# SublimeGit Documentation

***Release 1.0.36***

**Michael Pedersen**

**Jun 22, 2017**

# Contents

SublimeGit is a full-featured Git plugin for Sublime Text 2. It has been developed to be easy to get started with. If you're used to Git and dealing with Sublime Text packages, you can probably just install SublimeGit, and get right to work.

If installing Sublime Text packages, or using Git, is new to you, the *Quickstart* is a great place to start. It will get you set up, so you can go on to the tutorial.

---

**Note:** This documentation assumes some familiarity with Git. If you are not familiar with Git, be sure to check out the *More Information* section which contain links to a couple of resources for learning Git.

---

CHAPTER 1

Contents

# Quickstart

## Prerequisites

### Sublime Text 2 or 3

Needless to say, Sublime Text is required to use SublimeGit. Any version of Sublime Text 2 or 3 should work.

### Git

SublimeGit uses the Git command line interface, so you will need a recent version of Git. SublimeGit has been tested on Git 1.8+. To download a version of Git for your operating system, go to http://git-scm.com/downloads. If you are currently using version 1.7 or lower, some commands probably won't work.

You should make sure that git is accesible on your path. You can do this by running `git --version` in your terminal:

```
$ git --version
git version 1.8
```

**Note:** If you start Sublime Text from the terminal (e.g. using the `subl` command on OS X) your path inside Sublime Text might be different from the path you get if you start Sublime Text by clicking on the application.

To see your current path in Sublime Text, open up the console by selecting **View > Show Console** and execute the following python snippet:

```
import os; print os.getenv('PATH')
```

To verify that you have access to the Git executable from within Sublime Text, you can execute the following snippet, which will print `0` if everything worked as expected:

```
import os; os.system('git --version')
```

If this returns anything other than `0` you might need to explicitly set the path to your git executable. See the section *Git Executable Path* for information on how to do this.

### Git Configuration

For the moment, SublimeGit assumes that you have your environment set up so that commands working with remotes (e.g. pull, push and fetch) does not need to ask for user authentication. If that's not the case, and git asks for your username and password when pushing or pulling, then you will need to follow one of these fixes to make sure SublimeGit runs smoothly:

**SSH remotes:** When using SSH remotes with private keys which use passphrases, git will ask for the passphrase to authenticate. There is a safe way to make sure the passphrase is saved, and GitHub has a great guide to using it: https://help.github.com/articles/working-with-ssh-key-passphrases

**HTTPS remotes:** If you prefer HTTPS checkouts, then you will need to follow this guide: https://help.github.com/articles/set-up-git#password-caching

> **Warning:** It seems there can be some problems on Windows, especially when using git-bash and/or private keys with passphrases. For more information, and possible solution please see *Nothing Happens When Pushing, Pulling or Fetching From a Remote*

## Installation

There are many ways to install a package in Sublime Text, but we strongly recommend the use of Package Control, which makes it easy to install and uninstall packages, as well as automatically keeping them up to date. If you are not already using it, you should give it a try.

### Using Package Control

1. Open the Command Palette using **shift+command+p** (OS X) or **shift+ctrl+p** (Windows/Linux) or by selecting **Tools > Command Palette** from the menu bar.
2. Find and select the command **Package Control: Install Package**.
3. Find and select **SublimeGit**.
4. Restart Sublime Text.

**Note:** When you select the **Install Packages** command, it might take a little while for the list of packages to show up. You should be able to see that Package Control is working by watching the spinner in the lower left corner of the window.

### Installing From Package

1. Download the SublimeGit.zip file from https://release.sublimegit.net/SublimeGit.zip.
2. Unzip the package inside your Sublime Text package directory.

- **Windows**: %APPDATA%Sublime Text 2Packages
- **OS X**: ~/Library/Application Support/Sublime Text 2/Packages
- **Linux**: ~/.config/sublime-text-2/Packages

3. Restart Sublime Text.

---

**Note:** Note: If you are unsure where your Sublime Text package directory is, or it is hidden, you can browse to it by selecting **Preferences > Browse Packages** from within Sublime Text.

---

## Configuration

SublimeGit comes with sensible defaults, so if you don't need to add a license, and you can execute the command **Git: Version**, you can skip straight to the *Tutorial*.

### Git Executable Path

To open the default settings for SublimeGit, go to **Preferences > Package Settings > SublimeGit > Settings - Default**. This will show the default settings for SublimeGit. But do not edit this file! Instead, open up **Preferences > Package Settings > SublimeGit > Settings - User** and copy over any settings you wish to change.

If git is not on your path, and it's not possible for you to put git on your path (such as in a very controlled environment where you don't have administrator rights), then you can change the **git_executables** settings to point directly at your git installation.

Be sure to copy the entire thing into your **Settings - User** file, and change the paths accordingly. Be aware that each item in the list will be quoted on its own.

After performing these changes, your user settings might look like this:

```
{
    "git_executables": {
        "git": ["/usr/local/bin/git"],
        "git_flow": ["/usr/local/bin/git", "flow"],
        "legit": ["legit"]
    }
}
```

If you don't use the extensions, there is no need to change their paths.

### Enabling or Disabling Plugins

If you don't use a plugin, it might be annoying that its commands keep showing up. Change the **git_extensions** setting to get rid of them. After disabling git-flow, your local settings file would look like this:

```
{
    "git_extensions": {
        "git_flow": false,
        "legit": true
    }
}
```

## Adding a License

If you decide to buy a license, the email you receive will contain information on how to install it. There are two ways to do it, depending on how comfortable you are with Sublime Text. Also, we love You.

### Automatic

Run the command **SublimeGit: Add License** and follow the instructions. Almost couldn't be easier!

### Manual

Simply add the following to your SublimeGit User Settings file:

```
"email": "MY_EMAIL",
"product_key": "MY_LICENSE_KEY"
```

Replacing `MY_EMAIL` and `MY_LICENSE_KEY` with the correct values. If you've lost your license, send us an email at support@sublimegit.net and we'll get you sorted out.

---

**Note:** To find the correct settings file, navigate to **Preferences > Package Settings > SublimeGit > Settings - User**

---

## Using SublimeGit

Once you're all set up you should jump head-first into the *Tutorial*, which will take you through some basics on using SublimeGit.

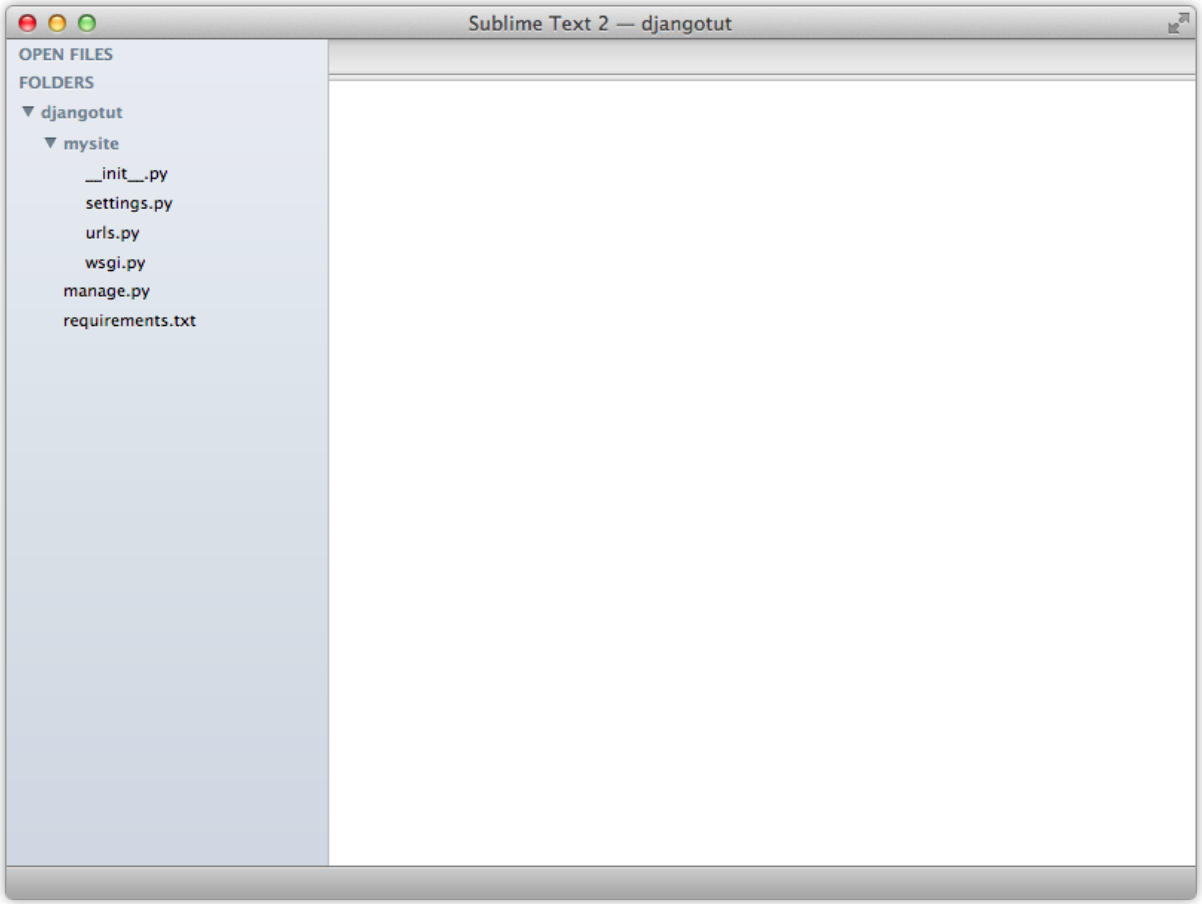
Alternatively, you can jump straight to the *Commands Reference*.

# Tutorial

This tutorial will take you through the usual stages of using SublimeGit for managing a project. We will go through building the Django tutorial application, and managing it in git. We'll be skipping lightly over the coding parts and focusing on using SublimeGit, so it should be usable for any project using git.

## Getting Set Up

The first thing we've done is made a virtual environment, set up a requirements.txt, installed django and initialized a django project. That means our project folder now looks something like this:




## Initialize a Repository

Now, there isn't much work worth saving in this project yet, but we're gonna start adding some, so now might be a good time to put the project in git.

To do this, first make sure that you have the project open in Sublime Text. It doesn't need to be opened as a Sublime Text project, and personally I prefer to just open folders directly from the command line, or through the **File > Open...** menu.

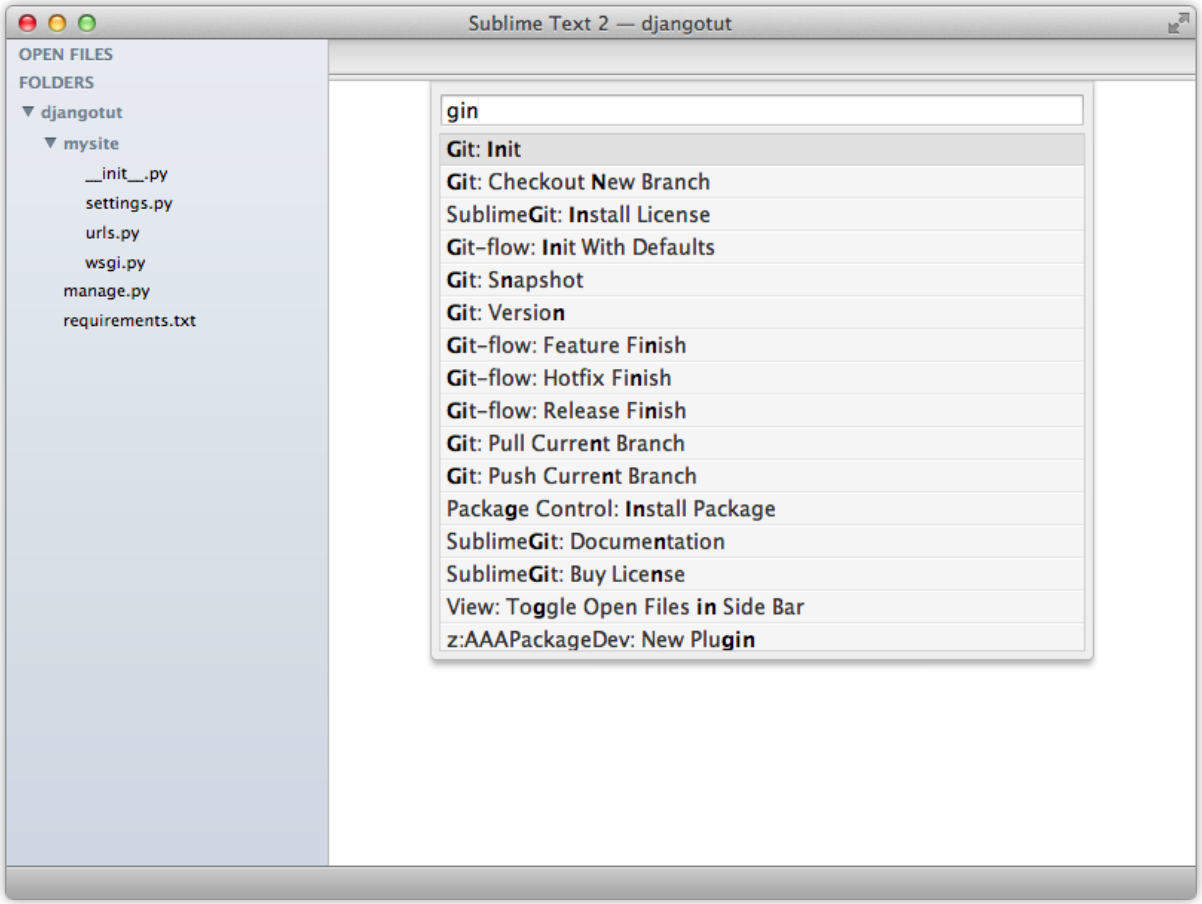
---

**Note:** Notice how we've opened the entire folder in Sublime Text. This isn't required, but it will make initializing the git repository much smoother, since SublimeGit will have an easier time figuring out where to put it.

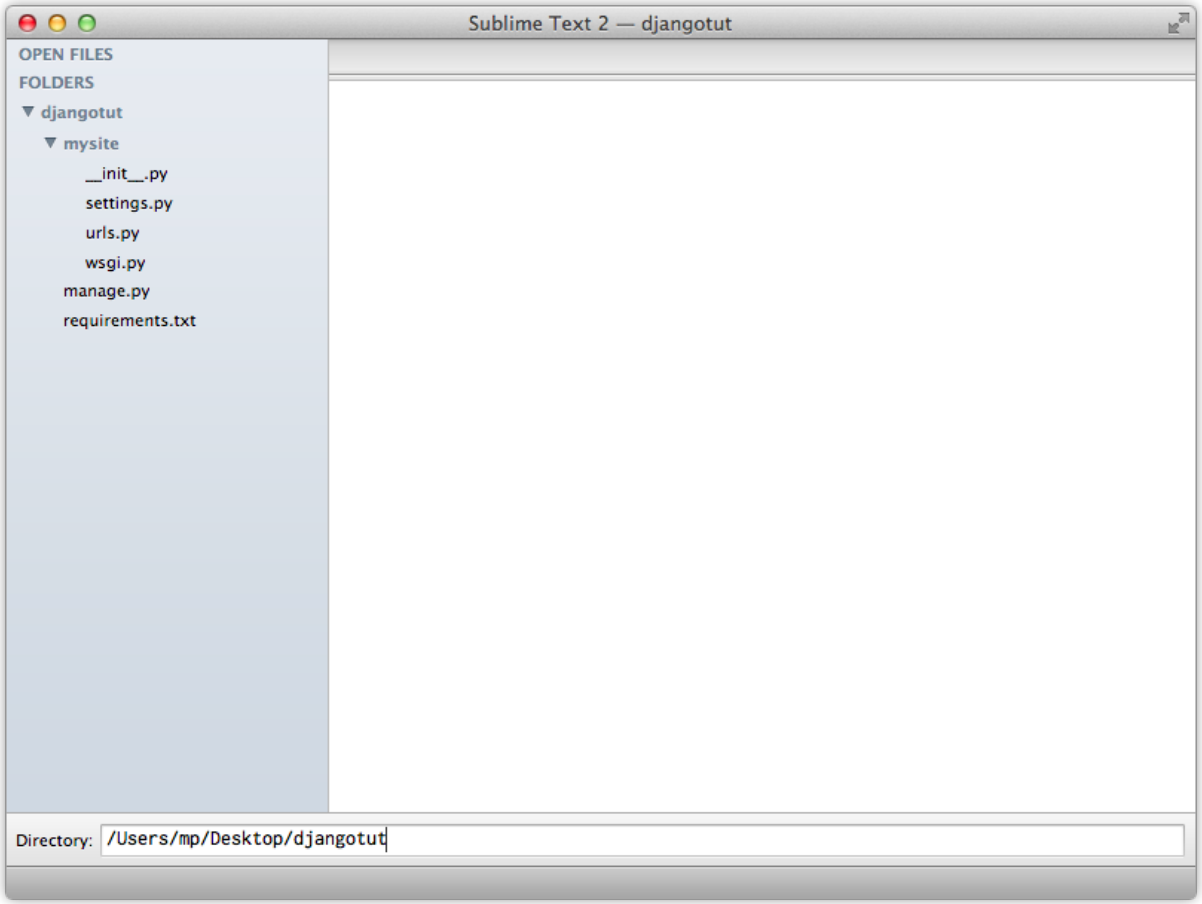
See the *Git: Init* command for more information on how SublimeGit chooses a path for the repo.

---

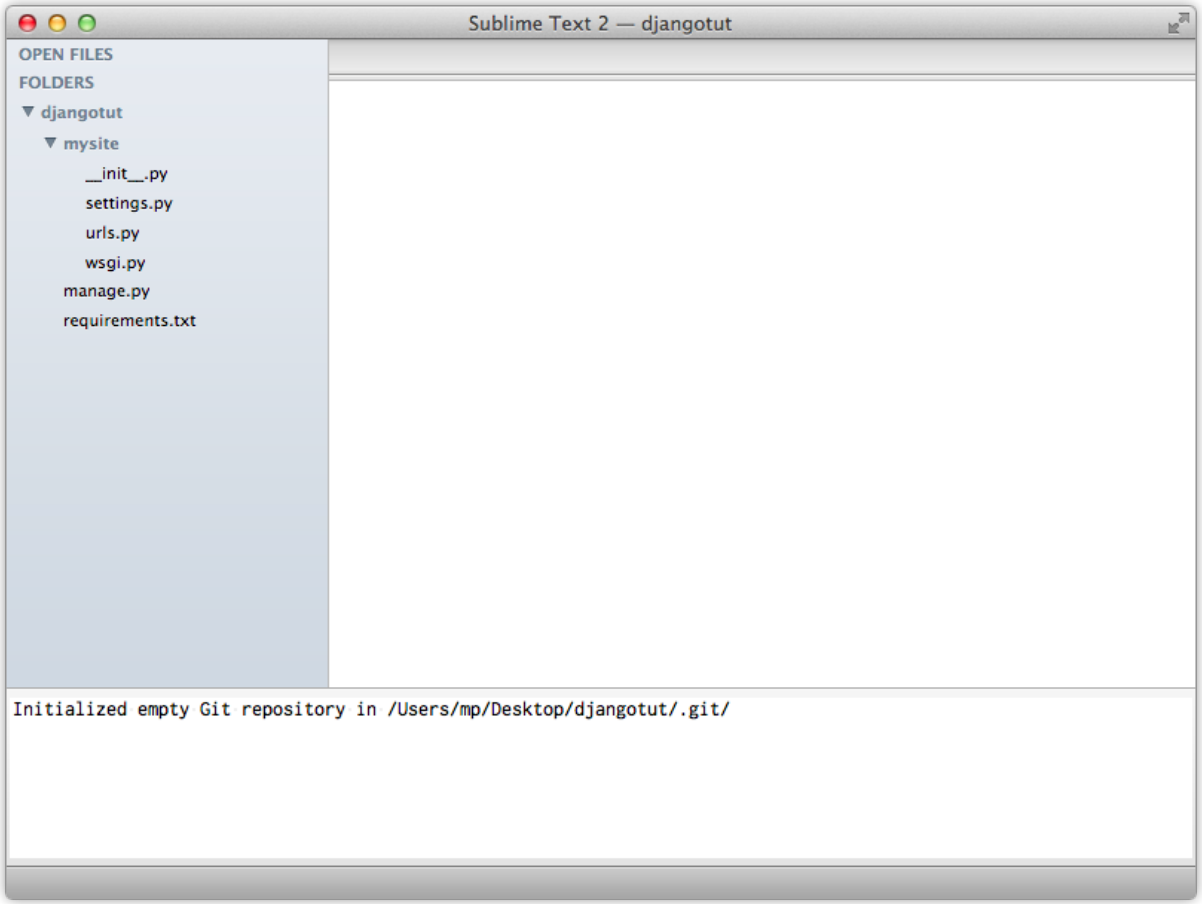
To initialize the git repository, open the command palette, and find the **Git: Init** command. But wait a minute. You don't want to be typing out full command names for everything. Luckily, Sublime Text is pretty intelligent about matching commands, so to find the **Git: Init** command, we should just be able to start typing. Generally, all of the commands in SublimeGit start with the **Git:** prefix, so let's try just typing `ginit`.




Once we're at `gin`, Sublime Text should have selected the right command. Now press enter, and you will be presented with a choice of where to put the repository on the bottom of the screen:




If you've opened our project as a folder, then the default value should be sufficient, and you can press enter to select it. After creating the git repository, SublimeGit will show you the output of the git command in a console window:




To dismiss this console window, press `escape`.

---

**Note:** Another way to initialize a repository is to just start using the **Git: Status** command. If you aren't on a repository, SublimeGit will ask you to initialize one.

---

## Adding Content

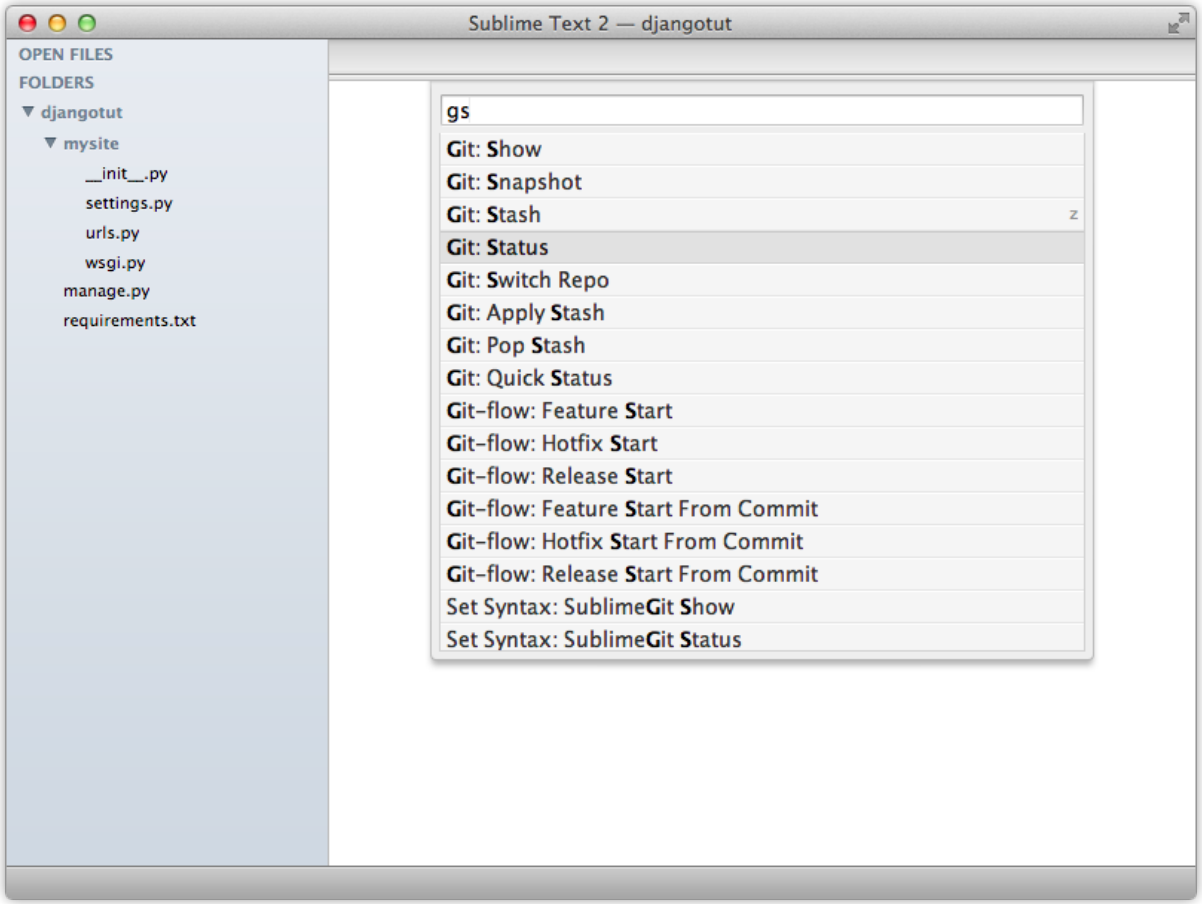
So now that we have an empty repository, we can start adding our files.

**Warning:** Using SublimeGit to perform the initial commit on a huge project (1000+ files) might not be the best way to go. Getting the list of untracked files from git, and formatting them nicely can take some time.

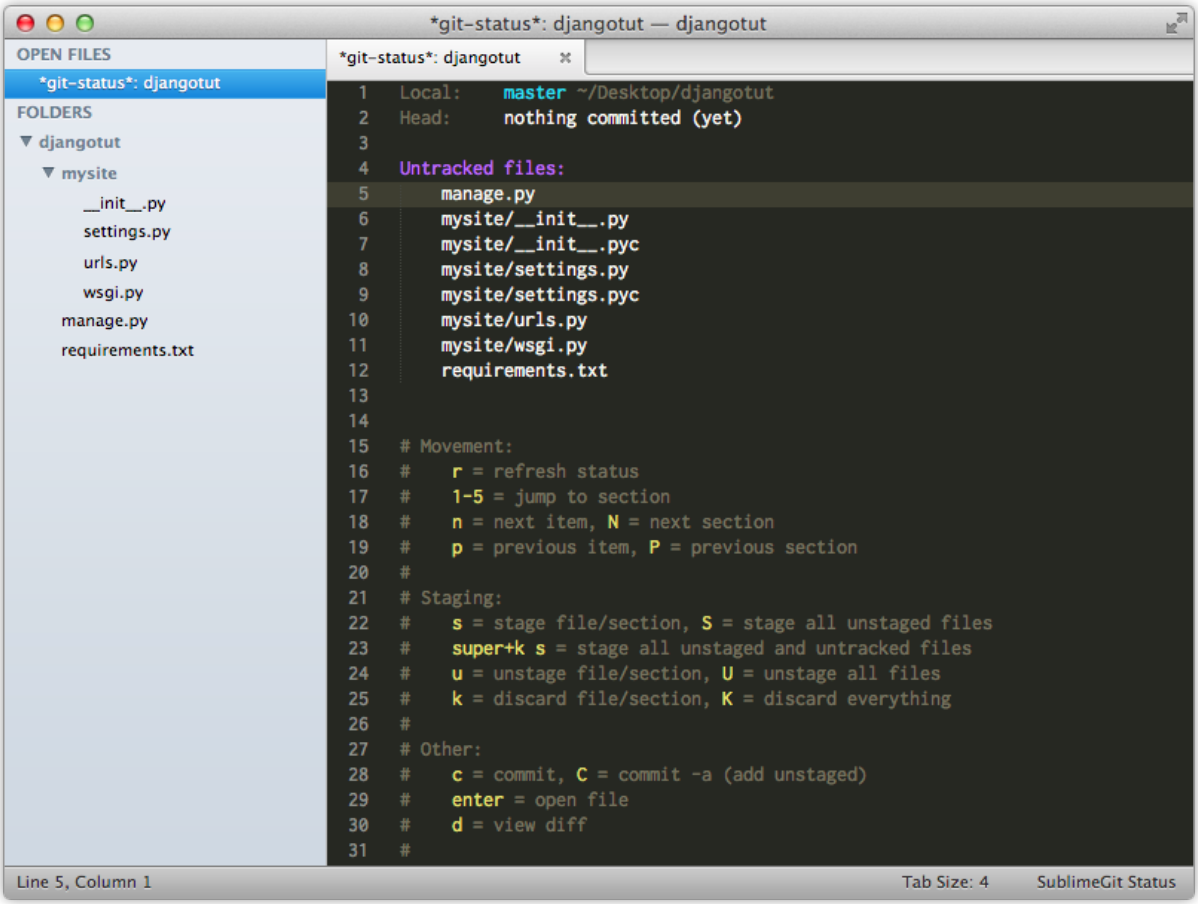
If you are dealing with a huge project, you might want to use the command line for the initial commit.

### The Status View

Most of the adding/staging/unstaging and so on happens from the *Git: Status* command, so let's run that. Typing `gs` in the command palette should be sufficient to bring it up:




After executing the command, we should see a status view looking something like this:




This view contains the following information for our new project:

**Local** Information on our current branch (master), and the location of our repository (~/Desktop/djangotut)

**Head** Info about the current HEAD commit. Since we haven't committed anything yet, there is nothing to show. After our first commit, we will be able to see the abbreviated SHA1, and the first line of the commit message.

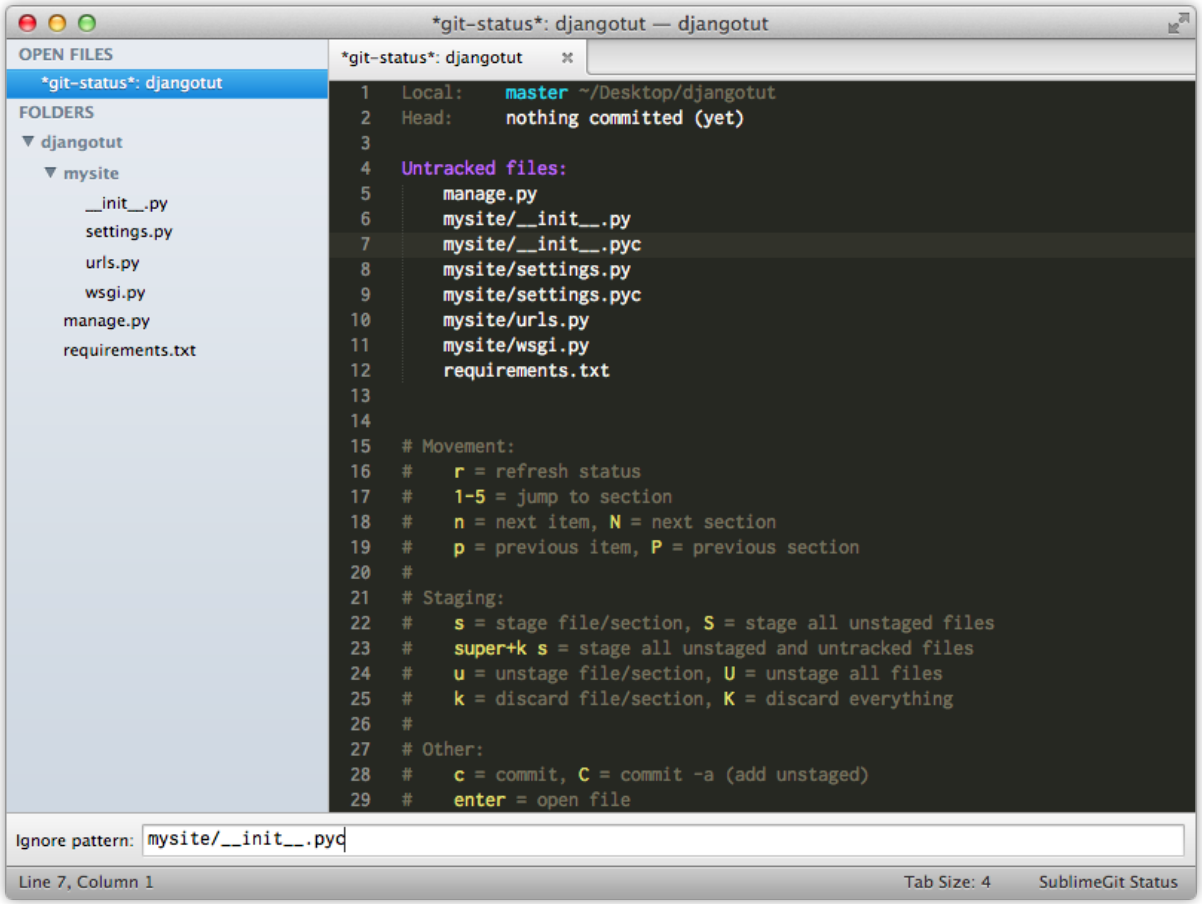
**Untracked files** This section shows files inside our project which have yet to be added to git. When we get a little further, some more sections will show up, such as unstaged changes, staged changes and stashes.

**Help** The bottom of the view shows the available keyboard shortcuts.

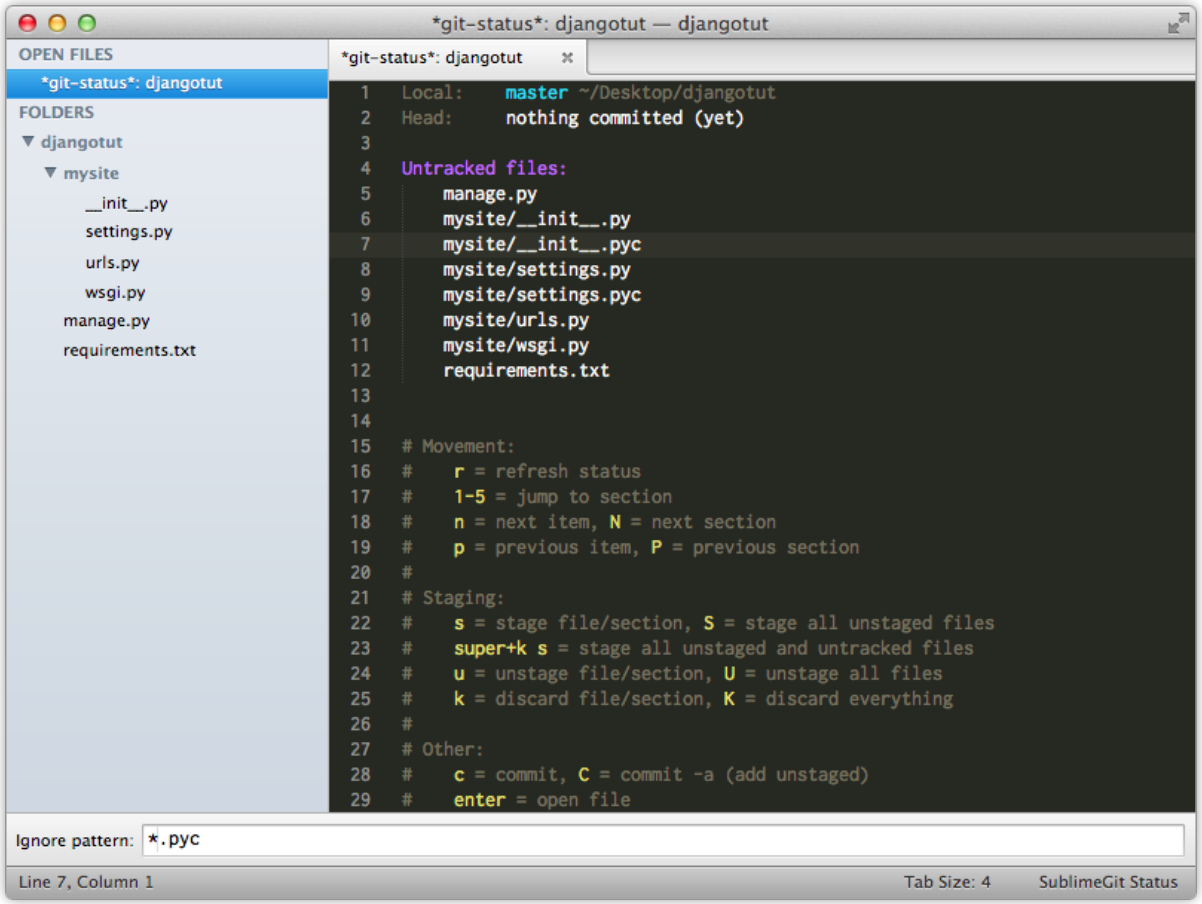
### Ignoring Files

Now, looking at our status view, we notice that we've got those pesky .pyc files. We definitely don't want to add those to git, so let's ignore them.

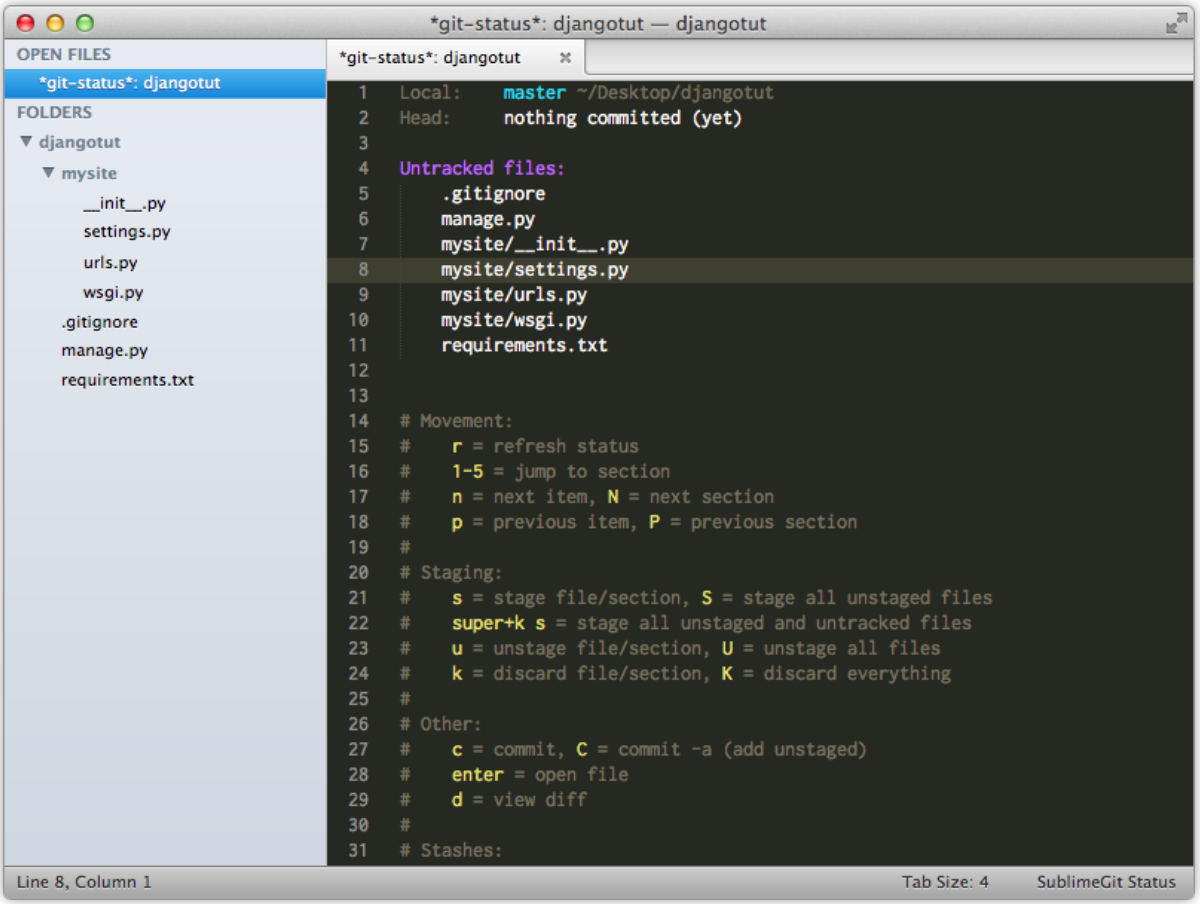
Pressing `i` on a file will add that file to the `.gitignore` for the repository. But we don't want to add just a single .pyc file, we want to add the pattern *.pyc so we don't have to deal with them again. Pressing `I` (capital I) will give you a choice of the pattern to add to the gitignore:




Now, let's change the pattern to *.pyc:




Pressing `enter` will ask you to confirm your choice, and after doing so, we can see that the .pyc files have been removed from the status view, and a gitignore file have been added:




### Adding Files

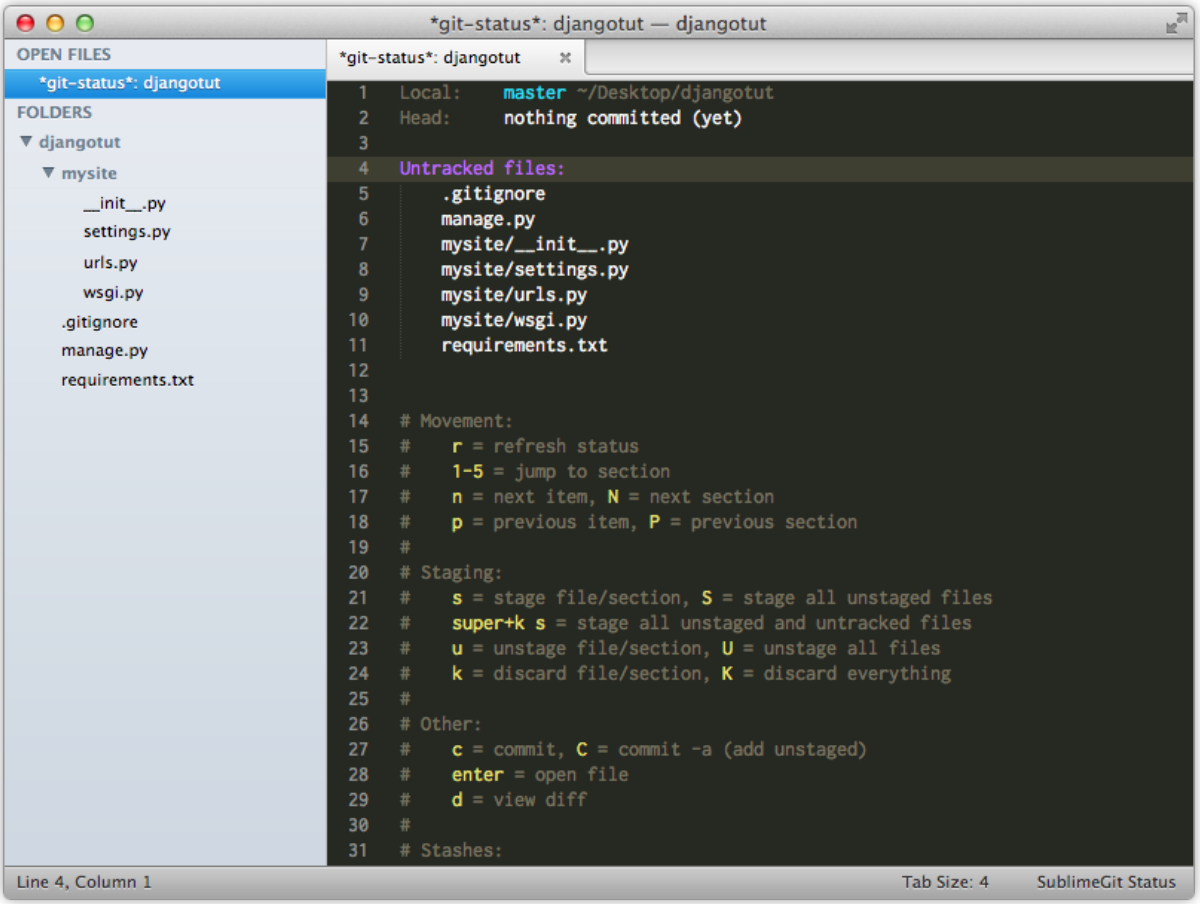
Now, to add the files, there are several different ways to go.

We can press `s` on each file individually, and allow the status window to update between each press. If we don't want to wait for the status window to update, we can also select all of the files we'd like to add, and then press `s`. Or we could use Sublime Text's awesome multiple caret feature and place a caret on every line before pressing `s`. This will add all of the files, since SublimeGit supports multiple selection.

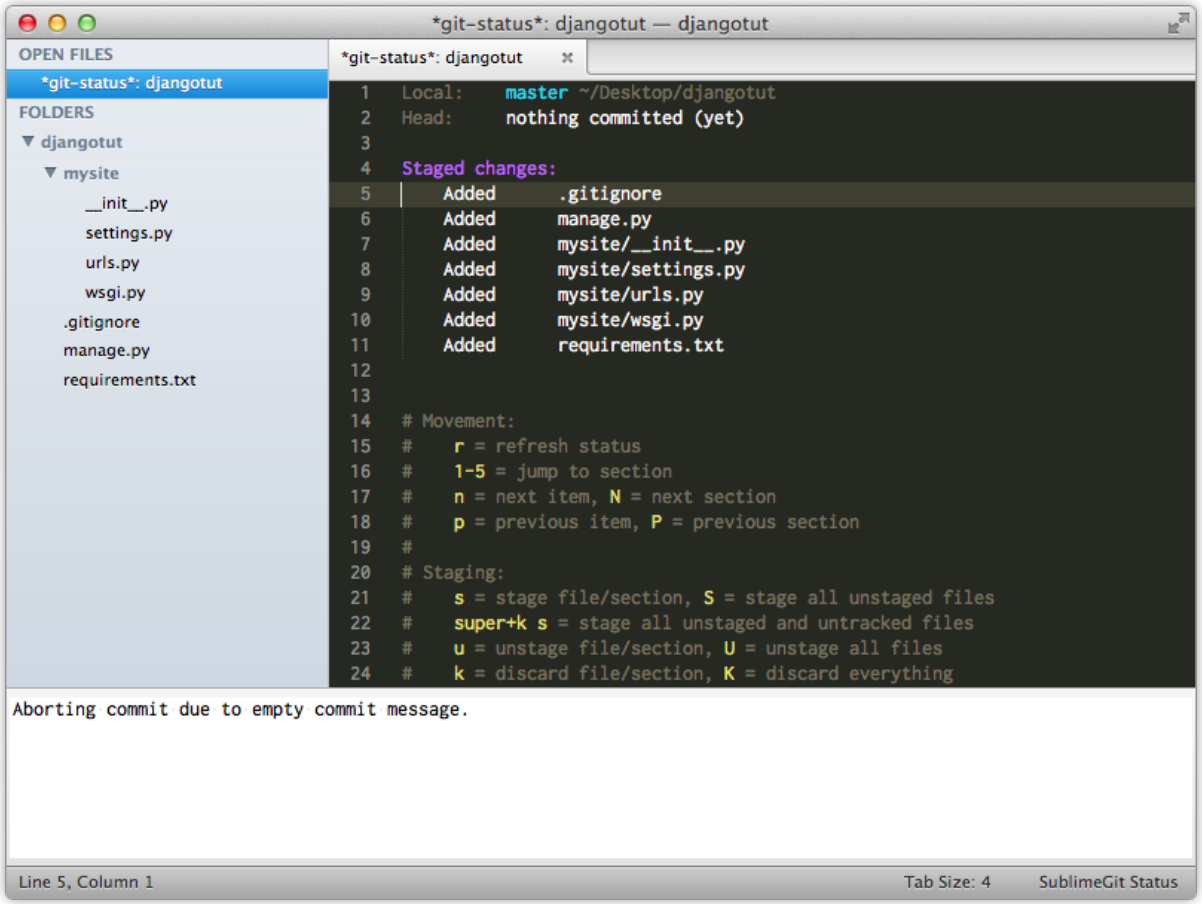
Another way to go would be placing the caret on the section header and pressing `s`.

Finally, we could press `ctrl+shift+s` which will add everything. This command can result in a lot of changes, which is why it's purposefully been made a little hard to type.

Let's go with placing the cursor on the section and pressing `s`:




Pressing `s` moves the files to the **Staged changes** section:



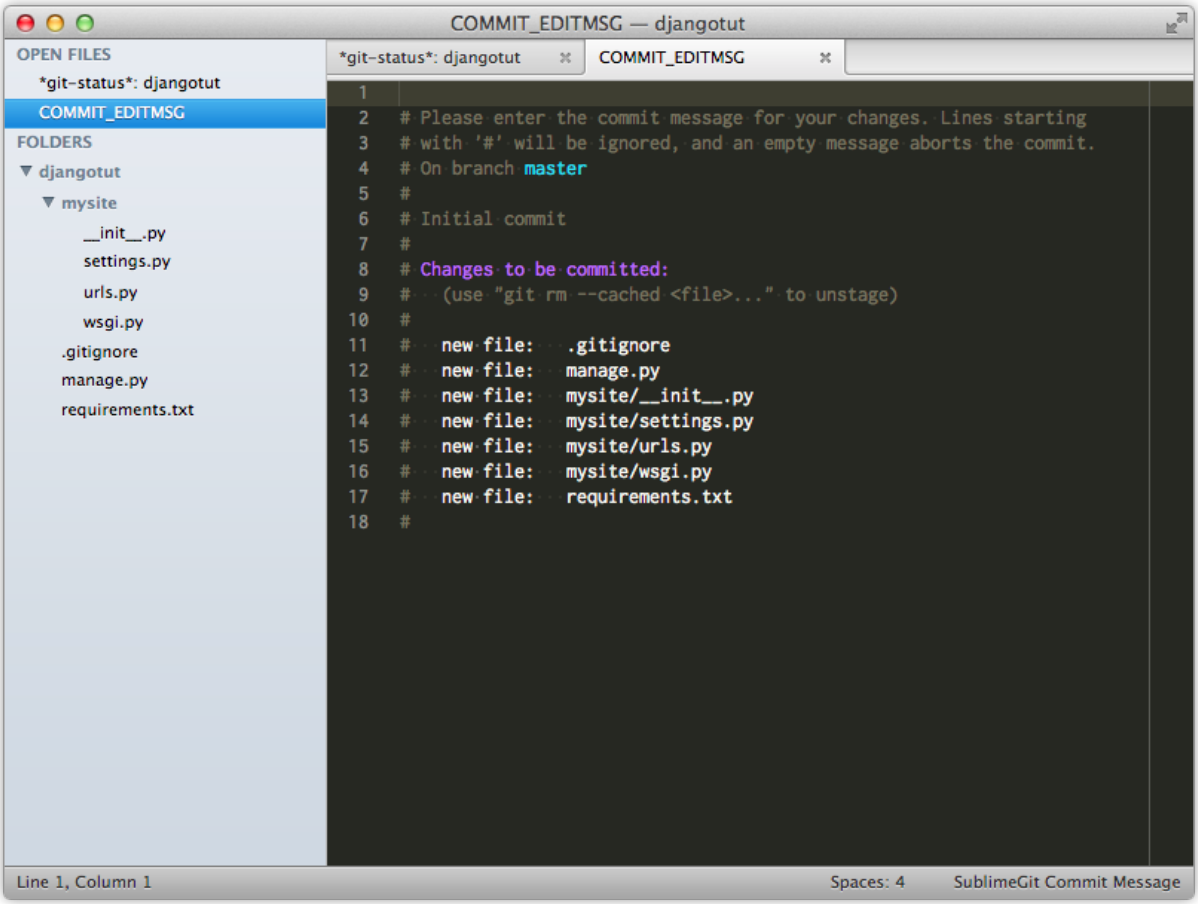

Now we are ready to make the initial commit on our project.

### Other Ways to Add Files

Using the status view isn't the only way to add files in SublimeGit. See *Adding files* in the *Commands Reference* for alternatives.

## Committing

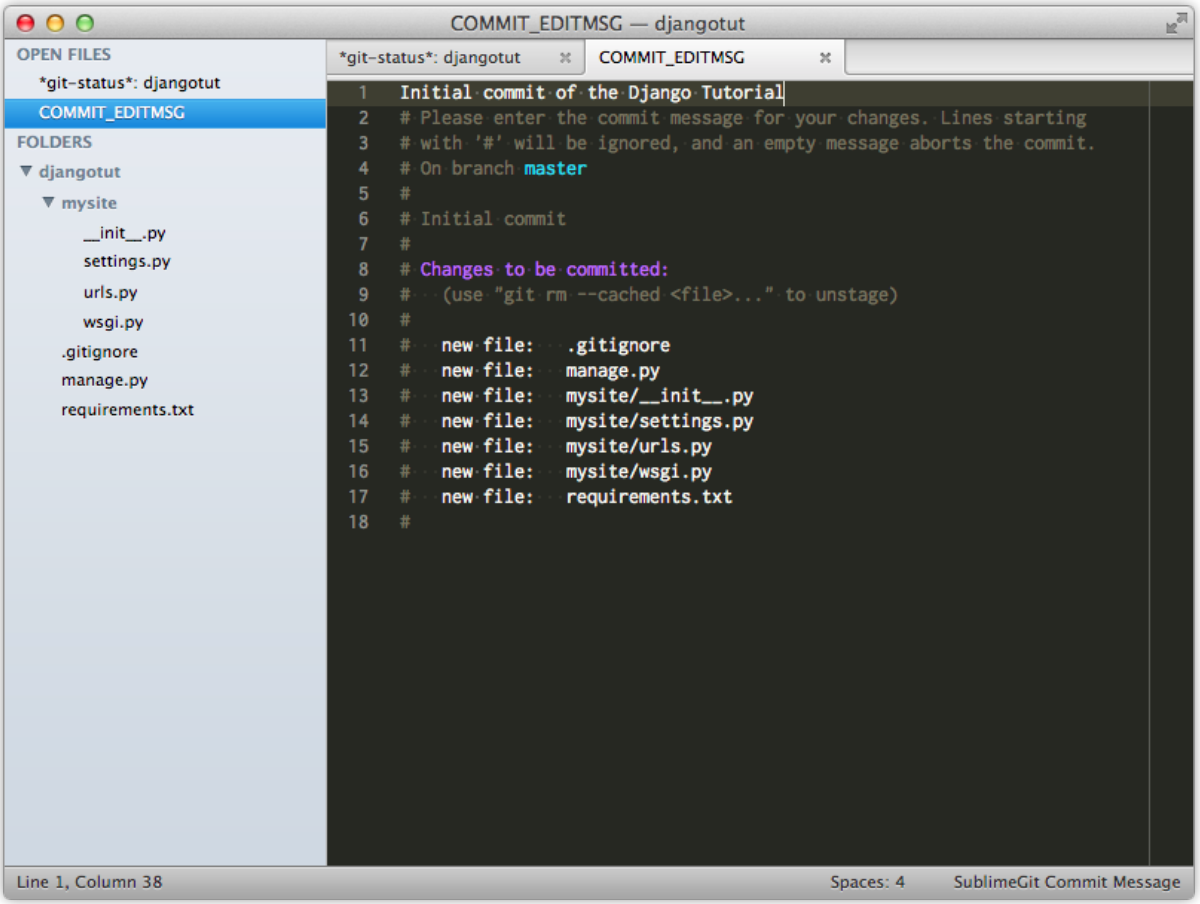
To enter the commit view, press `c` in the status view. This will bring op a view for you to enter a commit message, and place the caret so that you can start typing right away:



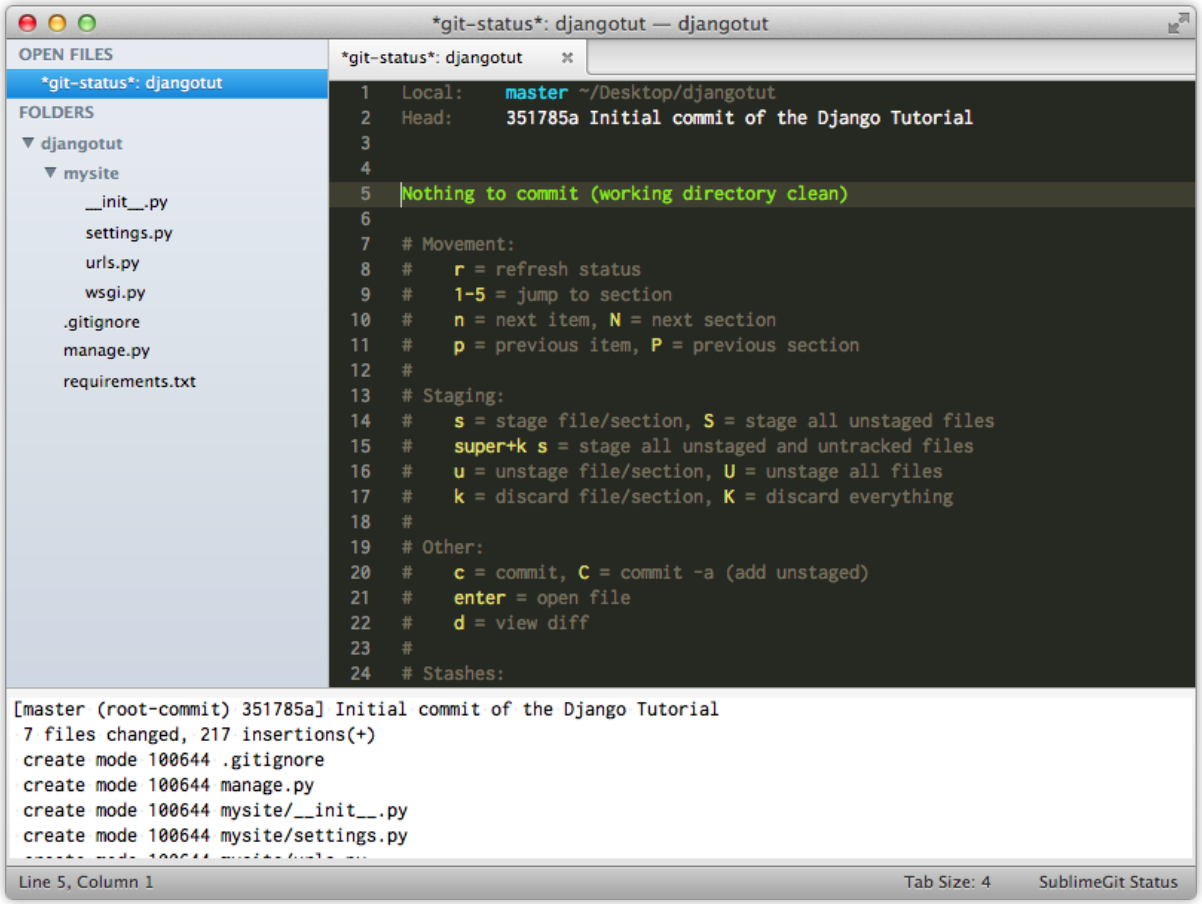

**Note:** This view contains a vertical ruler set at 72 characters. This is to encourage good commit message style, as detailed by Tim Pope and Linus Thorvalds among others.

If you write your commit messages like this, other developers will nod their head in quiet approval, a thousand adorable kittens will be saved, and riches will rain from the sky. Also, you'll get a nice git log, and pretty commit messages on github.

So let's type our commit message:




Once we're done with typing, closing the view will perform the commit and notify us in a console panel:

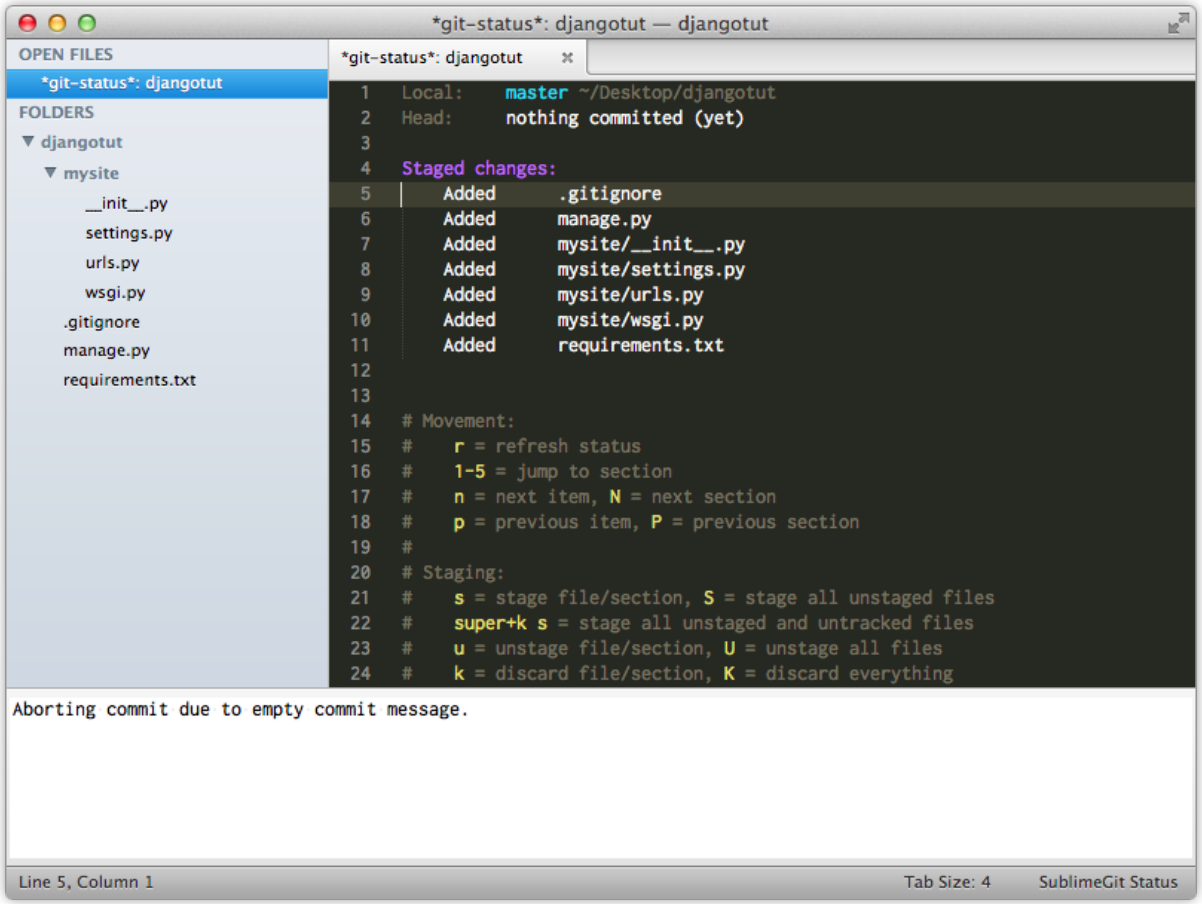



We can now see that our **Head** information has been updated, and that the working directory is clean.

### Aborting a Commit

What if we change our mind halfway through writing the commit message? At that point closing the file would commit a half-finished commit. The solution is simple. Just delete the commit message. This can be done by selecting everything (`cmd+a` on OS X, `ctrl+a` on Linux/Windows) followed by `delete`.

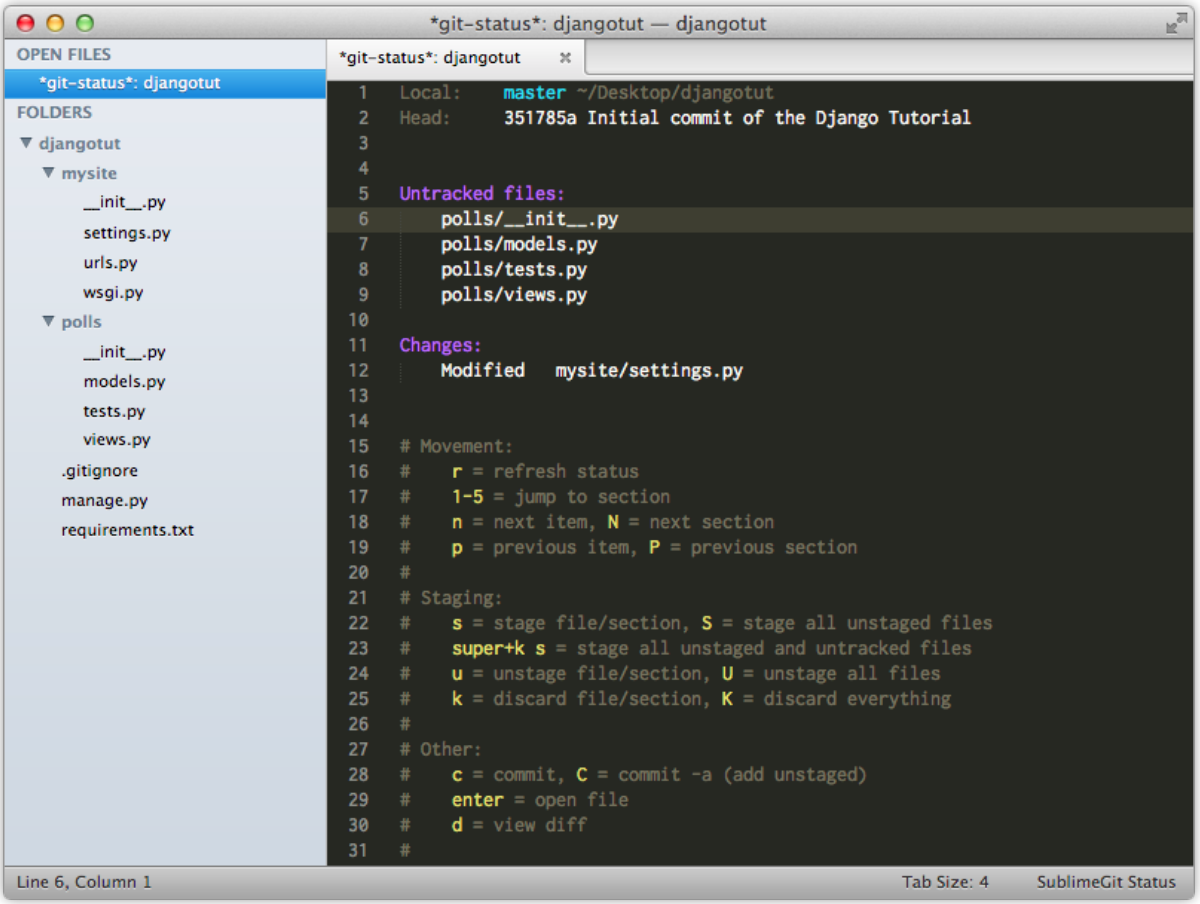
Closing an empty commit view will abort the commit, and let you know:




## Staging Changes

So now that we've made our initial commit, let's make some more changes to the project.

After changing the settings around and adding a polls app, the status view now looks like this:

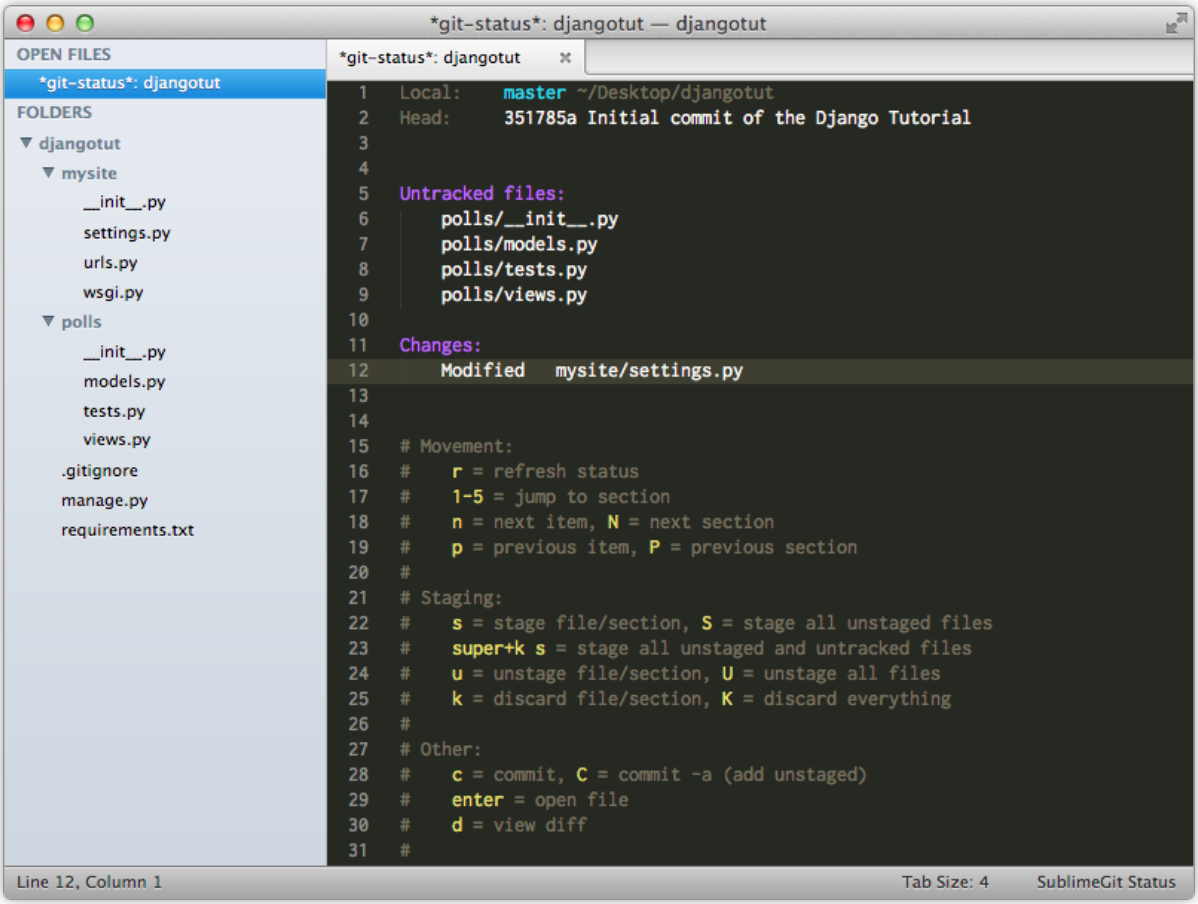



Since we are actually doing two separate things here we might want to split it up into two commits, one containing the changes to settings.py and the other containing our initial polls app.

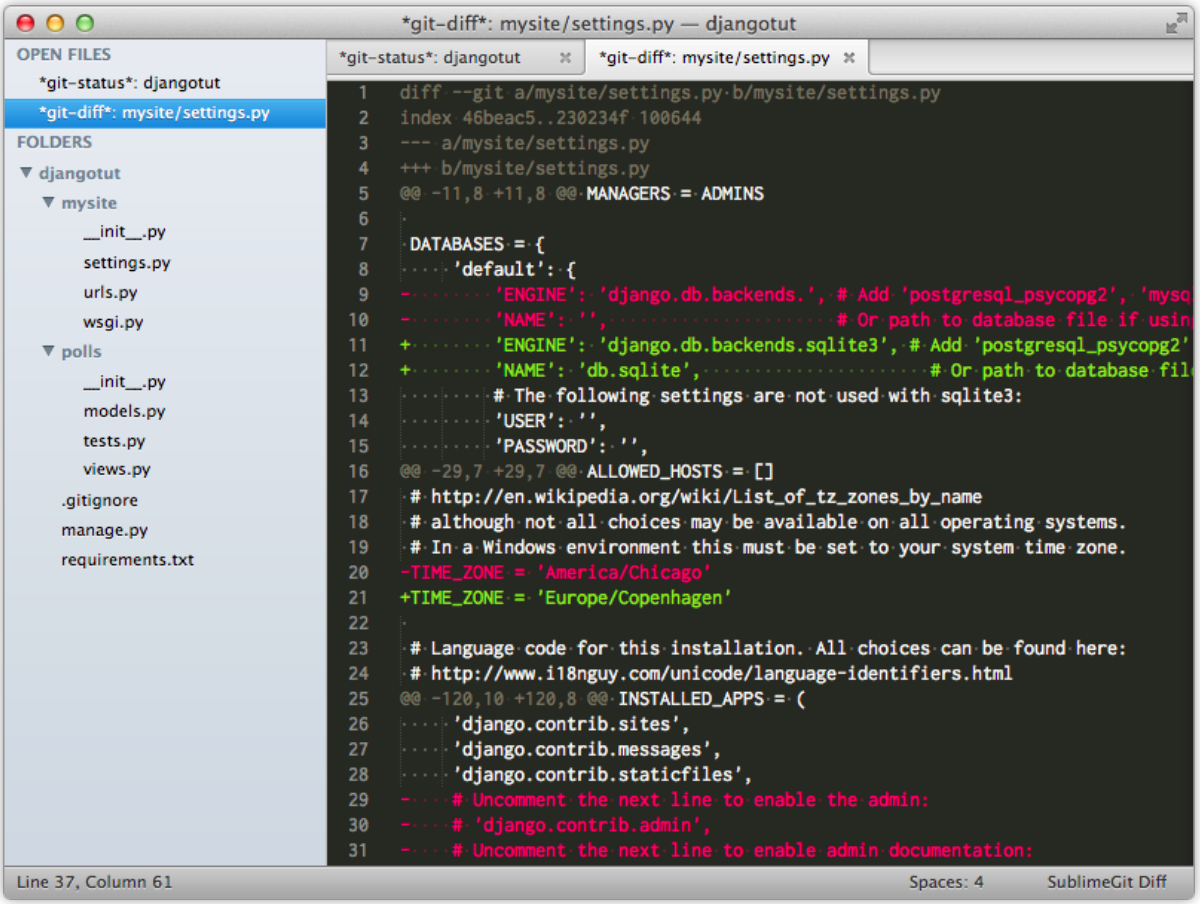
But first, let's take a look at what we've actually changed.

### Viewing Diffs

Navigate to the **Changes** section. A quick way to do this is by pressing `2` to jump to the second section, followed by `n` for next item. Another way would be by pressing `Nn` for next section followed by next file.




Once the caret is over the file, press `d` to open a diff view:

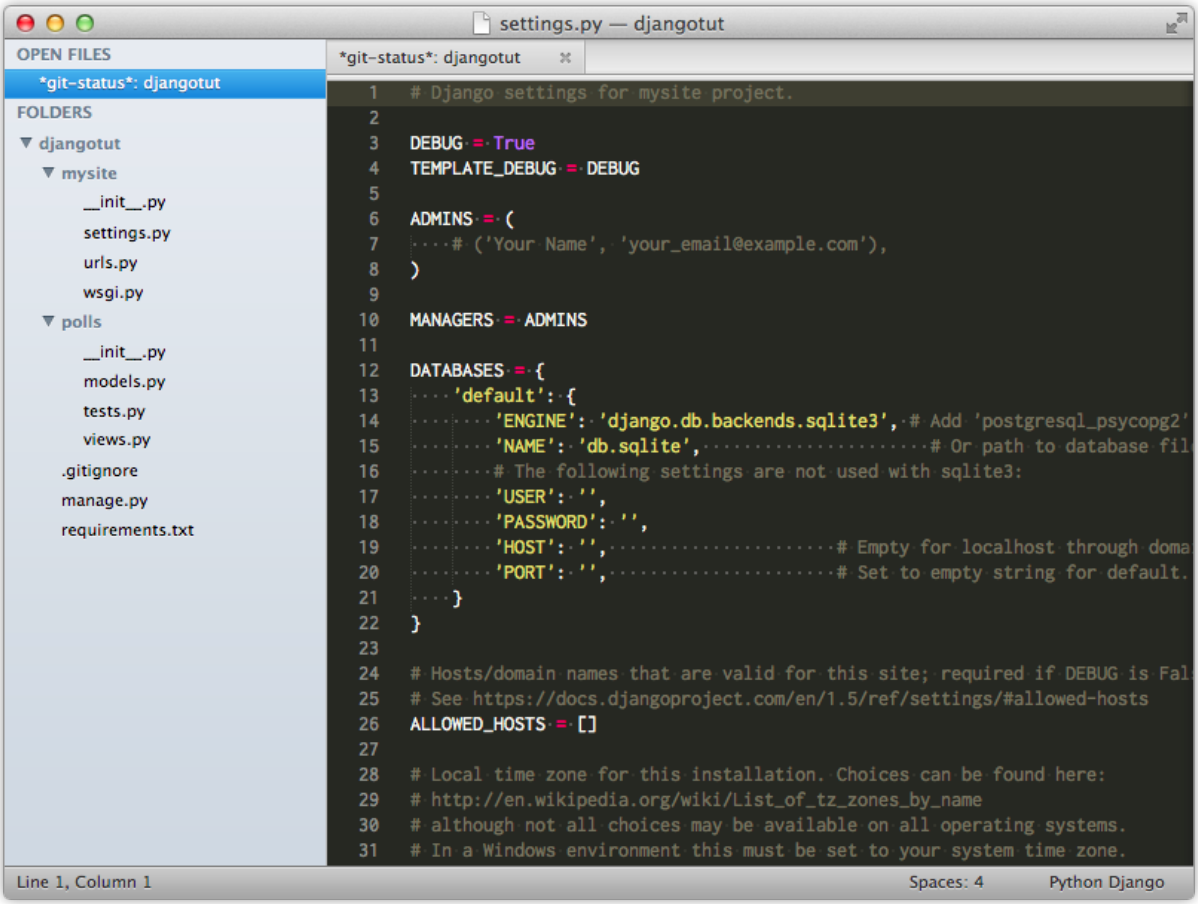



As we can see here, we've changed the database settings, the timezone, as well as enabled the admin application.

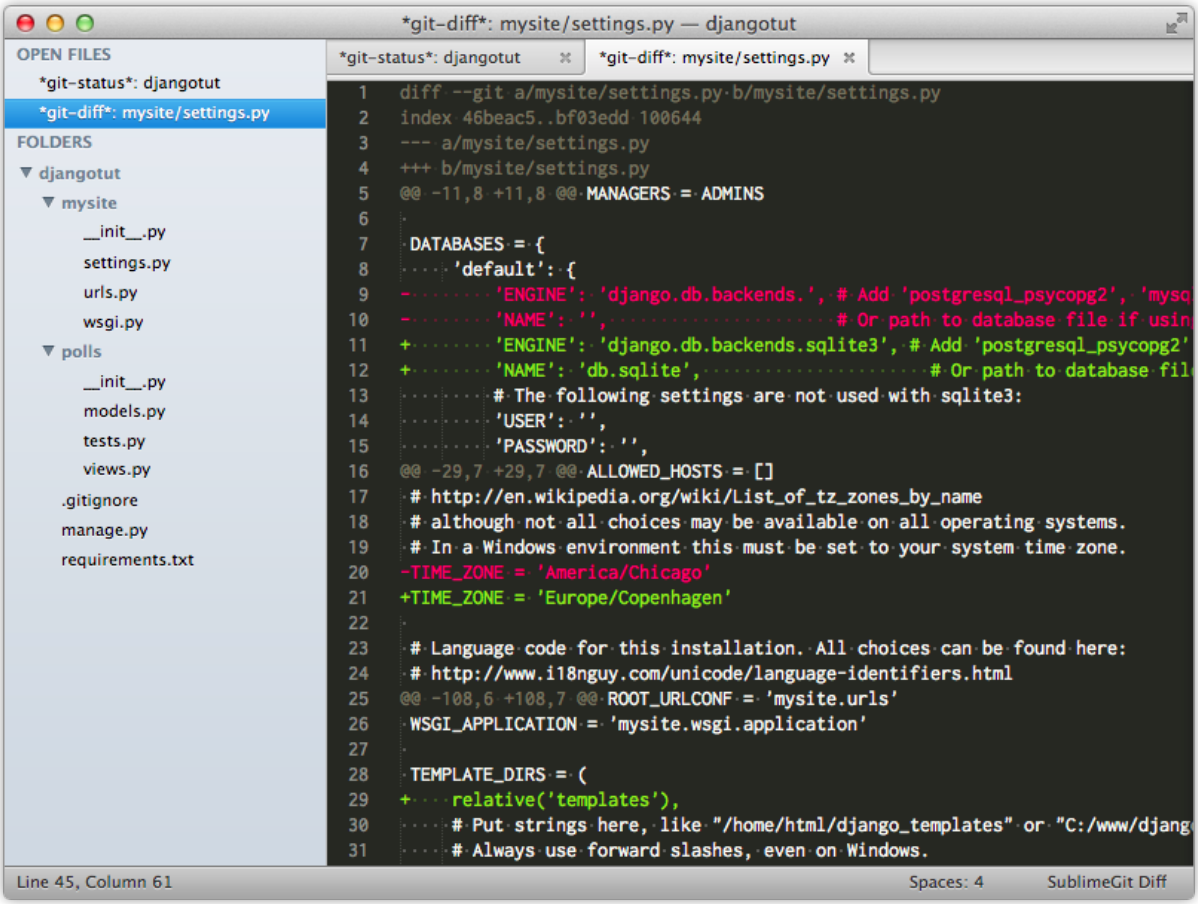
While viewing this diff, we realize that we probably need to add a template directory as well. So let's close the diff view, and open the file to add the template directory.

### Opening a File

Back in the status view, the caret should still be on the settings file. Pressing `enter` will open the file for editing:



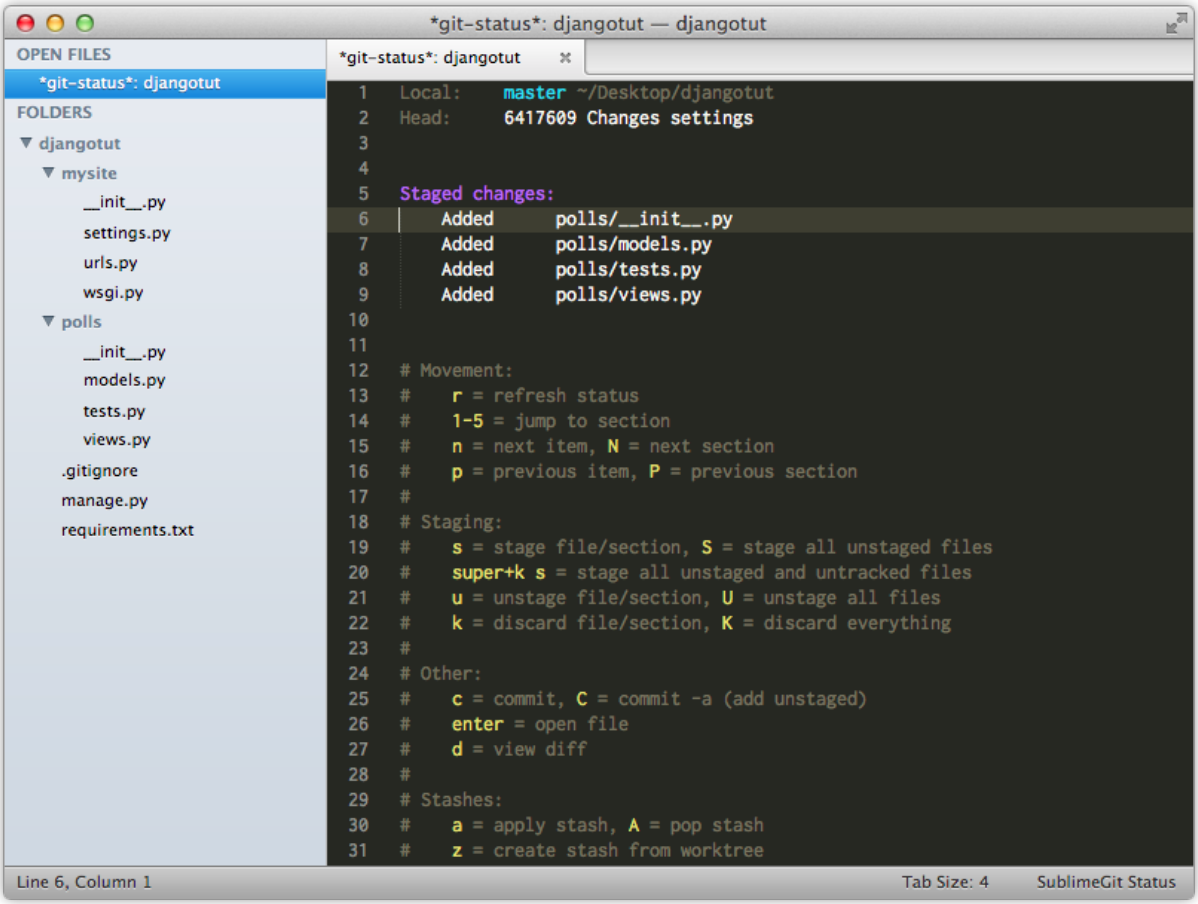

After adding the missing setting, we can view a diff again to see that the change has been picked up:




Now we're ready to commit. To do so, press `s` on the file, followed by `c` to open the commit view.

### Unstaging Files

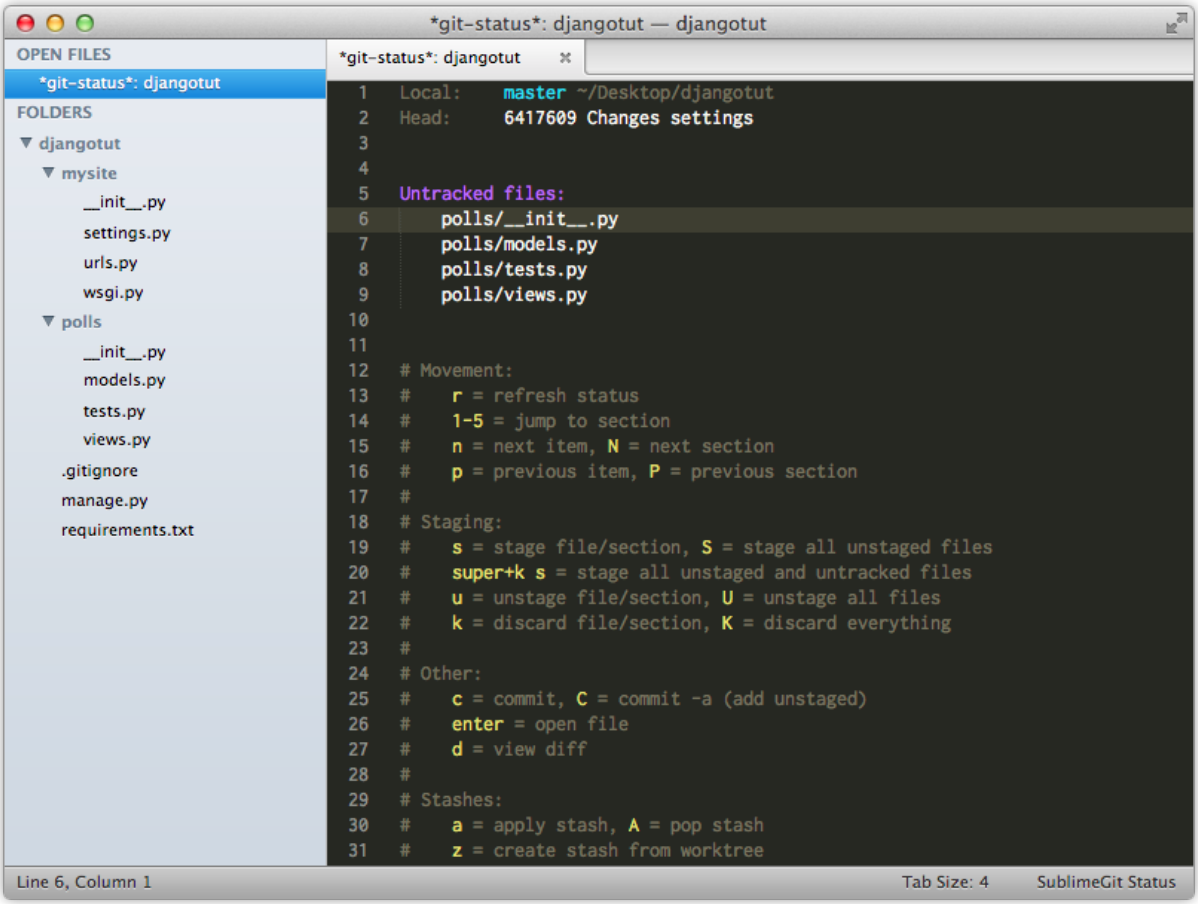
For the second commit, let's start by adding the polls application exactly as before:




But what if we want to do a little bit of work on it before adding it the first time? Now we've already staged it, so we need a way to undo that.

For that purpose, we can use `u` to unstage single files or entire sections, or `U` to unstage everything. This works exactly like the `s`/`S` commands described earlier.

Let's use the `U` command since that will unstage everything at once. Press `U` anywhere in the status view. The files will then jump back to the **Untracked Files** section:

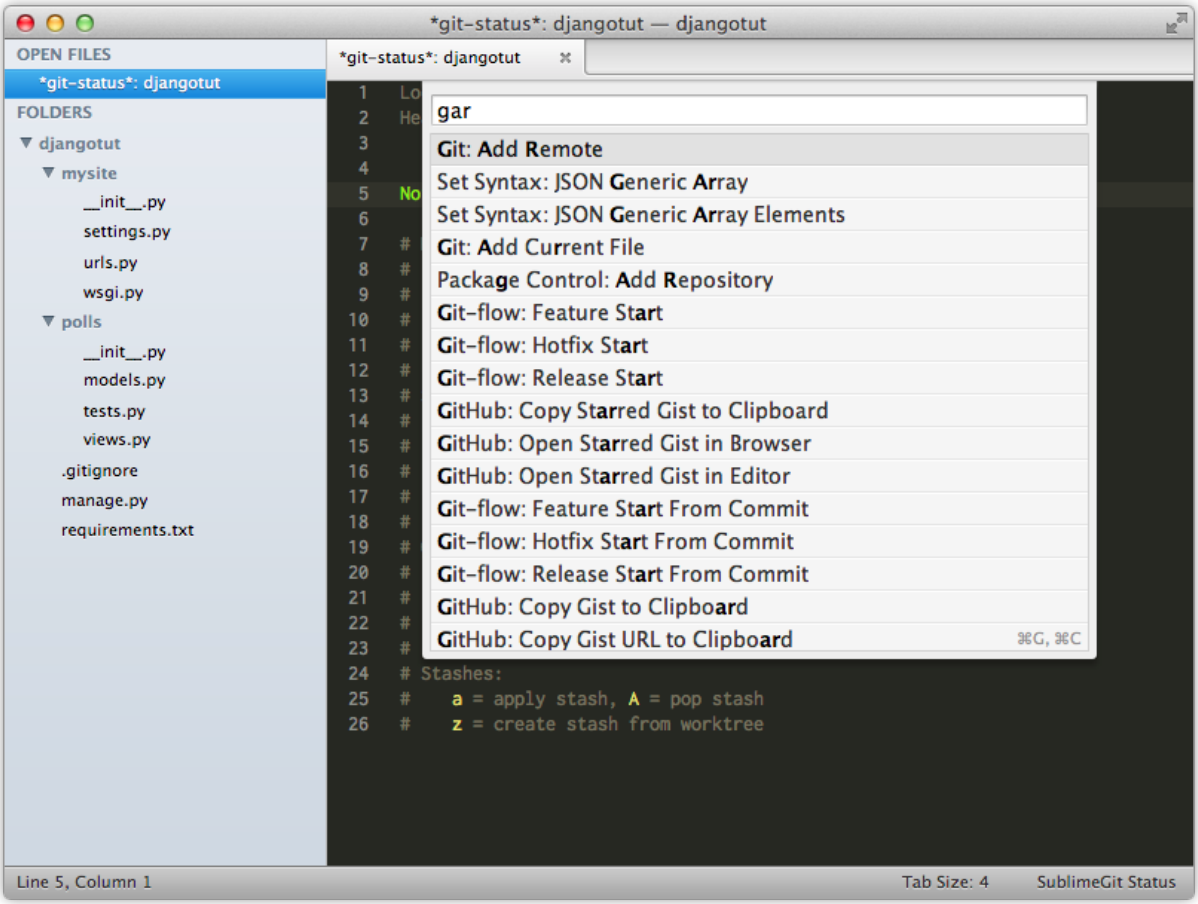



## Sharing Our Project With the World

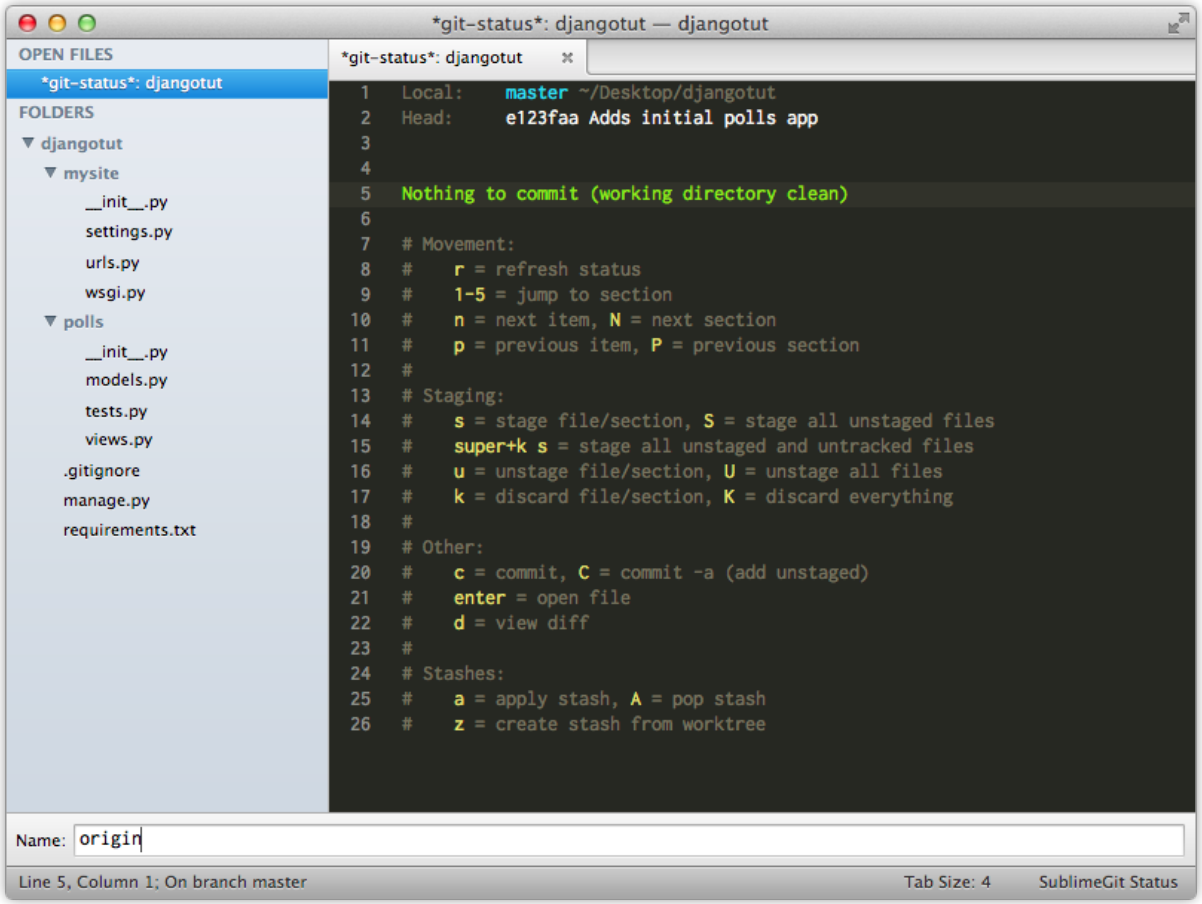
Now that we're getting some traction on our project, we might want to start sharing it with the world. To do that we've created a github (or bitbucket, or google code, or breanstalk, etc.) repository for it. In our case it has the url *git@github.com:SublimeGit/djangotut.git*.

### Adding a Remote

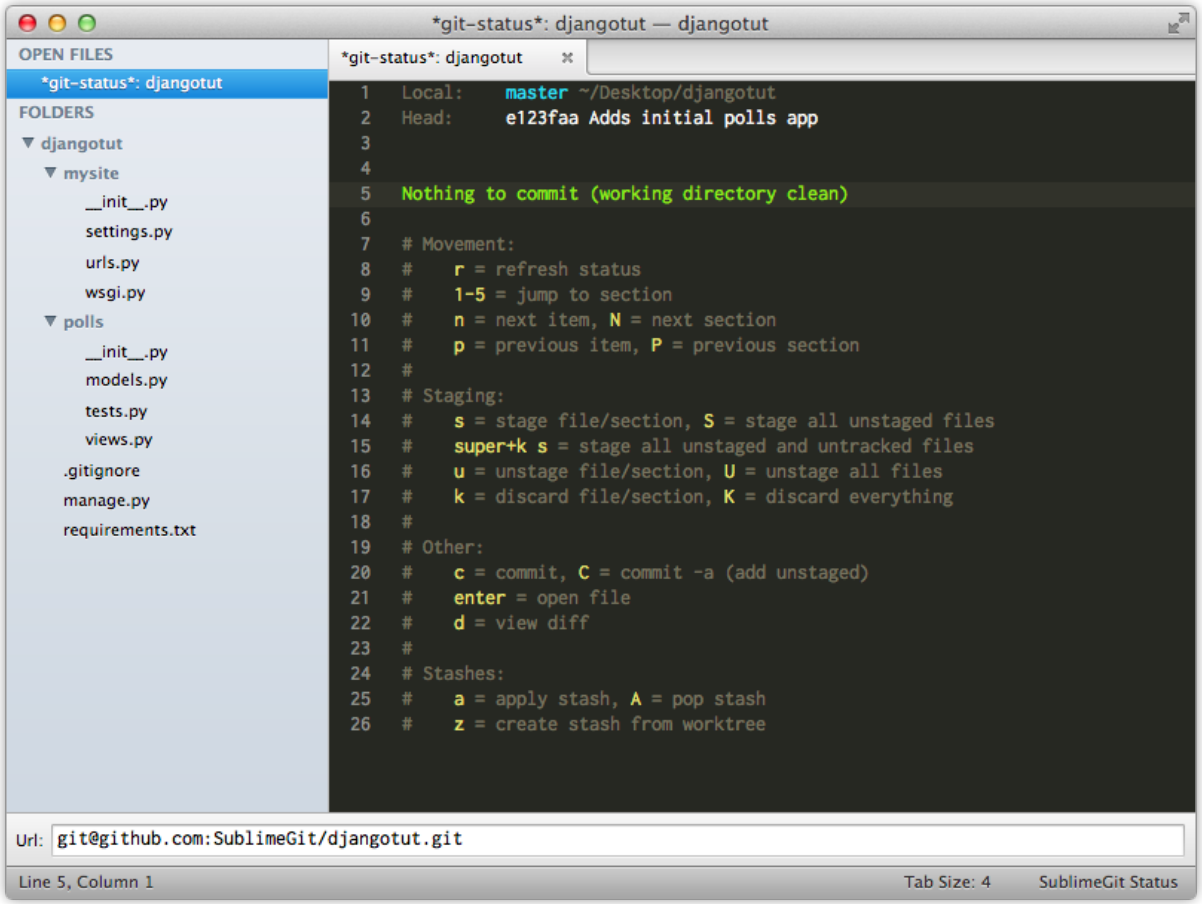
To add this remote, execute the command *Git: Add Remote*, again using Sublime Text's fuzzy matching to avoid typing all of it:




After selecting the command, we will be asked to provide a name for the remote. If this is the first remote we are adding, SublimeGit will assume we want to name it **origin** since that's the convention:




After pressing `enter` to confirm the name, we will be asked to add the url of the remote:




When the url is confirmed, SublimeGit will open the remote management interface. This is the same options you will get if you execute the *Git: Remote* command.

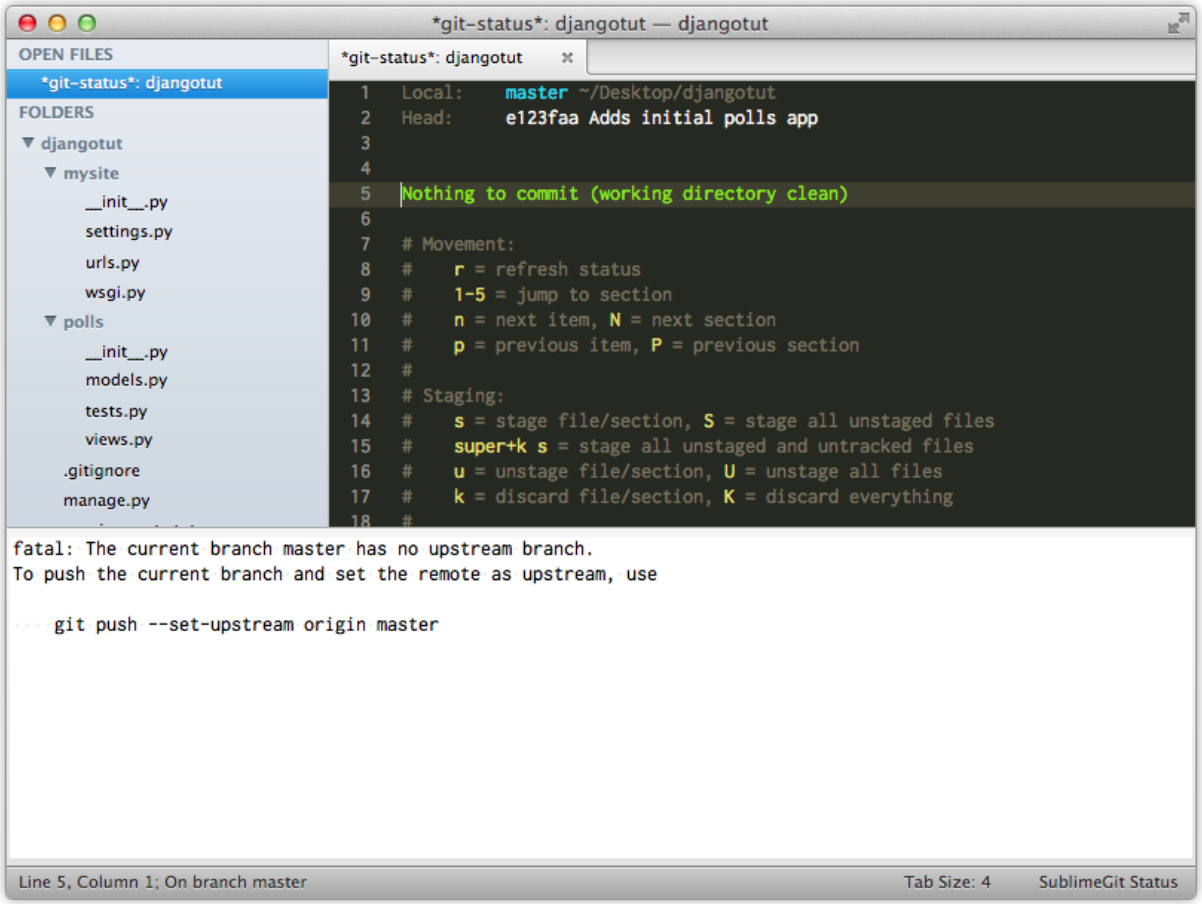
### Pushing

---

**Note:** If you added a remote which wasn't empty (such as when initilized with a README through github), you might need to execute the *Git: Pull* command before pushing.

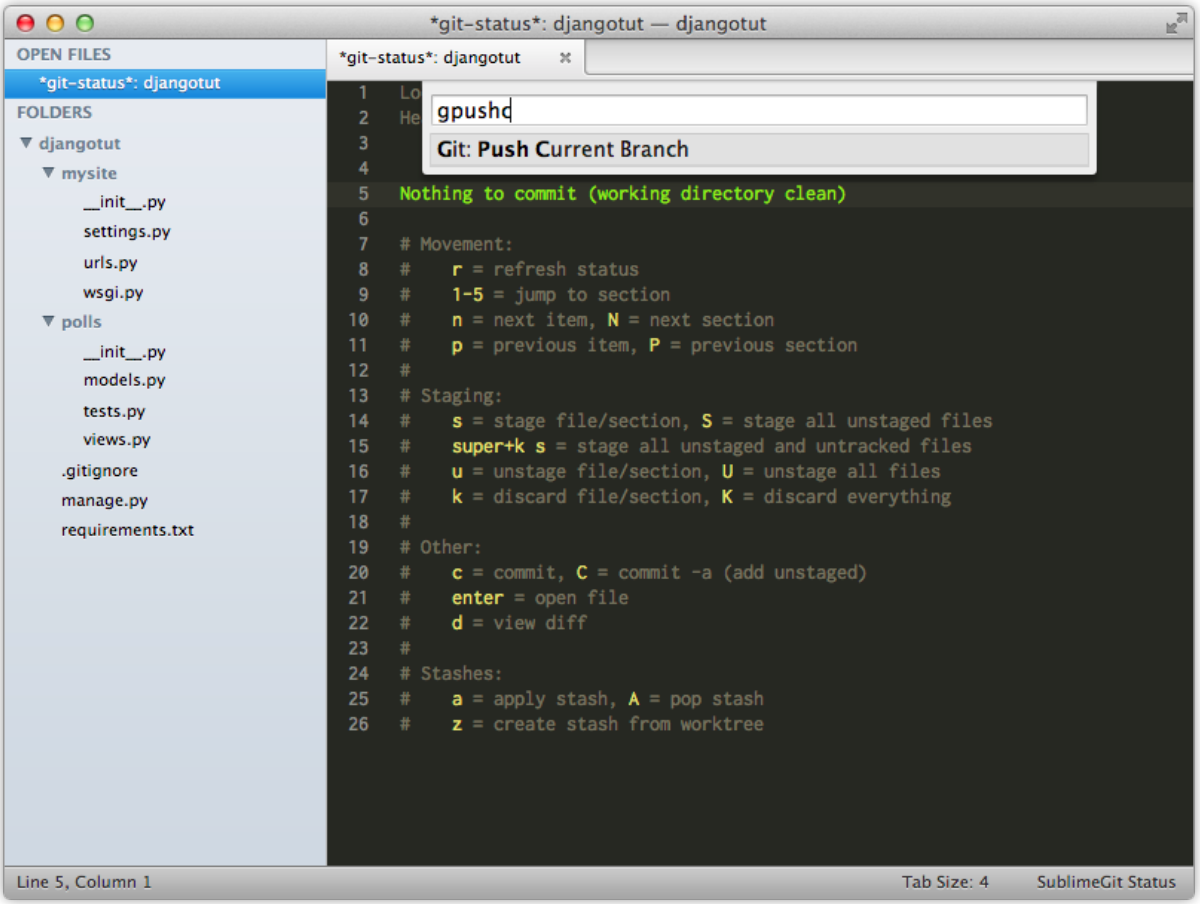
---

Now, if we try to just execute *Git: Push* we might be in for a surprise:

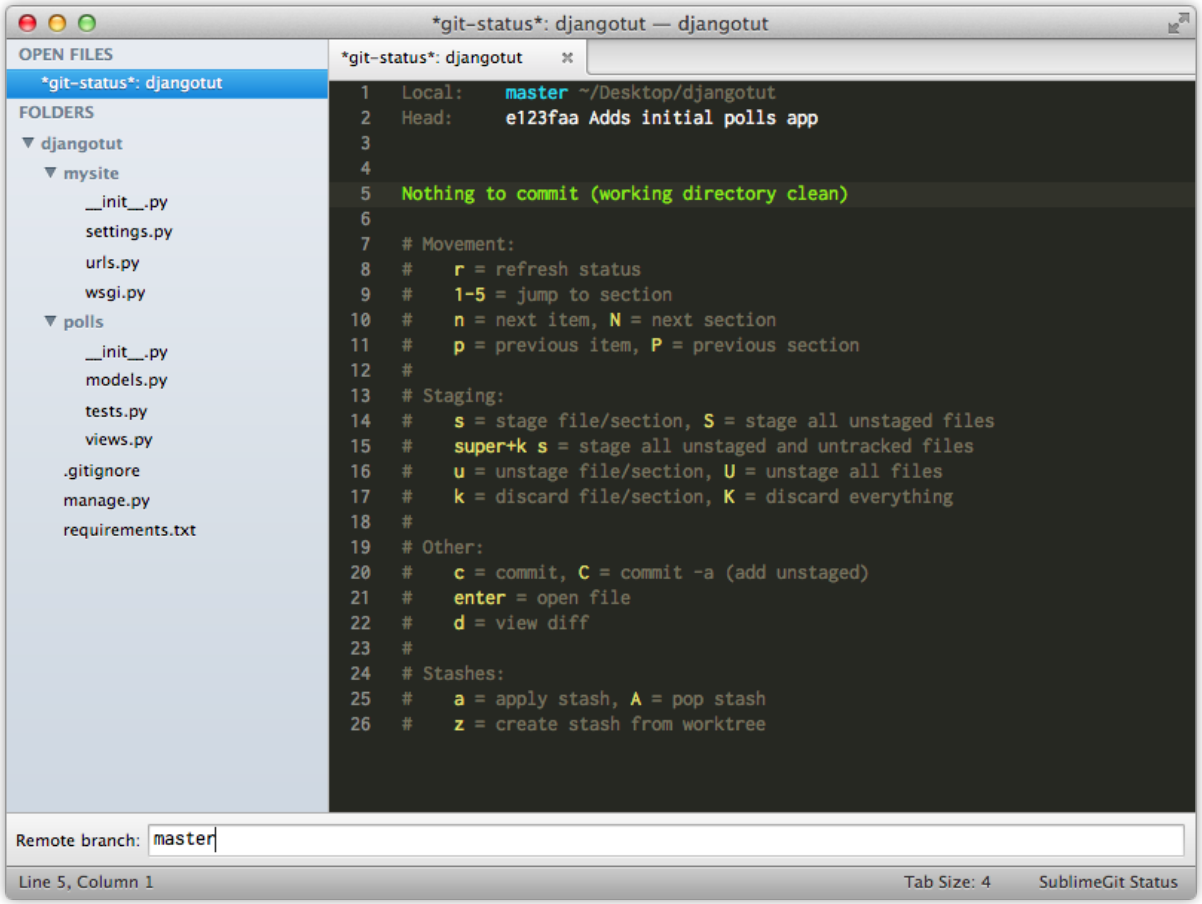



What gives? Well, since we've only just added the remote, without specifying it as the default remote for any of the branches, we need to push a little bit differently the first time around.

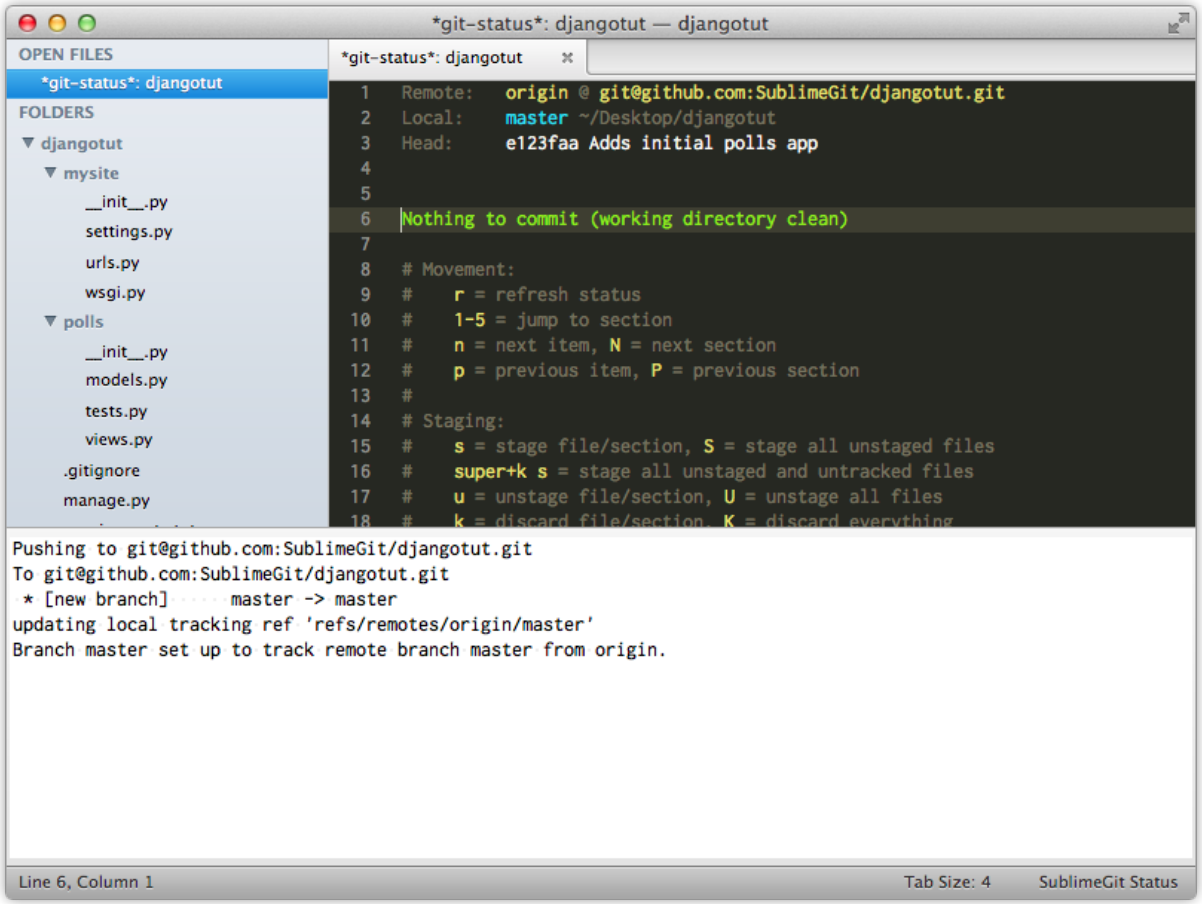
To do this, execute the command *Git: Push Current Branch*:




Then we have to enter the name of the branch on the remote. By default, the current branch name is selected:




After pressing `enter`, SublimeGit will push the branch to the remote, as well as set the necessary configuration to allow using the *Git: Push* and *Git: Pull* commands in the future:




Also notice how a remote section shows up in the status view. This shows the remote url, and the name of the remote.

## Branching

See the section *Branching and Merging* in the *Commands Reference*.

## Stashing

See the section *Stashing* in the *Commands Reference*.

## Finding Help

To find help on a specific git command, you can use the *Git: Help* command, which uses the built-in git html documentation.

### Further Reading

While this tutorial covers the most important parts of SublimeGit, there is a lot more to explore. Take a look at the *Commands Reference* for a list of all the available commands, or have a look at the *Plugins* section for information on how to use the SublimeGit plugins.

# Commands Reference

## Creating and Switching Repositories

**`Git: Init`**

Initializes a git repository in a specified directory.

An input panel will be shown in the bottom of the Sublime Text window, allowing you to edit the directory which will be initialized as a git repository. After choosing the directory, press `enter` to complete. To abort, press `esc`.

If the directory does not already exist, you will be asked if you want to create it. If the path already exists, but it is not a directory, or if it is a directory and already contains git repository, the command will exit with an error message.

---

**Note:** The initial suggestion for the directory is calculated in the following way:

1. The first open folder, if any.
2. The directory name of the currently active file, if any.
3. The directory name of the first open file which has a filename, if any.
4. The user directory of the currently logged in user.

---

**`Git: Switch Repo`**

Switch the active repository for the current Sublime Text window.

In SublimeGit, each window has an active repository. The first time you execute a git command, SublimeGit will try to find out which repository should be the active one for the current window. If there are multiple possible repositories, you will be presented with a list to choose from. Your selection will then be set as the active repository for the window.

If you generally only have one folder open per window in Sublime Text and don't use git submodules, then you probably won't have to switch repositories manually. However, there are some situations where it can be necessary to do so:

**Nested git repositories** If you are using git submodules, or some kind of package manager which uses git checkouts in a subfolder of your project to hold packages (such as Composer for PHP), and you want to explicitly specify that you are working inside the nested repository.

**Multiple folders or files** If you have multiple folders or multiple files, which are managed with git, open in the same Sublime Text window, and you want to switch the repository that you are currently working on.

---

**Note: How does SublimeGit find my repositories?**

Excellent question. SublimeGit will try it's best to guess which repository you are working on. In general it works something like this:

- Find the currently active file.
  - Is it a git view? Use that repository.
  - Is any of the parents a git repository? Use that.
- If that fails, find the currently active window.
  - Find a list of all possible directories:
    - The directories of any open folders.

∗The directories of any open files.

–Generate a list of all of the parents of these directories.

–Check to see if any of the directories or their parents are git repositories.

•Select a repository:

–If there is only one repository then use that.

–If there are more than one repository, present a list to choose from.

---

## Status

**Git: Status**
Documentation coming soon.

**Git: Quick Status**
Show an abbreviated status in the quick bar.

As an alternative to the full status window, a list of changed files is presented the quick bar. Next to each filename there is an abbreviation, denoting the files status.

This status contains 2 characters, X and Y. For paths with merge conflicts, X and Y show the modification states of each side of the merge. For paths that do not have merge conflicts, X shows the status of the index, and Y shows the status of the work tree.

The statuses are as follows:

- **' '** = unmodified
- **M** = modified
- **A** = added
- **D** = deleted
- **R** = renamed
- **C** = copied
- **U** = updated but unmerged
- **?** = untracked

Selecting an entry in the list will bring up a diff view of the file.

## Diffs

**Git: Diff**
Shows a diff of the entire repository in a diff view.

This diff is between the worktree and the index. Thus, these are the changes that you could ask git to add to the next commit.

For diff on a single file, either use the **Git: Quick Status** command, or press `d` when the cursor is on a file in the status view.

**Git: Diff Cached**
Shows the cached diff for the entire repository in a diff view.

The difference between this command and the **Git: Diff** command is that this command shows the difference between the staged changes (the changes in the index), and the HEAD. I.e. these are changes which you could tell git to unstage.

For diff on a single file, either use the **Git: Quick Status** command, or press **d** when the cursor is on a file in the status view.

## Blame

**Git: Blame**

Run git blame on the current file.

This will bring up a new window with the blame information to the left of the file contents, on a per-line basis. Lines which are selected when executing the commands will be marked with a dot in the gutter. When placing the cursor on a line, the summary of the commit will be shown in the status bar.

If the file has not been saved to the filesystem, or the file is not tracked by git, it's not possible to blame, and an error will be shown.

To navigate further into the blame information, a couple of keyboard shortcuts are available:

- `enter`: Show the commit in a new window (like Git: Show).
- `b`: Open a new blame starting at the given commit.

---

**Note:** These keyboard shortcuts support multiple selection, so you can potentially open **a lot** of tabs. If your action will open more than 5 tabs, you will get a warning asking if you want to continue. You can turn this warning off with the **git_blame_warn_multiple_tabs** setting.

---

**Settings**

- **`git_blame_warn_multiple_tabs`** – If set to `true`, SublimeGit will give you a warning when your action from a blame view will open more than 5 tabs. Set to `false` to turn this warning off.

## Adding files

**Git: Quick Add**

Adds one or more files to the staging area by selecting them from the quick bar.

A list of modified files are presented in the quickbar. Each file is marked with a letter, indicating it's status:

- **M** = modified
- **A** = added
- **D** = deleted
- **R** = renamed
- **C** = copied
- **?** = untracked

To add a file from the list, either click the file with the mouse, or use arrow up/arrow down or searching until you have the file you are looking for, and then press `enter`. After adding a file, the status list will update, allowing you to select another file to add. To dismiss the status list, press `esc`.

When there are no more files to add, the status list will show the usual git message for a clean working dir. To dismiss the list press `enter` or `esc`.

There are two special options at the bottom of the status list. To go to them quickly, press arrow up which will select the bottom-most option. These options are:

**+ All unstaged files** This option will add all changes to files git already knows about (all the files not marked with **?**).

**+ All files** This option will add all changes to files git already knows about, as well as all new files (files marked with **?**).

**Git: Add Current File**
This command adds the currently open file to the git staging area. It the –force switch, so the file will be added even if it matches a repository .gitignore pattern, or a global .gitignore pattern.

The file must have already been saved, otherwise it won't exist on the filesystem, and can't be added to git.

If the command completes successfully, no output will be given.

## Checking out files

**Git: Checkout Current File**
Documentation coming soon.

## Committing

**Git: Quick Commit**
Quickly commit changes with a one-line commit message.

If there are any staged changes, only those changes will be added. If there are no staged changes, any changed files that git know about will be added in the commit.

If the working directory is clean, an error will be shown indicating it.

After entering the commit message, press enter to commit, or esc to cancel. An empty commit message will also result in the commit being cancelled.

**Git: Quick Commit Current File**
Documentation coming soon.

**Git: Commit**
Documentation coming soon.

**Git: Amend Commit**
Documentation coming soon.

## Logs

**Git: Log**
Documentation coming soon.

**Git: Quick Log**
Documentation coming soon.

**Git: Quick Log Current File**
Documentation coming soon.

**Git: Show**
Documentation coming soon.

## Branching and Merging

**Git: Checkout**
Check out an existing branch.

This command allows you to select a branch from the quick bar to check out. The currently active branch (if any) is marked with an asterisk (*) to the left of its name.

**Git: Checkout Commit**
Check out a specific commit.

This command allows you to check out a specific commit. The list of commits will be presented in the quick bar, containing the first line of the commit message, the abbreviated sha1, as well as a relative and absolute date in the local timezone.

After checkout, you will be in a detached head state.

**Git: Checkout New Branch**
Create a new branch from the current HEAD and switch to it.

This command will show an input panel allowing you to name your new branch. After giving the branch a name, pressing enter will create the new branch and check it out. Pressing esc will cancel.

If a branch with the given name already exists, you will be asked if you want to overwrite the branch. Selecting cancel will exit silently, without making any changes.

**Git: Merge**
Documentation coming soon.

## Working with Remotes

**Git: Add Remote**
Add a named git remote at a given URL

You will be asked to provide the name and url of the remote (see below). Press `enter` to select the value. If you want to cancel, press `esc`.

After completion, the Git: Remote command will be run, to allow for further management of remotes.

**Name:** The name of the remote. By convention, the name *origin* is used for the "main" remote. Therefore, if your repository does not have any remotes, the initial suggestion for the name will be *origin*.

**Url:** The git url of the remote repository, in any format that git understands.

**Git: Remote**
Manage git remotes

Presents s list of remotes, including their push and pull urls. Select the remote to perform an action on it. After an action has been performed, the list will show up again to allow for further editing of remotes. To cancel, press `esc`.

Available actions:

**Show** Show information about the remote. This includes the push and pull urls, the current HEAD, the branches tracked, and the local branches which are set up for push and pull.

The result will be displayed in a panel in the bottom of the Sublime Text window.

**Rename** Rename the selected remote. An input field will appear allowing you to write a new name for the remote. If a new name is not provided, or `esc` is pressed, the action will be aborted.

**Remove** Remove the selected remote. All remote-tracking branches, and configuration for the remote is removed. You will be asked for confirmation before removing the remote.

**Set URL** Change the URL for the selected remote. An input fiels will appear allowing you to specify a new URL. The given URL will be used for both the push and pull URL. If a new URL isn't specified, or `esc` is pressed, the URL will not be updated.

**Prune** Delete all stale remote-tracking branches for the selected remote. Any remote-tracking branches in the local repository which are no longer in the remote repository will be removed.

## Fetching and Pulling

**`Git: Fetch`**
Fetches git objects from the remote repository

If there is only one remote configured, this remove will be used for fetching. If there are multiple remotes, you will be asked to select the remote to fetch from.

**`Git: Pull`**
Documentation coming soon.

**`Git: Pull Current Branch`**
Documentation coming soon.

## Pushing

**`Git: Push`**
Documentation coming soon.

**`Git: Push Current Branch`**
Push the current branch to a remote

This is the command to use if you are pushing a branch to a remote for the first time, or to a different remote than the configured upstream. Will push the current branch to a specified branch on the selected remote, creating the remote branch if it doesn't already exist.

If there is only one remote configured, that will be used, otherwise you will be asked to select a remote. If there are no remotes, you will be asked to add one.

You will be asked to supply a name to use for the branch on the remote. By default, the current branch name will be suggested.

> **Warning:** Trying to push when in a detached head state will give an error message. This is not generally something you want to do.

---

**Note:** This command shares a lot of similarities with the excellent git-publish command, which can be found at https://github.com/gavinbeatty/git-publish.

---

## Stashing

**Git: Stash**
Documentation coming soon.

**Git: Pop Stash**
Documentation coming soon.

**Git: Apply Stash**
Documentation coming soon.

**Git: Snapshot**
Documentation coming soon.

## Tags

**Git: Tag**
Documentation coming soon.

**Git: Add Tag**
Documentation coming soon.

## Custom Commands

**Git: Custom Command**
Execute a custom git command.

By default, this command will be run synchronously, and the output will be presented in a new view, with a title corresponding to the command.

However, it's also possible to use this command to build your own SublimeGit commands.

It takes 3 arguments:

- **cmd**: The command to execute (without the initial "git")
- **async**: `true` to run asynchronously, `false` otherwise. Default: `false`
- **output**: `"view"` for a new buffer, `"panel"` for an output panel, `null` for no output. Default: `"view"`
- **syntax: If output is set to `"view"`, the new buffer will get this syntax file. Should be a name along the** lines of `Packages/Python/Python.tmLanguage`. To see the current syntax for a view, execute `view.settings().get('syntax')` from the console.

---

**Note:** See *Custom Commands* for more information on how to create your own SublimeGit commands.

---

## Browsing Documentation

**Git: Help**
Search through installed Git documentation.

Every standard install of git contains a full set of manual pages in both text and html formats. This commands presents a list of available documentation in a quick bar to allow for easy access.

When a document has been selected, a webbrowser will be opened to show the help file. To abort the list without opening the document, press `esc`.

**Settings**

- **git_help_fancy_list** – If set to `true`, SublimeGit will try to parse the help document to show a nicer list containing a small excerpt from each document. This has a small performance cost the first time the list is generated. Set to `false` to fall back to simple format. Default: `true`
- **git_help_html_path** – If set to a directory, SublimeGit will look in the given directory for git help files. Set to `null` to make SublimeGit auto-detect the location of the help files.

---

**Note:** To find the location the installed documentation, you can execute:

```
$ git --html-path
/usr/local/Cellar/git/1.7.11.3/share/doc/git-doc
```

---

**Git: Version**
Shows the version of git which is installed

This corresponds to running:

```
$ git --version
git version 1.7.11.3
```

### SublimeGit

**SublimeGit: Version**
Show the currently installed version of SublimeGit.

**SublimeGit: Documentation**
Open a webbrowser to the online SublimeGit documentation.

### Gitk

**Gitk**
Documentation coming soon.

## Plugins

SublimeGit comes with plugins for Le-git and git-flow.

### Le-git

#### Branches

**Legit: Switch**
Documentation coming soon.

**Legit: Branches**
Documentation coming soon.

**Legit: Sprout**
Documentation coming soon.

**Legit: Harvest**
Documentation coming soon.

**Legit: Graft**
Documentation coming soon.

### Remotes

**Legit: Sync**
Documentation coming soon.

**Legit: Publish**
Documentation coming soon.

**Legit: Unpublish**
Documentation coming soon.

## Gitflow

### Features

**Git-flow: Feature Start**
Documentation coming soon.

**Git-flow: Feature Finish**
Documentation coming soon.

### Releases

**Git-flow: Release Start**
Documentation coming soon.

**Git-flow: Release Finish**
Documentation coming soon.

### Hotfixes

**Git-flow: Hotfix Start**
Documentation coming soon.

**Git-flow: Hotfix Finish**
Documentation coming soon.

# Keyboard Shortcuts

## Status View

### Movement

- `r`: Refresh status
- `1-5`: Jump to section
- `n`: Next item
- `N`: Next section
- `p`: Previous item
- `P`: Previous section

### Staging

- `s`: Stage file/section
- `S`: Stage all unstaged files
- `ctrl+shift+s`: Stage all unstaged and untracked files
- `u`: Unstage file/section
- `U`: Unstage all files
- `backspace`: Discard file/section
- `shift+backspace`: Discard everything

### Committing

- `c`: Commit
- `C`: Commit -a (add unstaged)
- `ctrl+shift+c`: Commit –amend (amend previous commit)
- `enter`: Open file
- `d`: View diff

### Stashes

- `a`: Apply stash
- `A`: Pop stash
- `z`: Create stash from worktree

## Blame View

- `enter`: Show the selected commit(s)
- `b`: Run blame starting from the selected commit(s)

### Diff View

#### Movement

- `n`: Next hunk
- `N`: Next file
- `p`: Previous hunk
- `P`: Previous file

#### Context

- `+`: Increase hunk context
- `-`: Decrease hunk context

#### Staging

- `s`: Stage hunk
- `u`: Unstage hunk

## Customizations

The defaults of SublimeGit are not for everyone. Here is a list of common customizations which you might or might not be right for you.

### Custom Commands

By using the Git: Custom Command action in SublimeGit, you can create your own SublimeGit aliases. If you have a command that you run often, you can save it with an alias and get access to it in the Sublime Text quick bar.

To do this, we need to first cover how to set up custom commands in Sublime Text. Inside your packages directory (Go to `Preferences > Browse Packages`) there will be a directory called `User`. Inside this directory, you can place files with the extension `.sublime-commands` and they will be picked up by Sublime Text. In the following we're only going to present a short example of how to use the **Git: Custom Command** to extend SublimeGit, but there are more fun things that can be done. For an overview of the format of these files, see the Sublime Text Docs on Command Files.

Now, create a file in the `User` directory and name it `Git.sublime-commands`. Add this to it:

```
[
    {
        "caption": "Git: Graph Log",
        "command": "git_custom",
        "args": {
            "cmd": "log --graph --oneline",
            "output": "panel",
            "async": false
        }
    },
    {
```

```
        "caption": "Git: Diff Master",
        "command": "git_custom",
        "args": {
            "cmd": "diff master",
            "output": "view",
            "async": true,
            "syntax": "Packages/SublimeGit/syntax/SublimeGit Diff.tmLanguage"
        }
    }
]
```

This tells Sublime Text that you want a command named "Git: Graph Log", and when it is executed, Sublime Text should run the command `git_custom` from SublimeGit, which should in turn execute `git log --graph --oneline` synchronously and present the output to you in a new panel. Same goes for the "Git: Diff Master" command, except the command will be asynchronous, the output will be in a view, and the view will have the specified syntax file.

As you can see, the custom commands can take different arguments. Please see *Custom Commands* for possible values of these arguments.

## Keyboard Shortcuts

For information on keybindings in general, please see the Sublime Text Docs.

### Run a Command (e.g. Git: Status)

If you want to figure out what a command is called, you can set Sublime Text to log all commands by executing the following snippet in the console:

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

After you've done that, all commands will then be logged to the console. Using this, you can see that the **Git: Status** command is called `git_status`.

With this information, you can add something like this to your keymap, to open git status when pressing `ctrl+alt+g`:

```
{ "keys": ["ctrl+alt+g"], "command": "git_status"}
```

---

**Note:** You can turn off the command logging again with:

```
sublime.log_commands(False)
```

---

### Add a Key Binding to a Command in the Status View

Let's say you want to have *t* add a tag from the status view. Naturally you don't want this shortcut to be available everywhere (that would make it quite hard to write anything). The solution for this is specifying that the shortcut should only be available in the status view, like so:

```
{ "keys": ["t"], "command": "git_add_tag",
    "context": [{ "key": "selector", "operator": "equal", "operand": "text.git-status
↪" }]
}
```

### Jump to a Specific Section in the Status View

It is possible to jump to a specific section in the git status view, with a set of shortcuts like this:

```
// Section shortcuts
{ "keys": ["ctrl+1"], "command": "git_status_move", "args": {"goto": "section:stashes
↪"},
    "context": [
        { "key": "selector", "operator": "equal", "operand": "text.git-status" }
    ]
},
{ "keys": ["ctrl+2"], "command": "git_status_move", "args": {"goto":
↪"section:untracked_files"},
    "context": [
        { "key": "selector", "operator": "equal", "operand": "text.git-status" }
    ]
},
{ "keys": ["ctrl+3"], "command": "git_status_move", "args": {"goto":
↪"section:unstaged_changes"},
    "context": [
        { "key": "selector", "operator": "equal", "operand": "text.git-status" }
    ]
},
{ "keys": ["ctrl+4"], "command": "git_status_move", "args": {"goto": "section:staged_
↪changes"},
    "context": [
        { "key": "selector", "operator": "equal", "operand": "text.git-status" }
    ]
},
{ "keys": ["ctrl+5"], "command": "git_status_move", "args": {"goto":
↪"section:unpushed_commits"},
    "context": [
        { "key": "selector", "operator": "equal", "operand": "text.git-status" }
    ]
}
```

**Warning:** These shortcuts will overwrite the "focus group" shortcuts built into Sublime Text.

## Color Scheme

SublimeGit uses a lot of different colors. Though great care has been taken in picking the SublimeGit colors to generally look good in the default Sublime Text themes, you might want to customize them.

### Setting a Different Color Scheme

If you want to use a different color scheme for some SublimeGit view altogether, you can do this by going to `Preferences > Settings > More > Syntax Specific - User` while having a SublimeGit view

open (i.e. the status or commit view), and then adding a color scheme setting for the given syntax like so:

```
"color_scheme": "Packages/Color Scheme - Default/Monokai.tmTheme"
```

#### Customizing Individual Colors

A full detailing of creating a color scheme is outside the scope of this documentation. A quick googling on `sublime text color schemes` or `textmate color schemes` should bring up plenty of resources.

To find out which scope you will need to colorize, put the cursor on the text in question, and press `ctrl+shift+p`. This will show the scope under the cursor in the status bar.

# Troubleshooting

## SublimeGit Can't Find Git

Please see *Git Executable Path*

## Nothing Happens When Pushing, Pulling or Fetching From a Remote

Please see *Git Configuration*. Below you will find solutions submitted by SublimeGit users:

#### Solution by Albert Santini (Issue #3)

I configured the git bash shell on windows 7 following GitHub help to start a ssh agent, because I don't want to type every time the passphrase for my ssh key.

I added to that configuration, in .bashrc, the following lines:

```
setx SSH_AUTH_SOCK $SSH_AUTH_SOCK 1> nul
setx SSH_AGENT_PID $SSH_AGENT_PID 1> nul
```

These lines add the environment variables to windows user profile.

So the git executable, configured in SublimeGit, can read the variables and use the correct protocol.

Firstly I start the bash shell and then I start SublimeText editor.

Now SublimeGit works perfectly.

#### Solution by Henry Mei (Issue #15)

I am outlining my workaround and hope this will be beneficial for anyone working with Windows.

It seems that SublimeGit requires credential storing for the command prompt (i.e. cmd.exe) and not Git Bash. I will assume that we're using msysgit. Make sure Git is added to your PATH.

1. Grab a copy of PuTTY, Plink, Pageant, and PuTTYgen from here and save them somewhere (e.g. I just threw them all in C:PuTTY).
2. Add a system variable called GIT_SSH that points to the location of Plink (e.g. C:PuTTYplink.exe). If you're using an older version of mysysgit, there was actually an option to use Plink instead of OpenSSH.

3. Generate your public/private key pair using PuTTYgen. Be sure to secure your key by using a passphrase. You should be generating a SSH-2 RSA key of typically 1024 bits. Save the private key somewhere, and add the public key generated to the list of SSH public keys on your GitHub account (i.e. go to github.com and look in your account settings).
4. Grab GitHub's public key. Use PuTTY to SSH into github.com. If you've never done this before, it should pop up an alert saying that the server's host key is not cached in the registry. Hit "Yes" to add the key to PuTTY's cache. After doing this, exit PuTTY. We won't be using it again.
5. Run Pageant. This will create an icon in your system tray. Double click to open a window where you can add your private key. The agent will sit in the background, much like ssh-agent, and provide authentication when necessary.

---

**Note:** If you tried the OpenSSH workaround detailed here, you can just convert your OpenSSH private key to a PuTTY key also using PuTTYgen (the public key will be same regardless). Your OpenSSH keys will be in ~.ssh, which is %USERPROFILE%.ssh . OpenSSH public keys have the *.pub extension and private keys no extension. PuTTY private keys have the *.ppk extensions. Make sure to choose the OpenSSH private key when opening with PuTTYgen and save it as a *.ppk.

---

As long as Pageant is running, any git calls through the command prompt should be automatically authenticated, allowing SublimeGit to not freeze.

Pageant will default to a clean session every time it runs, but it takes key paths as parameters (i.e. pageant.exe ... ). There are a few ways to make things easier. You can add the path to the keys after the target path in the Pageant shortcut (i.e. for me, this would be "C:PuTTYpageant.exe" %USERPROFILE%.sshid_rsa.ppk ) or just write a batch file to make it autostart in Windows. Pageant will always prompt for the passphrases of keys you auto-load on startup.

### Solution by Mario Basic (Issue #59)

If you are on Windows and when you try to push or pull using this plugin nothing happens or it pushes forever, You have to add a system variable to your SSH keys.

- Right-click on Computer
- Choose Properties
- Click on Advanced System Settings
- Click on Environment Variables
- In the bottom section (System Variables) Click on New
- For Variable name type: `HOME`
- For Variable path type: `C:\Users\your-user-folder\`
- Click OK

## The Output From Git Commands Look Weird (ANSI Escape Codes)

This happens if you have any of the `color.*` git options set to `true` (or `always`). SublimeGit tries to remove the colors on everything, but sometimes one slip through. If you see one in the wild, please report it at support@sublimegit.net.

To make sure that you don't get the escape codes in SublimeGit, but still get pretty colors when using git from the terminal, we recommend setting the `color.*` config values to `auto` like so:

```
git config --global color.ui auto
git config --global color.branch auto
git config --global color.diff auto
git config --global color.status auto
```

After which the relevant part of your `.gitconfig` will look something like this:

```
[color]
    diff = auto
    status = auto
    branch = auto
    ui = auto
```

# More Information

Here are some random links for getting started with, and using git and Sublime Text 2.

## Git

For getting started with git, the book "Pro Git" by Scott Chacon is chock-full of great information, and it's available for free: http://git-scm.com/book

The git reference manual is a great place to look up specifics for commands: http://git-scm.com/docs

PeepCode also has a couple of good screencasts on git:

- https://peepcode.com/products/git
- https://peepcode.com/products/advanced-git

For a more academic and in-depth treatment, which can be very helpful for wrapping your head around git, see: http://www.sbf5.com/~cduan/technical/git/

And last, but certainly not least, the help section on GitHub is excellent, and should be in your bookmarks: https://help.github.com/

## Git-flow

The original blog-post on the git-flow branching model is here: http://nvie.com/posts/a-successful-git-branching-model/

## Sublime Text 2

Please see the Sublime Text 2 Support page: http://www.sublimetext.com/support

The Sublime Text Forum is also very active, and helpful: http://www.sublimetext.com/forum/

CHAPTER 2

# Indices and tables

- genindex
- search

# Index

## T



## V