John Anderson

Open Source Hacker

Turning Vim into a modern Python IDE

Posted by sontek on 2011-05-07 04:37:55

Contents

- Intro
- Basic Editing and Debugging
  - Code Folding
  - Window Splits
  - Snippets
  - Task lists
  - Revision History
- Syntax Highlighting and Validation
  - Pep8
- Tab Completion and Documentation
- Code Navigation
  - Buffers
  - Fuzzy Text File Search
  - File Browser
  - Refactoring and Go to definition
  - Searching
- Integration with Git
- Test Integration
  - django nose
  - py.test
- Virtualenv
- Django
- Random Tips

TL;DR:

```
$ git clone https://github.com/sontek/dotfiles.git
$ cd dotfiles
$ ./install.sh vim
```

Download PDF Version

Intro

Back in 2008, I wrote the article Python with a modular IDE (Vim). Years later, I have people e-mailing me and commenting daily asking for more information, even though most of the information in it is outdated. Here is the modern way to work with Python and Vim to achieve the perfect environment.

Because one of the most important parts about a development environment is the ability to easily reproduce across machines, we are going to store our vim configuration in git:

```
$ mkdir ~/.vim/
$ mkdir ~/.vim/{autoload,bundle}
$ cd ~/.vim/
$ git init
```

The purpose of the autoload directory is to automatically load the vim plugin Pathogen, which we'll then use to load all other plugins that are located in the bundle directory. So download pathogen and put it in your autoload folder.

You'll need to add the following to your ~/.vimrc so that pathogen will be loaded properly. Filetype detection must be off when you run the commands so its best to execute them first:

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

Now lets add all of the vim plugins we plan on using as submodules to our git repository:

```
git submodule add http://github.com/tpope/vim-fugitive.git bundle/fugitive
git submodule add https://github.com/msanders/snipmate.vim.git bundle/snipmate
git submodule add https://github.com/tpope/vim-surround.git bundle/surround
git submodule add https://github.com/tpope/vim-git.git bundle/git
git submodule add https://github.com/ervandew/supertab.git bundle/supertab
git submodule add https://github.com/sontek/minibufexpl.vim.git bundle/minibufexpl
git submodule add https://github.com/wincent/Command-T.git bundle/command-t
git submodule add https://github.com/mitechie/pyflakes-pathogen.git
git submodule add https://github.com/mileszs/ack.vim.git bundle/ack
git submodule add https://github.com/sjl/gundo.vim.git bundle/gundo
git submodule add https://github.com/fs111/pydoc.vim.git bundle/pydoc
git submodule add https://github.com/vim-scripts/pep8.git bundle/pep8
git submodule add https://github.com/alfredodeza/pytest.vim.git bundle/py.test
git submodule add https://github.com/reinh/vim-makegreen bundle/makegreen
git submodule add https://github.com/vim-scripts/TaskList.vim.git bundle/tasklist
git submodule add https://github.com/vim-scripts/The-NERD-tree.git bundle/nerdtree
git submodule add https://github.com/sontek/rope-vim.git bundle/ropevim
git submodule init
git submodule update
git submodule foreach git submodule init
git submodule foreach git submodule update
```

Thats it! Now that we've got our vim configuration in git!

Now lets look at how to use each of these plugins to improve the power of vim:

Basic Editing and Debugging

Code Folding

Lets first enable code folding. This makes it a lot easier to organize your code and hide portions that you aren't interested in working on. This is quite easy for Python, since whitespace is required.

In your ~/.vimrc just add:

```
set foldmethod=indent
set foldlevel=99
```

Then you will be able to be inside a method and type 'za' to open and close a fold.

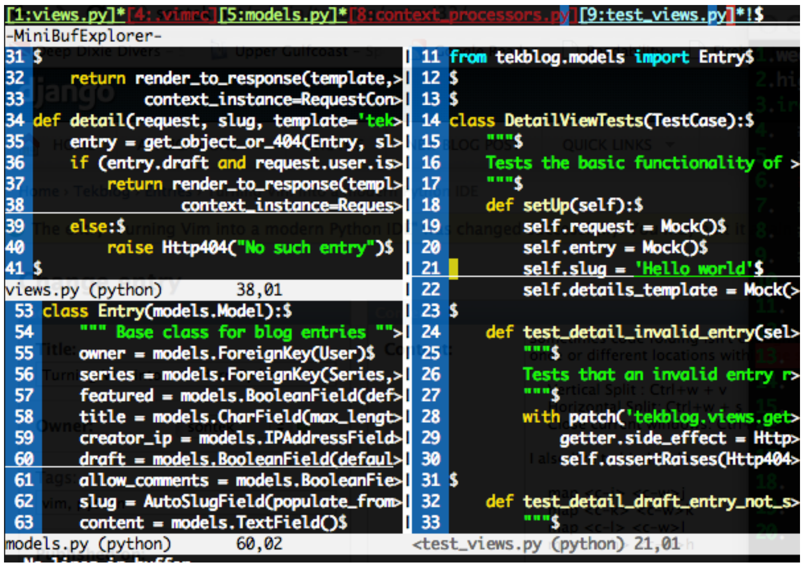
Window Splits

Sometimes code folding isn't enough; you may need to start opening up multiple windows and working on multiple files at once or different locations within the same file. To do this in vim, you can use these shortcuts:

```
Vertical Split : Ctrl+w + v
Horizontal Split: Ctrl+w + s
Close current windows: Ctrl+w + q
```

I also like to bind Ctrl+<movement> keys to move around the windows, instead of using Ctrl+w + <movement>:

```
map <c-j> <c-w>j
map <c-k> <c-w>k
map <c-l> <c-w>l
map <c-h> <c-w>h
```



```
[1:views.py]*[4:.vimrc][5:models.py]*[8:context_processors.py][9:test_views.py]*!$
-MiniBufExplorer-
$
    return render_to_response(template,>
            context_instance=RequestCon>
def detail(request, slug, template='tek>
    entry = get_object_or_404(Entry, sl>
    if (entry.draft and request.user.is>
        return render_to_response(templ>
                context_instance=Reques>
    else:$
        raise Http404("No such entry")$
$
views.py (python)        38,01
class Entry(models.Model):$
    """ Base class for blog entries "">
    owner = models.ForeignKey(User)$
    series = models.ForeignKey(Series,>
    featured = models.BooleanField(def>
    title = models.CharField(max_lengt>
    creator_ip = models.IPAddressField>
    draft = models.BooleanField(defaul>
    allow_comments = models.BooleanFie>
    slug = AutoSlugField(populate_from>
    content = models.TextField()$
models.py (python)       60,02
```

```
from tekblog.models import Entry$
$
$
class DetailViewTests(TestCase):$
    """$
    Tests the basic functionality of >
    """$
    def setUp(self):$
        self.request = Mock()$
        self.entry = Mock()$
        self.slug = 'Hello world'$
        self.details_template = Mock(>
$
    def test_detail_invalid_entry(sel>
        """$
        Tests that an invalid entry r>
        """$
        with patch('tekblog.views.get>
            getter.side_effect = Http>
            self.assertRaises(Http404>
$
    def test_detail_draft_entry_not_s>
        """$
<test_views.py (python) 21,01
```

Snippets

The next tweak that really speeds up development is using snipmate. We've already included it in our bundle/ folder so its already enabled. Try opening up a python file and typing 'def<tab>'. It should stub out a method definition for you and allow you to tab through and fill out the arguments, doc string, etc.

I also like to create my own snippets folder to put in some custom snippets:

```
$ mkdir ~/.vim/snippets
$ vim ~/.vim/snippets/python.snippets
```

Put this in the file:

```
snippet pdb
    import pdb; pdb.set_trace()
```

Now you can type pdb<tab> and it'll insert your breakpoint!

Task lists

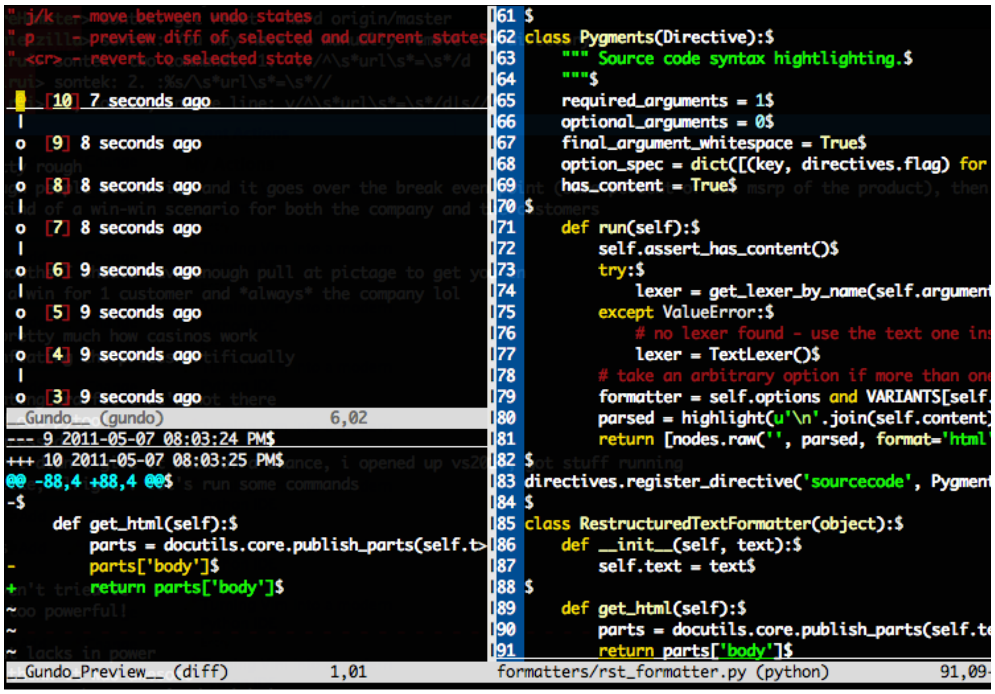
Another really useful thing is to mark some of your code as TODO or FIXME! I know we all like to think we write perfect code, but sometimes you just have to settle and leave a note for yourself to come back later. One of the plugins we included was the tasklist plugin that will allow us to search all open buffers for things to fix. Just add a mapping to open it in ~/.vimrc:

```
map <leader>td <Plug>TaskList
```

Now you can hit <leader>td to open your task list and hit 'q' to close it. You can also hit enter on the task to jump to the buffer and line that it is placed on.

Revision History

The final basic editing tweak I suggest everyone start utilizing is the Gundo plugin. It'll allow you to view diff's of every save on a file you've made and allow you to quickly revert back and forth:



```
" j/k  - move between undo states
" p    - preview diff of selected and current states
" <cr> - revert to selected state

 @  [10] 7 seconds ago
 |
 o  [9] 8 seconds ago
 |
 o  [8] 8 seconds ago
 |
 o  [7] 8 seconds ago
 |
 o  [6] 9 seconds ago
 |
 o  [5] 9 seconds ago
 |
 o  [4] 9 seconds ago
 |
 o  [3] 9 seconds ago
__Gundo__ (gundo)                          6,02
--- 9 2011-05-07 08:03:24 PM$
+++ 10 2011-05-07 08:03:25 PM$
@@ -88,4 +88,4 @@$
-$
     def get_html(self):$
         parts = docutils.core.publish_parts(self.t>
-        parts['body']$
+        return parts['body']$
~
~
~
__Gundo_Preview__ (diff)                   1,01
```

```
$
class Pygments(Directive):$
    """ Source code syntax hightlighting.$
    """$
    required_arguments = 1$
    optional_arguments = 0$
    final_argument_whitespace = True$
    option_spec = dict([(key, directives.flag) for
    has_content = True$
$
    def run(self):$
        self.assert_has_content()$
        try:$
            lexer = get_lexer_by_name(self.argument
        except ValueError:$
            # no lexer found - use the text one ins
            lexer = TextLexer()$
        # take an arbitrary option if more than one
        formatter = self.options and VARIANTS[self.
        parsed = highlight(u'\n'.join(self.content)
        return [nodes.raw('', parsed, format='html'
$
directives.register_directive('sourcecode', Pygment
$
class RestructuredTextFormatter(object):$
    def __init__(self, text):$
        self.text = text$
$
    def get_html(self):$
        parts = docutils.core.publish_parts(self.te
        return parts['body']$
formatters/rst_formatter.py (python)          91,09
```

Just bind a key in your .vimrc to toggle the Gundo window:

```
map <leader>g :GundoToggle<CR>
```

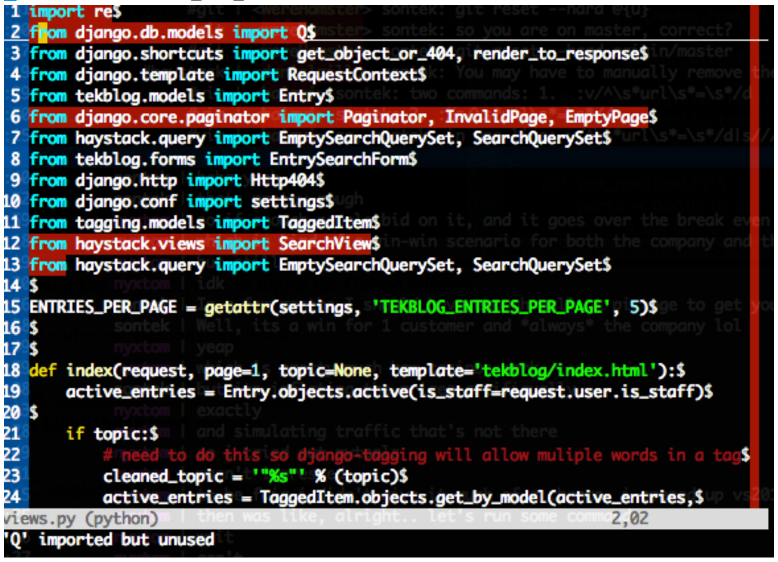
Syntax Highlighting and Validation

Simply enable syntax highlighting in your ~/.vimrc:

```
syntax on                           " syntax highlighing
filetype on                          " try to detect filetypes
filetype plugin indent on    " enable loading indent file for filetype
```

Because we enabled pyflakes when we added it as a submodule in ~/.vim/bundle, it will notify you about unused imports and invalid syntax. It will save you a lot of time saving and running just to find out you missed a colon. I like to tell it not use the quickfix window:

```
let g:pyflakes_use_quickfix = 0
```

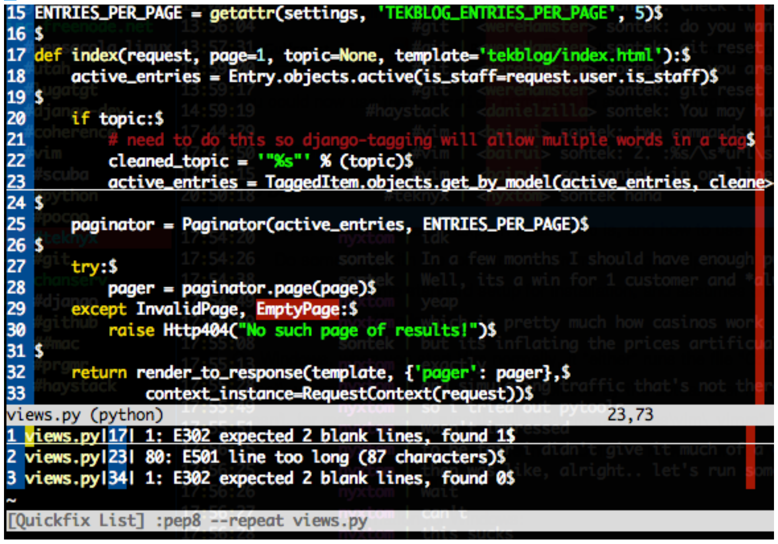


```
import re$
from django.db.models import Q$
from django.shortcuts import get_object_or_404, render_to_response$
from django.template import RequestContext$
from tekblog.models import Entry$
from django.core.paginator import Paginator, InvalidPage, EmptyPage$
from haystack.query import EmptySearchQuerySet, SearchQuerySet$
from tekblog.forms import EntrySearchForm$
from django.http import Http404$
from django.conf import settings$
from tagging.models import TaggedItem$
from haystack.views import SearchView$
from haystack.query import EmptySearchQuerySet, SearchQuerySet$
$
ENTRIES_PER_PAGE = getattr(settings, 'TEKBLOG_ENTRIES_PER_PAGE', 5)$
$
$
def index(request, page=1, topic=None, template='tekblog/index.html'):$
    active_entries = Entry.objects.active(is_staff=request.user.is_staff)$
$
    if topic:$
        # need to do this so django-tagging will allow muliple words in a tag$
        cleaned_topic = '"%s"' % (topic)$
        active_entries = TaggedItem.objects.get_by_model(active_entries,$
views.py (python)                                                  2,02
'Q' imported but unused
```

Pep8

The final plugin that really helps validate your code is the pep8 plugin, it'll make sure your code is consistent across all projects. Add a key mapping to your ~/.vimrc and then you'll be able to jump to each of the pep8 violations in the quickfix window:

```
let g:pep8_map='<leader>8'
```



```
15 ENTRIES_PER_PAGE = getattr(settings, 'TEKBLOG_ENTRIES_PER_PAGE', 5)$
16 $
17 def index(request, page=1, topic=None, template='tekblog/index.html'):$
18     active_entries = Entry.objects.active(is_staff=request.user.is_staff)$
19 $
20     if topic:$
21         # need to do this so django-tagging will allow muliple words in a tag$
22         cleaned_topic = '"%s"' % (topic)$
23         active_entries = TaggedItem.objects.get_by_model(active_entries, cleane>
24 $
25     paginator = Paginator(active_entries, ENTRIES_PER_PAGE)$
26 $
27     try:$
28         pager = paginator.page(page)$
29     except InvalidPage, EmptyPage:$
30         raise Http404("No such page of results!")$
31 $
32     return render_to_response(template, {'pager': pager},$
33             context_instance=RequestContext(request))$
views.py (python)                                                  23,73
1 views.py|17| 1: E302 expected 2 blank lines, found 1$
2 views.py|23| 80: E501 line too long (87 characters)$
3 views.py|34| 1: E302 expected 2 blank lines, found 0$
~
[Quickfix List] :pep8 --repeat views.py
```

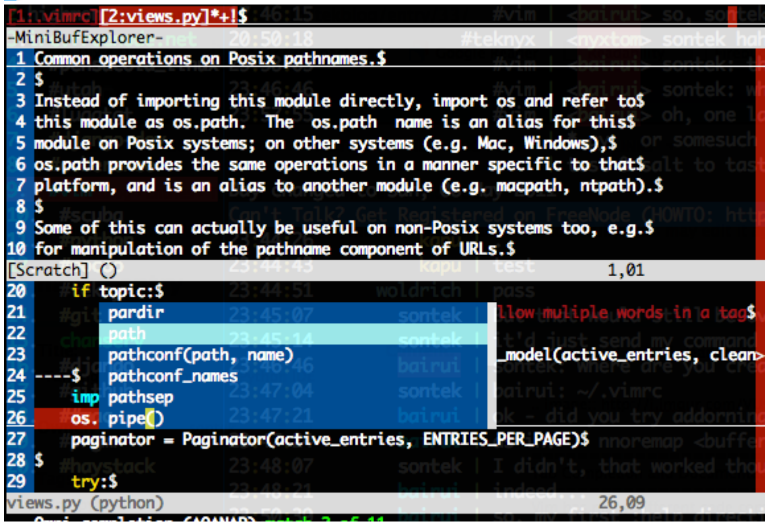
Tab Completion and Documentation

Vim has many different code completion options. We are going to use the SuperTab plugin to check the context of the code you are working on and choose the best for the situation. We've already enabled the SuperTab plugin in the bundle/ folder, so we just have to configure it to be context sensitive and to enable omni code completion in your ~/.vimrc:

```
au FileType python set omnifunc=pythoncomplete#Complete
let g:SuperTabDefaultCompletionType = "context"
```

Now we just enable the menu and pydoc preview to get the most useful information out of the code completion:

```
set completeopt=menuone,longest,preview
```



```
[1:.vimrc] [2:views.py]*+!$
-MiniBufExplorer-
Common operations on Posix pathnames.$
$
Instead of importing this module directly, import os and refer to$
this module as os.path.  The os.path name is an alias for this$
module on Posix systems; on other systems (e.g. Mac, Windows),$
os.path provides the same operations in a manner specific to that$
platform, and is an alias to another module (e.g. macpath, ntpath).$
$
Some of this can actually be useful on non-Posix systems too, e.g.$
for manipulation of the pathname component of URLs.$
[Scratch] ()                                          1,01
    if topic:$
        pardir                                  llow muliple words in a tag$
        path
        pathconf(path, name)                    _model(active_entries, clean>
----$   pathconf_names
    imp pathsep
    os. pipe()
    paginator = Paginator(active_entries, ENTRIES_PER_PAGE)$
$
    try:$
views.py (python)                                     26,09
```

We also enabled the pydoc plugin at the beginning with all the submodules; that gives us the ability to hit <leader>pw when our cursor is on a module and have a new window open with the whole documentation page for it.

Code Navigation

Buffers

The most important part about navigating code within vim, is to completely understand how to use buffers. There is no reason to use tabs. Open files with :e <filename> to place in a buffer. We already installed the minibufexpl plugin, so you will already visually see every buffer opened. You can also get a list of them doing :buffers.

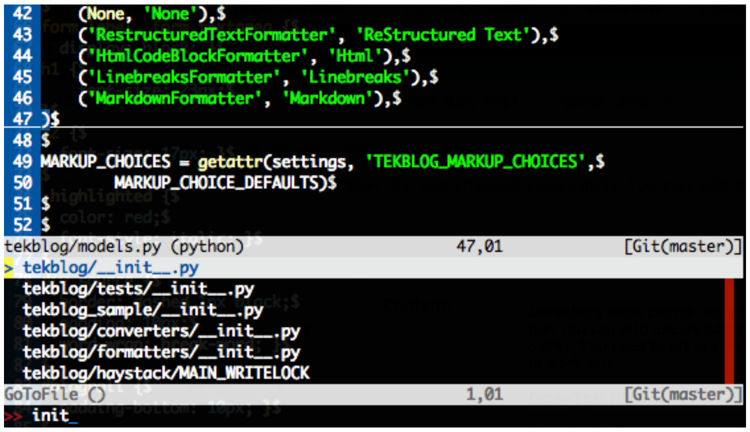
You can switch between the buffers using b<number>, such as :b1 for the first buffer. You can also use its name to match, so you can type :b mod<tab> to autocomplete opening the models.py buffer. You need to make sure you are using the minibufexpl from my github since it has patches that make it much better to work with.

To close a buffer you use :bd or :bw.

Fuzzy Text File Search

To make finding and opening files within your project even easier, we are going to use the command-t plugin. It does have some parts that need to be compiled, so its not already installed by adding it as a submodule. Go to your ~/.vim/bundle/command-t folder and run 'rake make'. Yes you need ruby installed. By default, command-t is bound to <leader>t. This will use fuzzy text matching to find any file in your project.

It also supports searching only through opened buffers, instead of files using <leader>b.

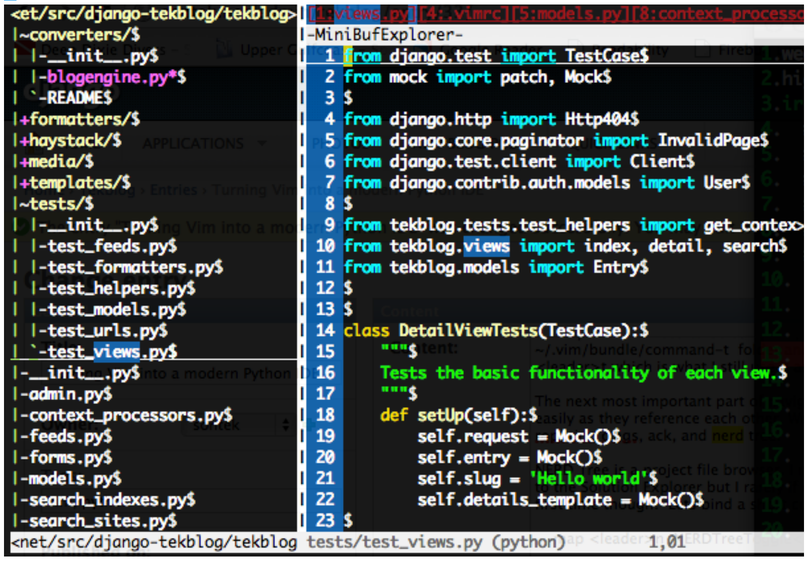


```
    (None, 'None'),$
    ('RestructuredTextFormatter', 'ReStructured Text'),$
    ('HtmlCodeBlockFormatter', 'Html'),$
    ('LinebreaksFormatter', 'Linebreaks'),$
    ('MarkdownFormatter', 'Markdown'),$
)$
$
MARKUP_CHOICES = getattr(settings, 'TEKBLOG_MARKUP_CHOICES',$
        MARKUP_CHOICE_DEFAULTS)$
$
$
tekblog/models.py (python)                     47,01            [Git(master)]
> tekblog/__init__.py
  tekblog/tests/__init__.py
  tekblog_sample/__init__.py
  tekblog/converters/__init__.py
  tekblog/formatters/__init__.py
  tekblog/haystack/MAIN_WRITELOCK
GoToFile ()                                     1,01            [Git(master)]
>> init_
```

File Browser

NERD Tree is a project file browser. I must admit I used this heavily back when I was migrating from Visual Studio and used to the Solution Explorer, but I rarely use it anymore. Command-T is usually all you'll need. It is useful when you are getting to know a new codebase for the first time though. Lets bind a shortcut key for opening it:

```
map <leader>n :NERDTreeToggle<CR>
```



```
<et/src/django-tekblog/tekblog>
|~converters/$
| |-__init__.py$
| |-blogengine.py*$
| `-README$
|+formatters/$
|+haystack/$
|+media/$
|+templates/$
|~tests/$
| |-__init__.py$
| |-test_feeds.py$
| |-test_formatters.py$
| |-test_helpers.py$
| |-test_models.py$
| |-test_urls.py$
| `-test_views.py$
|-__init__.py$
|-admin.py$
|-context_processors.py$
|-feeds.py$
|-forms.py$
|-models.py$
|-search_indexes.py$
|-search_sites.py$
<net/src/django-tekblog/tekblog
```

```
[1:views.py][4:.vimrc][5:models.py][8:context_processo
-MiniBufExplorer-
from django.test import TestCase$
from mock import patch, Mock$
$
from django.http import Http404$
from django.core.paginator import InvalidPage$
from django.test.client import Client$
from django.contrib.auth.models import User$
$
from tekblog.tests.test_helpers import get_contex>
from tekblog.views import index, detail, search$
from tekblog.models import Entry$
$
$
class DetailViewTests(TestCase):$
    """$
    Tests the basic functionality of each view.$
    """$
    def setUp(self):$
        self.request = Mock()$
        self.entry = Mock()$
        self.slug = 'Hello world'$
        self.details_template = Mock()$
$
tests/test_views.py (python)                 1,01
```

Refactoring and Go to definition

Ropevim is also a great tool that will allow you to navigate around your code. It supports automatically inserting import statements, goto definition, refactoring, and code completion. You'll really want to read up on everything it does, but the two big things I use it for is to jump to function or class definitions quickly and to rename things (including all their references).

For instance, if you are using django and you place your cursor over the class models.Model you reference and then called :RopeGotoDefintion, it would jump you straight to the django library to that class definition. We already have it installed in our bundles, so we bind it to a key to use it:

```
map <leader>j :RopeGotoDefinition<CR>
map <leader>r :RopeRename<CR>
```

Searching

The final tool that really speeds up navigating your code is the Ack plugin. Ack is similar to grep, but much better in my opinion. You can fuzzy text search for anything in your code (variable name, class, method, etc) and it'll give you a list of files and line numbers where they are defined so you can quickly cycle through them. Just bind the searching to a key:

```
nmap <leader>a <Esc>:Ack!
```

We use ! at the end of it so it doesn't open the first result automatically.

Integration with Git

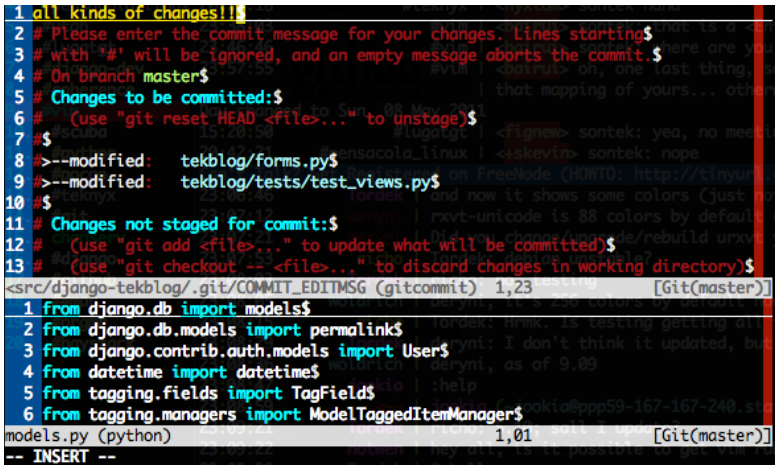
We installed 2 plugins, git.vim and fugitive, that give us all the integration we need. Git.vim will provide us syntax highlighting for git configuration files; fugitive provides a great interface for interacting with git including getting diffs, status updates, committing, and moving files.

Fugitive also allows you to view what branch you are working in directly from vim. Add this to your statusline in ~/.vimrc:

```
%{fugitive#statusline()}
```

The big commands you need to know:

- **Gblame**: This allows you to view a line by line comparison of who the last person to touch that line of code is.
- **Gwrite**: This will stage your file for commit, basically doing git add <filename>
- **Gread**: This will basically run a git checkout <filename>
- **Gcommit**: This will just run git commit. Since its in a vim buffer, you can use keyword completion (Ctrl-N), like test_all<Ctrl-N> to find the method name in your buffer and complete it for the commit message. You can also use + and - on the filenames in the message to stage/unstage them for the commit.



```
all kinds of changes!!$
# Please enter the commit message for your changes. Lines starting$
# with '#' will be ignored, and an empty message aborts the commit.$
# On branch master$
# Changes to be committed:$
#   (use "git reset HEAD <file>..." to unstage)$
#$
#>--modified:   tekblog/forms.py$
#>--modified:   tekblog/tests/test_views.py$
#$
# Changes not staged for commit:$
#   (use "git add <file>..." to update what will be committed)$
#   (use "git checkout -- <file>..." to discard changes in working directory)$
<src/django-tekblog/.git/COMMIT_EDITMSG (gitcommit)  1,23           [Git(master)]
from django.db import models$
from django.db.models import permalink$
from django.contrib.auth.models import User$
from datetime import datetime$
from tagging.fields import TagField$
from tagging.managers import ModelTaggedItemManager$
models.py (python)                                   1,01           [Git(master)]
-- INSERT --
```

Test Integration

django nose

Test runner integration really depends on the testing library you are using and what type of tests you are running but we included a great generic plugin called MakeGreen that executes off of vim's makeprg variable. So for instance, if you are using django with django-nose you could define a shortcut key in your ~/.vimrc like this:

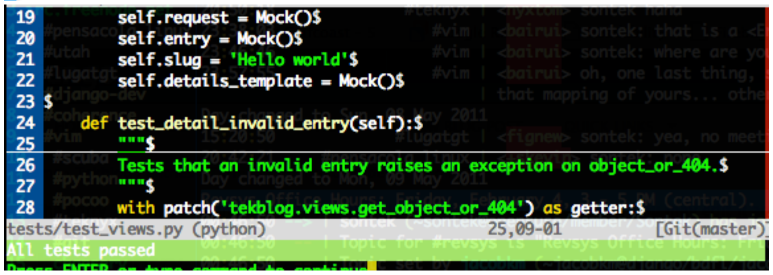
```
map <leader>dt :set makeprg=python\ manage.py\ test\|:call MakeGreen()<CR>
```

This will just give you a green bar at the bottom of vim if your test passed or a red bar with the message of the failed test if it doesn't. Very simple.

py.test

I also included the py.test vim plugin for those who prefer it. This plugin has a lot more functionality including executing individual tests by class, file, or method. You can also cycle through the individual assertion errors. I have the following bindings:

```
" Execute the tests
nmap <silent><Leader>tf <Esc>:Pytest file<CR>
nmap <silent><Leader>tc <Esc>:Pytest class<CR>
nmap <silent><Leader>tm <Esc>:Pytest method<CR>
" cycle through test errors
nmap <silent><Leader>tn <Esc>:Pytest next<CR>
nmap <silent><Leader>tp <Esc>:Pytest previous<CR>
nmap <silent><Leader>te <Esc>:Pytest error<CR>
```



```
        self.request = Mock()$
        self.entry = Mock()$
        self.slug = 'Hello world'$
        self.details_template = Mock()$
$
    def test_detail_invalid_entry(self):$
        """$
        Tests that an invalid entry raises an exception on object_or_404.$
        """$
        with patch('tekblog.views.get_object_or_404') as getter:$
tests/test_views.py (python)                    25,09-01        [Git(master)]
All tests passed
```

Virtualenv

Vim doesn't realize that you are in a virtualenv so it wont give you code completion for libraries only installed there. Add the following script to your ~/.vimrc to fix it:

```
" Add the virtualenv's site-packages to vim path
py << EOF
import os.path
import sys
import vim
if 'VIRTUAL_ENV' in os.environ:
    project_base_dir = os.environ['VIRTUAL_ENV']
    sys.path.insert(0, project_base_dir)
    activate_this = os.path.join(project_base_dir, 'bin/activate_this.py')
    execfile(activate_this, dict(__file__=activate_this))
EOF
```

Django

The only true django tweak I make is before I open vim I'll export the DJANGO_SETTINGS_MODULE environment so that I get code completion for django modules as well:

```
export DJANGO_SETTINGS_MODULE=project.settings
```

Random Tips

If you want to find a new color scheme just go to http://code.google.com/p/vimcolorschemetest/ to preview a large selection.