` object, which represents the tab that was just created. For more details about the parameters for each event, check the resources in the Learning more section.

What events are available?

Sublime Text 2 defined 14 events, and Sublime Text 3 adds another dozen, for a total of more than two dozen events that plugins can listen for.

Some of the most useful events include `on_pre_save` and `on_post_save`, which are executed immediately before and after a file is saved, `on_query_completions`, which runs whenever Sublime's autocomplete feature is triggered, and `on_query_context`, which fires when Sublime checks context rules before executing a command.

Then there's the dangerous `on_modified`, which runs every time the user makes a change. That means every keystroke, potentially. You can quickly bog down Sublime by listening to this event.

Do events come with any gotchas?

While event listeners are quite useful, there are several potential pitfalls to watch out for when you're designing an event listener-based plugin.

For starters, remember that you may not be the only one listening for an event. Other plugins can register for the same event, and Sublime determines the order in which event callbacks are executed. You can't be sure whether your plugin will execute before or after others.

Related to this, you also can't prevent other plugins from responding to an event.

Let's consider a brief example to see how these restrictions could throw a monkey wrench into one possible implementation of the KeyBindingInspector plugin. The `on_text_command` event fires whenever a `TextCommand` executes, which would allowing our plugin to be notified whenever an editing-related command runs. We could then examine the command history to determine which command was just executed.

But it's possible that another plugin is also listening for `on_text_command` and responding to the event by triggering a command of its own. In this case, the most recent command in the history isn't the one that was triggered by the keyboard shortcut but the one that was triggered by a plugin that cut in line in front of us. And we can't prevent this from happening, since there's no way to make sure we're first in line.

Another caution associated with event listeners: Avoid performing slow or computationally expensive operations in them, such as loading files from the file system, calling web services or processing large blocks of text with regular expressions.

This is especially important with events that are triggered frequently, such as `on_modified` and `on_selection_modified`. Careless use of these events can cause the user interface to hang after every key press, making the editor unusable.

9 Hello World, event listener edition

With that background, let's start our venture into the world of event listeners by converting the Hello World plugin from a text command into an event listener.

We're going to throw these changes away shortly, so if you're using Git or another version control system, make sure you've committed the files we created in the last section.

First, rename the `ExampleCommand` class, and change its base class to `EventListener`:

```
import sublime, sublime_plugin

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

The actual name of an event listener class isn't significant, but it shouldn't end in `Command`.

Now delete the `run` method, and in its place define an `on_window_command` method:

```
import sublime, sublime_plugin

class KeyBindingListener(sublime_plugin.EventListener):

~~def run(self, edit):~~
~~    self.view.insert(edit, 0, "Hello, World!")~~
def on_window_command(self, window, name, args):
    print("The window command name is: " + name)
    print("The args are: " + str(args))
```

By defining the `on_window_command` method, we're notifying Sublime that we'd like to be notified whenever a `WindowCommand` executes. We'll cover `WindowCommand` in more detail later, but you can think of them as commands that don't operate on the contents of an existing tab. Examples of window commands include opening and closing layout panes and tab groups, switching tabs, and creating a new tab.

As a brief aside, you'll need to use Sublime Text 3 for this portion of the tutorial. I didn't realize it when I was working through this example originally, but `on_window_command` is new in Sublime Text 3. I'd intended to target Sublime Text 2 with the initial version of this plugin, but while I was working on this part I was still getting my bearings and didn't realize that I'd chosen an implementation that wouldn't be backward compatible.

The `on_window_command` method accepts four arguments. The first, `self`, is a reference to the current instance of `KeyBindingListener`. The `window` argument points to the active Sublime window. The last two are the most relevant for our purposes: `name` is a string containing the name of the command that is executing, and `args` is a list of arguments that the command will receive, which might include things like context rules that determine whether a command should fire.

The last two lines just print the name of the currently executing command to the Sublime console, as well as the list of the arguments that the command will receive when it executes.

Let's try out the new plugin. Since we're modifying a previously installed plugin, all that you need to do to activate the plugin is save the file. Sublime will load our new event listener automatically.

Open the console by clicking **View | Show console** or with the keyboard shortcut **Ctrl+`**. Showing the console is a window command, so it triggers our plugin, and when the console appears, you'll see a message similar to the following:

```
The window command name is: show_panel
The args are: {'panel': 'console', 'toggle': True}
```

To trigger another window command, click **View | Side Bar | Hide Side Bar**. The side bar closes, and you see the following output:

```
The window command name is: toggle_side_bar
The args are: None
```

Handling events

Now that we have a basic event handler working, we need to determine which events to handle in order to achieve our goals for this plugin. Remember that we want the plugin to execute every time the user presses a keyboard shortcut, then let the user know what command the shortcut is firing.

At first glance, `on_window_command` and its counterpart, `on_text_command`, seem perfect for our purposes. Let's add an event handler for `on_text_command` now and see if this pair of events will meet our requirements.

Below the definition for `on_window_command`, add a definition for `on_text_command`:

```
import sublime, sublime_plugin

class KeyBindingListener(sublime_plugin.EventListener):
    def on_window_command(self, window, name, args):
        print("The window command name is: " + name)
        print("The args are: " + str(args))

    def on_text_command(self, window, name, args):
        print("The text command name is: " + name)
        print("The args are: " + str(args))
```

Save the plugin to reload it, and we’re ready to test again.

First, try triple-clicking a line of text. This fires a text command that selects the entire line, and you should see a message in the console similar to this:

```
The text command name is: drag_select
The args are: {'event': {'x': 661.1171875, 'y': 181.07421875, 'button': 1}}
```

Now position the cursor somewhere in your plugin code and press **Ctrl+Space**, which triggers the autocomplete popup. You’ll see the following message:

```
The text command name is: auto_complete
The args are: None
```

Not bad. We’re triggering key bindings and getting information about the commands that Sublime is executing, and it seems like we’re well on our way.

There’s a catch, though. Try launching the command palette by pressing **Ctrl+Shift+P** on Windows and Linux or **Shift+Cmd+P** on Mac. The command palette appears, and in the console you’ll see … nothing. What black magic is this?

To get a better understanding of what is going on, let’s enable command logging. Run the following command in the Sublime console:

```
sublime.log_commands(True)
```

Now every time a command executes, it’s name is printed to the console. Open the command palette again, and this time you’ll see the command name and its arguments:

```
command: show_overlay {"overlay": "command_palette"}
```

As it turns out, some Sublime key bindings don't execute window or text commands. The `show_overlay` command is defined in Sublime's native core. While the keyboard shortcuts for showing the command palette are bound in the same `.sublime-keymap` files as other Sublime commands, the command itself isn't exposed to event listeners in the same way as commands that are defined in Python classes.

And this is just one example. There are other commands that are defined in the same way, and they're all beyond the reach of our plugin code. Our plan to use `on_window_command` and `on_text_command` seems to have hit a dead end.

And now that we see how these events operate, it's clear that there's another fatal flaw in this design. These events only fire when a *command* is triggered, but we want our plugin to fire whenever a *key binding* is pressed, regardless of whether it's already bound to a command yet.

Maybe `on_window_command` and `on_text_command` just aren't the best choices. What other events might be useful?

Most of Sublime's other events are tied to opening, closing and saving files or activating or deactivating the Sublime window, shifting to and from another application. But there are two events that seem like they could be useful: `on_modified` and `on_query_context`. Can we use these to supplement our existing events?

To see `on_modified` in action, add an `on_modified` handler:

```python
import sublime, sublime_plugin

class KeyBindingListener(sublime_plugin.EventListener):
    def on_window_command(self, window, name, args):
        print("The window command name is: " + name)
        print("The args are: " + str(args))

    def on_text_command(self, window, name, args):
        print("The text command name is: " + name)
        print("The args are: " + str(args))

    def on_modified(self, view):
        print("The on_modified event fired!")
```

Save the file. Now lets see when this event is triggered. In the plugin file, hit **Enter** twice to insert a couple of blank lines, then type `def` to trigger Sublime's autocompletion for the Python function snippet. Press **Tab** to insert the snippet. In the console, you'll see something like the following:

```
The on_modified event fired!
The on_modified event fired!
The text command name is: insert_best_completion
The args are: {'default': '\t', 'exact': False}
The on_modified event fired!
```

The `on_modified` event fires any time you make a change to a text buffer–adding or removing characters. It will fire when a command inserts text into the editing pane, such as a snippet or autocompletion. But it doesn't help us with the key binding plugin we're trying to write.

And what about `on_query_context`? To get a feel for how this event works, make sure you've deleted the code inserted by the `def` snippet, then add these lines to your plugin file:

```python
import sublime, sublime_plugin

class KeyBindingListener(sublime_plugin.EventListener):
    def on_window_command(self, window, name, args):
        print("The window command name is: " + name)
        print("The args are: " + str(args))

    def on_text_command(self, window, name, args):
        print("The text command name is: " + name)
        print("The args are: " + str(args))

    def on_modified(self, view):
        print("The on_modified event fired!")

    def on_query_context(self, view, key, operator, operand, match_all):
        print("The on_query_context event fired!")
```

Save the file. Now open a new tab, then use the command palette to execute the **Set Syntax: JavaScript** command. Type the following into your unsaved JavaScript file:

```
function () {
```

Check the console output, and you'll see something like this:

```
The on_modified event fired!
The on_query_context event fired!
The text command name is: insert_snippet
The args are: {'contents': '{$0}'}
The on_modified event fired!
```

The `on_query_context` event fired before the `insert_snippet` command executed. When you're editing a JavaScript file, typing a { character may seem like it's inserting an opening brace, but it's actually a keyboard shortcut that uses a snippet to insert the opening and closing curly braces, then position the cursor between them.

This key binding is defined in Sublime's default key binding file, and it looks like this:

```
// Auto-pair curly brackets
{ "keys": ["{"], "command": "insert_snippet",
    "args": {"contents": "{$0}"}, "context":
    [
        { "key": "setting.auto_match_enabled", "operator": "equal",
            "operand": true },
        { "key": "selection_empty", "operator": "equal",
            "operand": true, "match_all": true },
        { "key": "following_text", "operator": "regex_contains",
            "operand": "^(?:\t| |\\)|]|\\}|$)", "match_all": true }
    ]
},
```

The key here is the `context` array. Before the snippet executes, Sublime checks the surrounding text to make sure it's appropriate to insert a closing curly bracket. The rules specified in the key binding's `context` array determine whether the bound command should fire. These context rules give you fine-grained control over when a key binding should execute. For example, you could bind the same shortcut to insert different snippets based on the language of the file you're editing.

In this case, the context rules specify that the user's settings must allow auto closing of braces, the user must not have any text selected, and the text immediately following the cursor must match a specific set of characters.

If you look at the signature of `on_query_context`, you can see that each of the parameters matches a property in the objects in the `context` array.

That's all interesting, but does it help us with our key binding plugin? The `on_query_context` event clearly fires when keyboard shortcuts are triggered, so in that sense it seems like it might be useful. However, it's only triggered when a key binding includes a `context` element to restrict when it can operate. Many keyboard shortcuts don't define a context and thus won't trigger this event.

10 Dead end

It looks like we’re at an impasse in our attempt to create this plugin using event listeners. Sublime simply doesn’t expose the right hooks for us to respond to every single keyboard shortcut.

And even if we were able to capture every keyboard shortcut, there’s still the problem of mapping shortcuts to commands, and my research didn’t turn up any way to query Sublime to find out what command a shortcut will trigger.

Fortunately, there’s another promising approach we can try. It won’t work quite the same way, but it’ll allow the user to accomplish the same thing in the end.

Ready to switch gears?