Vimrunner Documentation

Release 1.0.1

Andrei Chiver

Apr 01, 2017

Contents

Module that implements a client and server interface useful for controlling a vim server. This module could be used for unit testing or integration testing for a Vim plugin written in Python. Or you can use it to interactively control a Vim editor by Python code, for example, in an Ipython session.

This work tries to be the python equivalent of Vimrunner ruby gem found at: http://rubydoc.info/gems/vimrunner/index

I thank the author(s) for the effort and nice level of abstraction they put in this gem.

class `vimrunner.vimrunner.`**`Client`**(*server*)

Client that has a reference to a Vim server. Useful to send keys, commands, expressions to manipulate Vim.

`add_plugin`(*dir*, *entry_script=''*)

Adds a plugin to Vim's runtime. Initially, Vim is started without sourcing any plugins to ensure a clean state. This method can be used to populate the instance's environment.

dir - The base directory of the plugin, the one that contains

its autoload, plugin, ftplugin, etc. directories.

entry_script - The Vim script that's runtime'd to initialize the plugin (optional)

Examples:

```
>>> client.add_plugin('/home/andrei/.vim/my_plugin/', 'plugin/rails.vim')
```

Returns nothing.

`append_runtimepath`(*dir*)

Appends a directory to Vim's runtimepath.

dir - The directory added to the path

Returns nothing. Eg:

```
>>> client.append_runtimepath("/path/to/a/plugin/dir")
```

`command`(*cmd*)

Send commands to a Vim server. Used for Vim cmds and everything except for calling functions. Eg:

```
>>> client.command("ls")
```

`echo`(*expression*)

Echo the expression in Vim. Eg:

```
>>> # get list of directories where plugins reside
>>> client.echo("&runtimepath")
>>> # output color brightness
>>> client.echo("&bg")
>>> # echo a string in Vim
>>> client.echo('"testing echo function with a string"')
>>> # or double quotes need to be escaped
>>> client.echo(""testing echo function with a string"")
```

Returns the String output.

`edit`(*filename*)

Edits the file filename with Vim.

Note that this doesn't use the '–remote' Vim flag, it simply types in the command manually. This is necessary to avoid the Vim instance getting focus.

filename - a String that can be a relative or absolute path

Returns, if the file is found, a string with the name of the document otherwise it returns an empty string. Eg:

```
>>> # suppose 'test' folder is in pwd:
>>> result = client.edit('test/a-file.txt')
>>> result
'"test/a-file.txt" 10L, 304C'
```

```
>>> # otherwise an absolute path is needed:
>>> client.edit('/home/user/path_to_file/file.txt')
```

eval(*expression*)
Calls the server's remote_expr() method to evaluate the expression.

Returns the String output of the expression, stripped by useless whitespaces. Eg:

```
>>> # get the line number of the cursor
>>> client.eval('line(".")')
```

Note that Vim makes a clear distinction between ‘ and ”.

feedkeys(*keys*)
Send keys as if they come from a mapping or typed by a user. Vim's usual remote-send functionality to send keys to a server does not respect mappings. As a workaround, the feedkeys() function can be used to more closely simulate user input.

Example: We want to send 3 keys: Ctrl w p and according to Vim docs you would write: ‘<C-w>p’ but these keys need to be escaped with a backslash ‘’:

```
>>> # in Vim you would write
>>> :call feedkeys("\<C-w>p")
>>> # this function can be used like this:
>>> client = Client(server)
>>> client.feedkeys('\<C-w>p')
>>> client.feedkeys('\<C-w>k')
```

get_active_buffer()
Get the current (active) vim buffer. Returns a string with the buffer number.

insert(*text*)
Switches Vim to insert mode and types in the given text at current cursor position. Eg:

```
>>> client.insert('Hello World!')
```

normal(*keys=’‘*)
Switches Vim to normal mode and types in the given keys.

prepend_runtimepath(*dir*)
Prepends a directory to Vim's runtimepath. Use this instead of append_runtimepath() to give the directory higher priority when Vim runtime's a file.

dir - The directory added to the path

Eg:

```
>>> client.prepend_runtimepath('/home/user/plugin_dir')
```

quit()
Exit Vim.

read_buffer(*lnum*, *end='*', *buf=None*)
Reads lines from buffer with index 'buf' or, by default, from the current buffer in the range lnum -> end. Uses vim's getbufline().

Returns one string with the lines joined with newlines '\n' marking the end of each line. Eg:

```
>>> one_line = client.read_buffer("1")
>>> two_lines = client.read_buffer("1", "2")
>>> all_lines = client.read_buffer("1", "$")
>>> two_lines = client.read_buffer("line('$') - 1", "'$'")
```

search(*text*, *flags='*, *stopline='*, *timeout='*)
Starts a search in Vim for the given text. The result is that the cursor is positioned on its first occurrence. For info about the rest of the args, check :help search.

source(*script*)
Source a script in Vim server.

script - a filename with an absolute path.

You can see all sourced scripts with command('script')

type(*keys*)
Invokes one of the basic actions the Vim server supports, sending a key sequence. The keys are sent as-is, so it'd probably be better to use the wrapper methods, normal(), insert() and so on. Eg:

```
>>> client.type(':ls <Enter>')
```

write_buffer(*lnum*, *text*)
Writes one or more lines to current buffer, starting from line 'lnum'. Calls vim's setline() function.

lnum - can be a number or a special character like $, '.'. etc.

text - can be a string or a list of strings.

Returns '0' or '1', as strings.

Eg:

Input is a string

```
>>> client.write_buffer("2", "write to line number 2")
>>> client.write_buffer("'$'", "write to last line")
>>> client.write_buffer(""$"", "write to last line")
>>> client.write_buffer("'$'", "['last line', 'add after last line']")
>>> client.write_buffer("line('$') + 1", "add after last line")
```

Input is a list

```
>>> l = ['last line', 'add after last line']
>>> client.write_buffer("'$'",l)
```

Pay attention, simple and double quotes matter.

class vimrunner.vimrunner.**Server**(*name='*, *executable='vim'*, *vimrc='*, *noplugin=True*, *extra_args=['-n']*)
Represents a remote Vim editor server. A Server has the responsibility of starting a Vim process and communicating with it through the client - server interface. The process can be started with one of the "start*" family of methods:

start_in_other_terminal()

start_gvim()

start()

The server can be stopped with "kill" method, but it is recommended to use client's "quit" method .

If given the servername of an existing Vim instance, it can control that instance without the need to start a new process.

A Client would be necessary as an actual interface, though it is possible to use a Server directly to invoke –remote-send and –remote-expr commands on its Vim instance.

Example:

```
>>> vim = Server("My_server")
>>> client = vim.start_in_other_terminal()
>>> client.edit("some_file.txt")
```

check_is_running(*timeout*)
Raises a RuntimeError exception if it can't find, during timeout, a Vim server with the same name as the one given at initialization during timeout.

connect(*timeout=5*)
Connect to a running instance of Vim server. Returns a client. Eg:

```
>>> vim = Server(name="SOME_SERVER_NAME")
>>> client = vim.connect()
```

is_running()
Returns a Boolean indicating wheather server exists and is running.

kill()
Kills the Vim instance started in a subprocess. Returns nothing. It is useless if you connected to server with connect(). In that case use quit() instead.

kill() works with vim, but not with gvim.

quit()
Used to send to server the :qa! command. Useful when we connected to server instead of starting it in a subprocess with start().

remote_expr(*expression*)
Evaluates an expression in the Vim server and returns the result. A wrapper around –remote-expr. Note that a command is not an expression, but a function call or a variable is.

expression - a String with a Vim expression to evaluate.

Returns the String output of the expression. Eg:

remote_expr('&shiftwidth')

remote_send(*keys*)
Sends the given keys to Vim server. A wrapper around –remote-send.

keys - a String with a sequence of Vim-compatible keystrokes.

Returns nothing. Eg:

$ vim –servername VIMRUNNER –remote-send ':qa! <Enter>'

server_list()
Retrieves a list of names of currently running Vim servers.

Returns a List of String server names currently running.

start(*timeout=5*, *testing=False*)
Starts Vim server in a subprocess, eg.:

```
>>> subprocess.call("vim -n --servername GOTOWORD", shell=True)
```

but we don't want to wait for Vim to complete and to block this script so we need some thread like behaviour that is obtained using the multiprocessing module.

testing - flag useful for tests when you don't want to start Vim server

Returns a client connected to Vim server.

start_gvim()
Start a GUI Vim. Returns a Client().

start_in_other_terminal()
Start vim in a terminal other than the one used to run this script (test script) because vim will pollute the output of the test script and vim will malfunction. Returns a Client. We need something like:

x-terminal-emulator -e 'sh -c "python vim_server_no_gui.py"'

It is useful when testing a vim plugin to launch vim in other terminal so that the test script's output doesn't get polluted by vim.

vimrunner.vimrunner.**create_vim_list**(*values*)
creates the Vim editor's equivalent of python's repr(a_list).

```
>>> create_vim_list(['first line', 'second line'])
'["first line", "second line"]'
```

values - a list of strings

We need double quotes not single quotes to create a Vim list. Returns a string that is a properly written Vim list of strings. This result can be fed to vim's eval function to create a list in vim.

CHAPTER 1

Indices and tables

- genindex
- modindex
- search

Python Module Index

v

Index