SublimeRope Documentation

Release

Julian Eberius

January 22, 2017

Contents

1 Getting started **3**
- 1.1 SublimeRope . . . 3
  - 1.1.1 Basic Usage . . . 3
  - 1.1.2 Using Package Control . . . 3
  - 1.1.3 Available Commands . . . 3
- 1.2 Some notes . . . 4
  - 1.2.1 License . . . 4

2 Auto Completion **5**
- 2.1 Case-Insensitive Completions . . . 5
- 2.2 Manual Completions . . . 5
- 2.3 Simple Completion . . . 6
- 2.4 How to get autocompletion on dot typing? . . . 6
- 2.5 Some Considerations . . . 6

3 Cache Mechanisms in SublimeRope **7**
- 3.1 The Project Cache . . . 7
- 3.2 Adding modules to Autoimport Modules List . . . 7
- 3.3 The old way: editing Rope python path . . . 8

4 Auto Import **9**
- 4.1 Auto Import Improvements . . . 9
- 4.2 Some Considerations . . . 9

5 Jump to Global **11**
- 5.1 Some Considerations . . . 11

6 Show Documentation **13**
- 6.1 Some Considerations . . . 13

7 Refactoring **15**
- 7.1 Renaming . . . 15
- 7.2 Extract Method . . . 16
- 7.3 Extracting Decorated Methods . . . 16
- 7.4 Extract Variable . . . 16
- 7.5 Restructuring . . . 17

8 Key Bindings **19**

9 Indices and tables **21**

Contents:

CHAPTER 1

Getting started

Ready to get started? This page gives you a basic introduction to SublimeRope. If you're looking for detailed information about auto completions go to *Auto Completion* page. For information about refectoring go to *Refactoring* page.

1.1 SublimeRope

SublimeRope adds Python completions and some IDE-like functions to Sublime Text 2, through the use of the Rope library.

SublimeRope has been tested on the latest dev build working well on OSX, Linux and Windows.

1.1.1 Basic Usage

Just unzip / git clone the folder SublimeRope into ST2's Packages folder. Basic completion should work and all commands should be reachable through the Command Palette.

1.1.2 Using Package Control

The recommended way to install SublimeRope into ST2 is by using Package Control.

1.1.3 Available Commands

- Completions, which hook into Sublime's normal completion system (Ctrl+Space and as-you-type)
- Go to Definition
- Show Documentation
- Refactor->Rename
- Refactor->Extract Method
- Refactor->Extract Variable
- Refactor->Inline Variable
- Refactor->Restructure
- Jump to Global: Shows a list of project globals in a quickview and allows to jump to them.

- Import assist: Looks for possible imports from the project starting with the prefix under the cursor. Will automatically insert the "from X import Z" statement.
- Regenerate Global Module Cache
- Regenerate Project Cache

1.2 Some notes

SublimeRope is a Free Software project developed and maintained by Julian Eberius and a few collaborators in their free time. To review a full list of SublimeRope developers just look at the project's GitHub Page.

1.2.1 License

Sublime Rope is distributed under the Free Software Foundation General Public License terms:

```
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
```

Have a look at LICENSE.txt file for more information.

Note: This project uses code from other open source projects (Rope) which may include licenses of their own.

CHAPTER 2

Auto Completion

ST2 offers three types of completions itself:

- Word completions (complete from words present on the current buffer)
- Explicit completions (those that you can define in syntax-specific files, such as `Python.sublime.completions`)
- Autocompletions from plugins.

Note: Some people prefer to disable the default word and/or explicit completions that ST2 offers while using SublimeRope. This can be achieved by setting either of the following settings to `true` in the User Settings for SublimeRope settings in your ST2 Preferences Menú.

- `suppress_word_completions`
- `suppress_explicit_completions`

2.1 Case-Insensitive Completions

Some users prefer to get more completion proposals by ignoring case of what they type and what Rope has indexed. By setting `case_sensitive_completion` to false you can enable this behavior. For example, when typing "inde" SublimeRope will offer "IndexError".

2.2 Manual Completions

On larger projects, generating the completion proposals can take a noticeable amount of time. Waiting half a second is no problem usually, but with ST2 automatic as-you-type completions it can slow down your typing. Just set the "complete_as_you_type" option in your SublimeRope.sublime_settings or in your project settings as "rope_complete_as_you_type" to false, and you will only get the lightning fast buffer-based completions from ST2. Then you can manually trigger the more intelligent Rope-based completions using the "PythonManualCompletionRequestCommand". Bind it to some nice key, e.g. ctrl+alt+space, and you can get them easily when you need them.

2.3 Simple Completion

SublimeRope provides completion suggestions based on the rope library, but also offers a fall-back method if rope suggest nothing. Albeit it is sometimes useful, it is disabled by default because some users feels that it slows down the plugin too much.

You can enable it setting `use_simple_completion` to true in SublimeRope settings file.

2.4 How to get autocompletion on dot typing?

Some users want SublimeRope shown the completions just after typing the dot in for example `os.`, ST2 gives us a simple way to setup this desirable behavior.

We should just edit the `Python.sublime-settings` file in `Package/User` (or create it in case that you don't have one already). In this way, this behavior should be only applicable to our Python files and not in all the files we edit with ST2:

```
"auto_complete_triggers": [ {"selector": "source.python - string - comment - constant.numeroc", "cha
```

This line makes autocomplete not appear when we type dot inside strings or comments but fails with literal hexadecimal number (seems to be a bug in ST2 itself).

2.5 Some Considerations

The list of completions that you will get back when auto completion triggers is determined by the *Cache Mechanisms in SublimeRope*.

CHAPTER 3

Cache Mechanisms in SublimeRope

Rope library uses binary cache files to offer auto completion, import assist, jump to global and documentation features. Exists different ways of generate and/or regenerate those cache files using SublimeRope.

3.1 The Project Cache

When you create a new Rope project, you should use the SublimeRope command Regenerate Project Cache to get the autocompletion, jumps, documentation and auto imports working for the modules under your project root.

This action can be archieved using the ST2 Command Palette.

3.2 Adding modules to Autoimport Modules List

This is the recommended way of get global modules auto imports and friends working.

You can also get autocompletion and friends working for modules not under your project root simply adding them to the list of `autoimport_modules` in your `SublimeRope.sublime-settings` file and then run the `Regenerate Global Module Cache` command from ST2 Command Palette or generating a key binding for it.

Note: Take care, there is also another command, `Regenerate Project Cache`, which rebuilds the cache just for your project not for global modules.

You can specify `rope_autoimport_modules` in your project settings instead of your global `SublimeRope.sublime-settings` file.

Note: note that although using the global configuration file works, using your project settings file is the recommended way

To archieve this you should navigate to `Project->Edit Project` and add your modules there in the `settings` section. If this section doesn't exists just create it:

```
"settings":
{
    "rope_autoimport_modules":
    [
        "twisted.*",
        "numpy.*",
        "libsaas.*",
        ...
    ]
}
```

Warning: Non-trivial/nested modules like numpy or twisted will have to be added in the form `twisted.*` or Rope will not index them correctly.

3.3 The old way: editing Rope python path

Warning: This is *NOT* the recommended way of settings cache to get all auto completions working anymore.

Basically, anything you want completions for (or auto import, jump to globals, documentation) has to be in the Rope's python path. You can extend the Rope's python path editing the `.ropeproject/config.py` (If you don't know what `.ropeproject` directory is you should look at create_project)

If you are using virtualenv for your project, add the path to the `virtualenv` in `.ropeproject/config.py` if you don't set it at project creation (there should be a commented-out already line in the `set_prefs` configuration key in the file):

```
prefs.add('python_path', '/Users/<username>/Development/Twisted-12.0/')
```

Also, if you are not using the same Python as ST2 (e.g. using a custom Python os OSX or using the latest Ubuntu release), add your site-packages (dist-packages in some Linux distributions), so that ST2 picks them up:

```
prefs.add('python_path', '/usr/lib/python2.7/site-packages')
```

A special case are Django projects, which use global imports and not relative ones, e.g., in views.py they use "import Project.App.models" and not just "import models". In this case, you project has to be on the Python path as well. Add the parent dir of your project:

```
prefs.add('python_path', '/Users/<username>/my_django_projects')
```

CHAPTER 4

Auto Import

SublimeRope offers auto importing features through Rope library. It looks for possible imports from the project starting with the prefix under the cursor.

In order to use auto import you should look for `Rope: Import Assist` in the Control Palette or just add a shortcut in your configuration. We provide a serie of possible *Key Bindings* in this Documentation as well.

4.1 Auto Import Improvements

When Auto Import Improvements are enabled all the undefined words in the current file should be checked (in a different thread) looking for an auto import for them. If SublimeRope finds a possible autoimport for them should show you so you can choose it and the import will be automatically applied to your file.

You can disable Auto Import Improvements in your SublimeRope's settings file just setting the `use_autoimport_improvements` to false if you feels that its slowing down your ST2.

4.2 Some Considerations

The list of modules that you will get when doing auto import is determined by the *Cache Mechanisms in SublimeRope*.

CHAPTER 5

Jump to Global

To use the jump to global feature in SublimeRope just select the `Rope:  Jump to Global` entry in the ST2's Control Palette, a list of all the possible globals that you can jump on will be shown.

5.1 Some Considerations

The available global jumps positions that will be accesible are determined by the *Cache Mechanisms in SublimeRope*

CHAPTER 6

Show Documentation

To use the show documentation feature in SublimeRope just write the name of the function you are going to use and select the `Rope:  Show Documentation` from the ST2's Control Palette. We suggest you use a key binding for this action.

6.1 Some Considerations

The available documentation that will be accesible are determined by the *Cache Mechanisms in SublimeRope*.

CHAPTER 7

Refactoring

SublimeRope offers a few refactors through Rope library.

7.1 Renaming

Consider the following code:

```
class First(object):
    """First Class"""
    def __init__(self):
        self.id = 1

    def change_id(self, arg):
        self.id = id

    def print_id(self):
        print self.id

first = First()
first.change_id(123)
first.print_id()
```

We can just put the cursor over id in the `__init__` method and rename it to `new_id` after that our code should looks like:

```
class First(object):
    """First Class"""
    def __init__(self):
        self.new_id = 1

    def change_id(self, arg):
        self.new_id = new_id

    def print_id(self):
        print self.new_id

first = First()
first.change_id(123)
first.print_id()
```

After renaming `print_id` method to `identify_me` all the references in our project should change to our new function name.

Note: ST2 already offer some interesting features to archieve the same results really fast

7.2 Extract Method

Let's imagine we have the following code:

```
def some_func():
    a = 10
    b = 20
    c = ``a * 2 + b * 3``
```

After performing extract method of the highlighted operation above we should get:

```
def some_func():
    a = 10
    b = 20
    c = new_func(a, b)

def new_func(a, b):
    return a * 2 + b * 3
```

7.3 Extracting Decorated Methods

Extract method can handle static and class methods that use the decorators `@staticmethod` and `@classmethod` as for example in:

```
class First(object):

    @staticmethod
    def a_method(arg):
        aux = arg * 2
```

After extract a * 2 as a method called `twice` we should get:

```
class First(object):

    @staticmethod
    def a_method(arg):
        aux = First.twice(arg)

    @staticmethod
    def twice(arg):
        return arg * 2
```

7.4 Extract Variable

Imagine we have this expression:

```
x = 2 * 3
```

After extract a variable with name `six` we should have:

```
six = 2 * 3
x = six
```

7.5 Restructuring

A restructuring is a program transformation; not as well defined as other refactorings like rename. In its basic form, we have a `pattern` and a `goal`. Consider we were not aware of the `**` operator and wrote our own:

```
def pow(x, y):
    result = 1
    for i in range(y);
        result *= x
    return result

print pow(2, 3)
```

When we realice that `**` exists we want to use it wherever `pow` is used. We can use a pattern like:

```
pattern: pow(${param1} ** ${param2})
```

Goal can be something like:

```
goal: ${param1} ** ${param2}
```

The matched names in pattern should be replaced with the string that was matched in each occurrence. So the outcome of the restructuring should be:

```
def pow(x, y):
    result = 1
    for i in range(y):
        result *= x
    return result

print 2 ** 3
```

It seems to be working but what if pow is imported in some module or we have some other function defined in some other module that uses the same name and we don't want to change it. Wildcard arguments come to rescue. Wildcard arguments is a mapping; Its keys are wildcard names that appear in the pattern (the names inside `${...}`).

The values are the parameters that are passed to wildcard matchers. The arguments a wildcard takes is based on its type.

For checking the type of a wildcard, we can pass `type=value` as an argument; value should be resolved to a python variable (or reference). For instance for specifying `pow` in this example we can use `mod.pow`. As you see, this string should start from module name. For referencing python builtin types and functions you can use `__builtin__` module (for instance `__builtin__.int`).

For solving the mentioned problem, we change our pattern. But goal remains the same:

```
pattern: ${pow_func}(${param1}, ${param2})
goal: ${param1} ** ${param2}
```

Consider the name of the module containing our `pow` function is `mod`. args can be:

```
pow_func: name=mod.pow
```

If we need to pass more arguments to a wildcard matcher we can use , to separate them. Such as name: `type=mod.MyClass,exact`.

This restructuring handles aliases; like in:

```
mypow = pow
result = mypow(2, 3)
Transforms into:

mypow = pow
result = 2 ** 3
```

If we want to ignore aliases we can pass exact as another wildcard argument:

```
pattern: ${pow}(${param1}, ${param2})
goal: ${param1} ** ${param2}
args: pow: name=mod.pow, exact
```

`${name}`, by default, matches every expression at that point; if exact argument is passed to a wildcard only the specified name will match (for instance, if exact is specified , `${name}` matches name and x.name but not var nor (1 + 2) while a normal `${name}` can match all of them).

CHAPTER 8

Key Bindings

SublimeRope provides no default keybindings at the moment, so you need to set them yourself. We suggest this bindings as a template that you can modify at your convenience:

```
{ "keys": ["ctrl+r", "ctrl+d"], "command": "goto_python_definition", "context":
    [
        { "key": "selector", "operator": "equal", "operand": "source.python" }
    ]
},
{ "keys": ["ctrl+r", "ctrl+h"], "command": "python_get_documentation", "context":
    [
        { "key": "selector", "operator": "equal", "operand": "source.python" }
    ]
},
{ "keys": ["ctrl+r", "ctrl+r"], "command": "python_refactor_rename", "context":
    [
        { "key": "selector", "operator": "equal", "operand": "source.python" }
    ]
},
{ "keys": ["ctrl+r", "ctrl+e"], "command": "python_refactor_extract_method", "context":
    [
        { "key": "selector", "operator": "equal", "operand": "source.python" }
    ]
},
{ "keys": ["ctrl+r", "ctrl+v"], "command": "python_refactor_extract_variable", "context":
    [
        { "key": "selector", "operator": "equal", "operand": "source.python" }
    ]
},
{ "keys": ["ctrl+r", "ctrl+n"], "command": "python_refactor_inline_variable", "context":
    [
        { "key": "selector", "operator": "equal", "operand": "source.python" }
    ]
},
{ "keys": ["ctrl+r", "ctrl+j"], "command": "python_jump_to_global", "context":
    [
        { "key": "selector", "operator": "equal", "operand": "source.python" }
    ]
},
{ "keys": ["ctrl+r", "ctrl+i"], "command": "python_auto_import", "context":
    [
        { "key": "selector", "operator": "equal", "operand": "source.python" }
    ]
},
{ "keys": ["ctrl+r", "ctrl+c"], "command": "python_regenerate_cache", "context":
    [
        { "key": "selector", "operator": "equal", "operand": "source.python" }
    ]
},
{ "keys": ["ctrl+r", "ctrl+f"], "command": "python_refactor_restructure", "context":
    [
        { "key": "selector", "operator": "equal", "operand": "source.python"}
    ]
},
{ "keys": ["ctrl+r", "ctrl+m"], "command": "python_generate_modules_cache", "context":
    [
        { "key": "selector", "operator": "equal", "operand": "source.python"}
    ]
},
{ "keys": ["ctrl+alt+space"], "command": "python_manual_completion_request", "context":
    [
        { "key": "selector", "operator": "equal", "operand": "source.python" }
    ]
},
```

CHAPTER 9

Indices and tables

- modindex
- search