List of vim plugins I use

August 14, 2011

List of vim plugins I use with mini tutorials

I am a vim user, and by user I mean I do all(not counting using the textarea inside the web browser) of my editing inside vim. Even when I need to use a word processor, I first type my content inside vim and then open the word processor to format it.

As any vim user knows that Vim experience is not complete without the use of plugins, here is the list of plugins I use on daily basis.

A word about my vim setup

I manage my vim/bash configs in separate directory `~/dotfiles`. Here is what it looks like.

```
$ ls
bash  bin  desktop  utils  vim
```

`bash` directory contains my my `.bashrc`, `.bash_aliases` etc. `bin` - not same as `~/bin`, contains scripts I use on day to day basis. `desktop` contains contains configurations exported from my Ubuntu desktop apps(e.g. my compiz-unity profile).

Here is what my `vim` directory looks like

```
$ ls
autoload  bundle  sessions  undodir  vimrc
```

Everything is *soft linked* to the relevant locations from here as shown by following commands:

```
$ ln -s ~/dotfiles/vim ~/.vim
$ ln -s ~/dotfiles/vim/vimrc ~/.vimrc
$ ln -s ~/dotfiles/bash/bashrc ~/.bashrc
$ ln -s ~/dotfiles/bash/aliases ~/.bash_aliases
```

Pathogen helps me keep the `.vim` directory clean. When pathogen is used to manage plugins, it is the only plugin that needs to be installed directly by copying the files into `.vim` directory.

Installation

I recommend starting with a blank `.vim` directory.

```
$ mv ~/.vim ~/vim_old                                              # 1.
Backup your old .vim directory
$ mkdir .vim                                                       # 2. Creat
a blank .vim directory
$ git clone git://github.com/tpope/vim-pathogen.git pathogen       # 3. Clone
the pathogen repo
$ mv pathogen/autoload ~/.vim/autoload                             # 4. Move
pathogen to .vim directory
```

I also track my configurations with git; you can use whatever version control system you prefer(Hg, Bzr or God forbid svn).

```
$ cd .vim
$ git init
$ git add .
$ git commit -m "Initial commit"
```

Now edit your `.vimrc` and add following lines to the top

```
call pathogen#runtime_append_all_bundles()
call pathogen#helptags()
```

That's it! Pathogen is now installed.

Usage

Any other plugins will be installed by simply copying the plugin files into `~/.vim/bundle/plugin_name` directory. Generally, you will download the plugin, extract it and move to `~/.vim/bundle/plugin_name`.

```
$ mv /path/to/plugin ~/.vim/bundle/plugin_name
```

Although not required by pathogen, I prefer to manage plugins using git submodules. Let's install fugitive plugin for Git integration to demonstrate the process.

```
$ cd ~/.vim
$ git submodule add git://github.com/tpope/vim-fugitive.git
bundle/fugitive
$ git submodule init && git submodule update
```

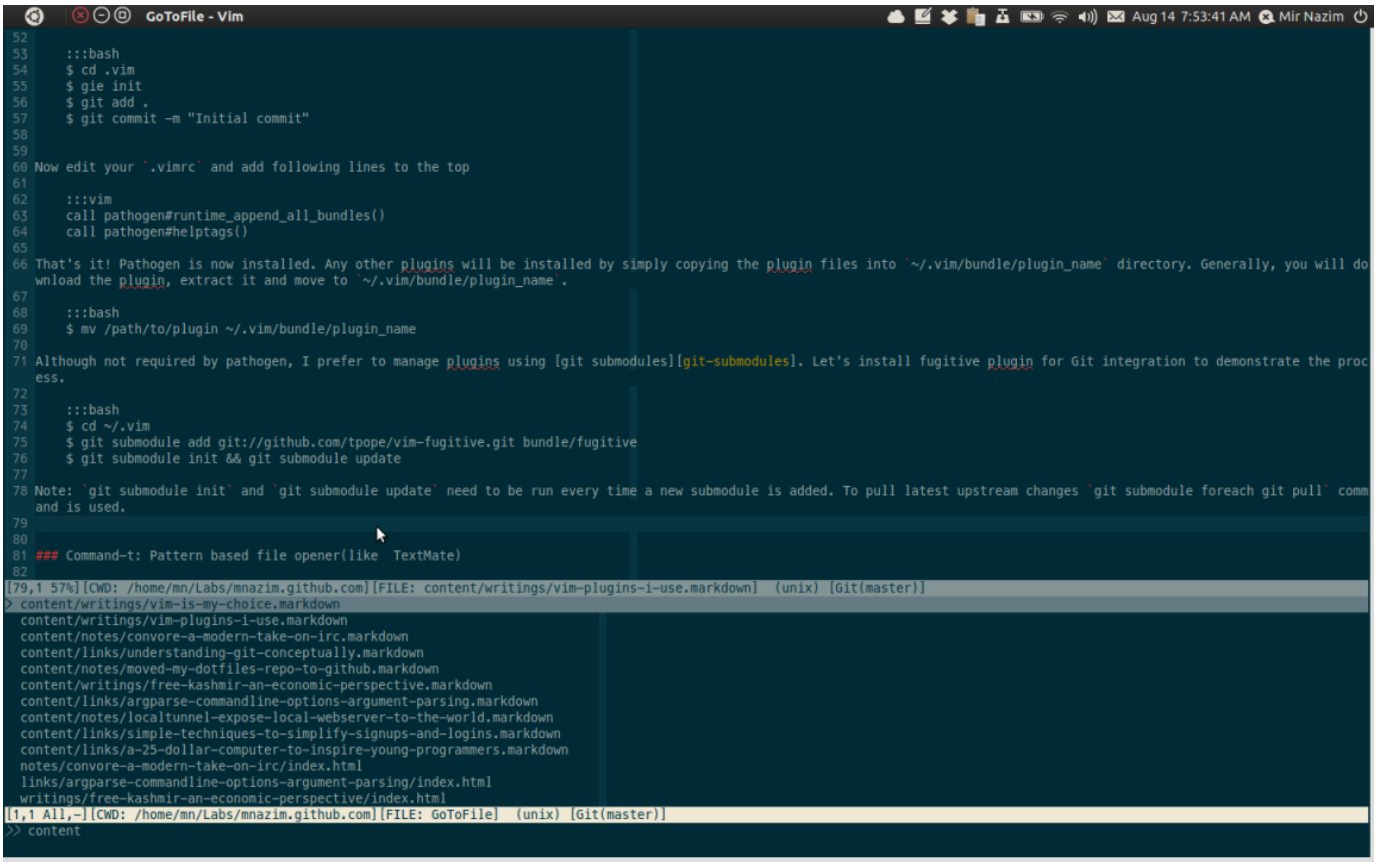
Note: `git submodule init` and `git submodule update` need to be run every time a new

submodule is added. `git submodule foreach git pull` command is used to pull latest upstream changes.

The Command-T plug-in for VIM provides an extremely fast, intuitive mechanism for opening files with a minimal number of keystrokes. It's named "Command-T" because it is inspired by the "Go to File" window bound to `Command-T` key in TextMate.

Files are selected by typing characters that appear in their paths, and are ordered by an algorithm which knows that characters that appear in certain locations (for example, immediately after a path separator) should be given more weight.

Here are is a screenshot of command-t in action while writing this very blog post.



Installation

Command-t is developed in ruby therefore ruby needs to be installed on your system. Here is how to installation on Ubuntu.

```
$ sudo aptitude install ruby ruby-dev
$ git submodule add git://git.wincent.com/command-t.git bundle/command-t
$ git submodule init && git submodule update
$ cd ~/.vim/bundle/command-t/ruby/command-t/
$ ruby extconf.rb
$ make
```

Detailed installation instructions are available at Command-t homepage

Usage

Command-t provides three functions

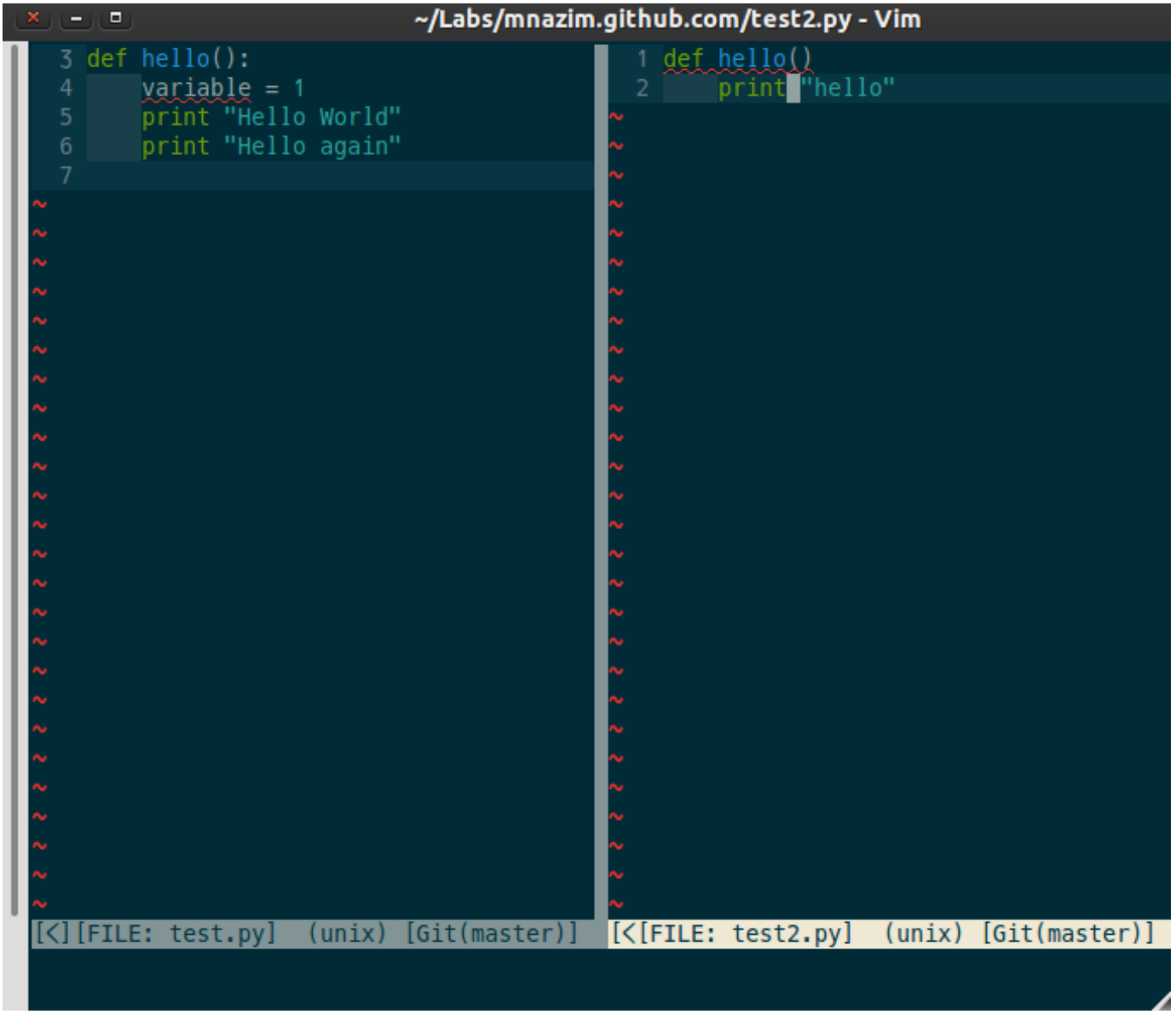
1. `CommandT` - opens filelist in current directory
2. `CommandTBuffer` - opens currently open buffers
3. `CommandTFlush` - re-read file list in current directory

I have mapped these functions as following:

```
noremap <leader>o <Esc>:CommandT<CR>
noremap <leader>0 <Esc>:CommandTFlush<CR>
noremap <leader>m <Esc>:CommandTBuffer<CR>
```

Pyflakes is syntax checking and linting library for python. The vim plugin for same is provides me with syntax check right inside vim. It notifies me of any module I have imported but not used, variables I have assigned and not used, syntax errors etc.

Check out the red squiggly lines pointing out an unused variable and a indentation error in the screenshot below; a relevant error message will appear in the statusbar when the cursor is on the error in question.



Installation

```
$ cd ~/.vim
$ git submodule add git://github.com/kevinw/pyflakes-vim.git
bundle/pyflakes
$ git submodule init && git submodule update
```

Usage

Pyflakes should start working automatically as soon as you install it.

NERDCommenter provides a bunch of key mapping for working with comments in a very fast and efficient manner.

Installation

```
$ cd ~/.vim
$ git submodule add git://github.com/scrooloose/nerdcommenter.git
```

```
bundle/nerdcommenter
$ git submodule init && git submodule update
```

Usage

Some useful key mappings that I use regularly are:

- `[count]<leader>ci` - Toggles the comment state of the selected line(s) individually.
- `[count]<leader>cy`- Same as cc except that the commented line(s) are yanked first.
- `<leader>c$` - Comments the current line from the cursor to the end of line.
- `<leader>cA` - Adds comment delimiters to the end of line and goes into insert mode between them.

For complete list of NERDCommenter commands see `:help NERDCommenter`

If had to pick the most awesome vim plugin, it would definitely be Fugitive. It provides an amazingly deep Git integration for vim.

Installation

```
$ cd ~/.vim
$ git submodule add git://github.com/tpope/vim-fugitive.git
bundle/fugitive
$ git submodule init && git submodule update
```

Usage

I am not even going to attempt describing Fugitive usage here. I would rather point you to the awesome 5 Part Fugitive Screecasts Series by Drew Neil. Go, Learn!

Solarized is the awesome colorscheme for vim(and many other apps) by Ethan Schoonover. It provides both light and dark versions. You have already seen the dark colorscheme in the screenshots included above. Here is a screenshots of light version.

Installation

```
$ cd ~/.vim
$ git submodule add  git://github.com/altercation/vim-colors-solarized.git
bundle/solarized
$ git submodule init && git submodule update
```

Usage

Solarized Vim README contains detailed configuration documentation; here is how I configured my instance

```
set background=dark
let g:solarized_termtrans=1
let g:solarized_termcolors=256
let g:solarized_contrast="high"
let g:solarized_visibility="high"
colorscheme solarized
```

If you want to use solarized in the terminal vim you will need to set `TERM` environment variable.

```
export TERM="xterm-256color"
```

You can also use CSAprox plugin to use gvim themes inside terminal vim.

The rest

While the plugins described above are the ones without which I cannot even imagine working sanely, they are not the only ones. I use a variety of utility and syntax plugins. Some of them are:

- Surround - For editing the surroundings of text.
- Better CSS Syntax - Provides better CSS syntax highlighting.
- Vim CSS Color - Sets background of color hex codes to what they are.
- CSAprox - Allows use of GVim color schemes in almost all terminals
- Syntax plugins for various programming languages like JavaScript, HTML/XML, PHP, etc.

For curious minds, my vim configurations(along with some other stuff) is available at github.com/mnazim/dotfiles.

Updates

August 22, 2011: Corrected some mistakes and updated some obsolete repositories to active ones, thanks to the good people over at Hacker News, Reddit and Stéfan van der Walt who took the time to drop me an email.

August 26, 2011: A few readers reported confusing language in the para defining soft-linking of files and directories inside `~/dotfiles` to relevant locations. Changed naration and added the example commands to be more explicit.