# Douglas Documentation

***Release 0.1***

**Doulgas CC0**

**September 27, 2017**

# Contents

# CHAPTER 1

## Summary

Douglas is a file-based blog system written in Python with the following features:

- compiler
- WSGI application
- runs as a CGI script (woo-hoo!)
- plugin system for easy adjustment of transforms
- Jinja renderer
- basic set of built-in plugins

Douglas is a rewrite of Pyblosxom.

CHAPTER 2

# Quick start

1. Install:

```
$ pip install https://github.com/willkg/douglas/archive/master.zip#egg=douglas``
```

2. Create a new blog:

```
$ douglas-cmd create blog
$ cd blog
```

3. Edit the configuration
4. Write a blog entry

```
$ vi entries/firstpost.txt
```

5. Compile the blog

```
$ douglas-cmd compile
```

6. Copy the static assets (JS, CSS, images, ...)

```
$ douglas-cmd collectstatic
```

7. Preview it locally

```
$ douglas-cmd serve
```

8. Copy it to your server

CHAPTER 3

# Part 1: Douglas user manual

Documentation for installing, configuring and tweaking Douglas for your purposes.

## About Douglas

### What is this?

Douglas is a file-based blog system written in Python with the following features:

- compiler
- WSGI application
- runs as a CGI script (woo-hoo!)
- plugin system for easy adjustment of transforms
- Jinja renderer
- basic set of built-in plugins

Douglas is a rewrite of Pyblosxom.

### Status

There are other file-based blog systems out there that have a more complete feature set. I continued this one because at the time, it was easier to continue working on this than to switch.

However, I've now switched to nikola. It's pretty swell. It was easy to switch my Pyblosxom/Douglas blog over.

Ergo, this project is dead for now.

## Project

**Code** https://github.com/willkg/douglas

**License** MIT

**Issues** https://github.com/willkg/douglas/issues

**Docs** https://douglas.readthedocs.io/

**Status** **Extreme Alpha**

## Requirements

- Python 2.7
- possibly other requirements depending on what plugins you install

## Quickstart for compiling a new blog

1. Create a virtual environment
2. Activate the virtual environment
3. Install Douglas into your virtual environment:

```
pip install https://github.com/willkg/douglas/archive/master.zip#egg=douglas
```

4. Create a new blog structure:

```
douglas-cmd create <blog-dir>
```

   For example: `douglas-cmd create blog`

5. Edit the `blog/config.py` file. There should be instructions on what should get changed and how to change it.
6. Change directories to `blog` and then render the site:

```
douglas-cmd compile
```

7. Collect the static assets:

```
douglas-cmd collectstatic
```

8. Copy the `compiled_site/` directory tree to where they're available for serving by your web server.

## Where to go from here

Each file in `blog/entries/` is a blog entry. They are text files. You can edit them with any text editor.

The blog is rendered using Jinja2. The templates are in the `blog/themes/` directory. A theme consists of:

- a `content_type` file which has the mimetype of the output being rendered (e.g. `text/html`)
- an `entry.<themename>` file which is used when rendering a page with a single entry
- an `entry_list.<themename>` file which is used when rendering a page with a bunch of entries (e.g. category list, date archive list, front page, ...)

- additional template files required by plugins as specified by those plugins
- static assets like CSS files, JS files and images in the `static/<themename>/` subdirectory

The following plugins which come with Douglas are enabled by default in your `load_plugins` config property:

`douglas.plugins.draft_folder`

Creates a draft folder that you can view on the web-site, but doesn't show up in the archive links. This makes it easier for other people to review entries before they're live.

The draft dir is `blog/drafts/`.

When you want to make an entry live, you move it from `blog/drafts/` to `blog/entries/`.

`douglas.plugins.published_date`

Add `#published YYYY-MM-DD HH:MM` to the metadata in your blog entries. That's the published date for the blog entry rather than the mtime of the file.

Douglas comes with other useful plugins. Refer to the documentation for a list.

You can write your own plugins and put the plugin files in `blog/plugins/` and add the plugin Python module to the `load_plugins` list in your `config.py` file.

# License

## Douglas



# What's new in Douglas

## Version 0.1: In development

Changes:

- Initial writing.

# Douglas on the command line

Douglas comes with a command line tool called `douglas-cmd`. It allows you to create new blogs, verify your configuration, run compiling, render single urls, and run command line functions implemented in plugins.

For help, do:

```
$ douglas-cmd --help
```

It'll list the commands and options available.

If you tell it where your config file is, then it'll list commands and options available as well as those implemented in plugins you have installed.

For example:

```
$ douglas-cmd --config=/path/to/config.py --help
```

For more information on compiling, see *Deploy Douglas as a Compiled HTML Site*.

# Deploy Douglas as a Compiled HTML Site



## Summary

Compiling your blog to static HTML allows you to generate your blog and push it via scp or ftp to host most anywhere. If your blog isn't interactive or you need to host your blog on a system that doesn't let you run a CGI script or a WSGI app, then this is the easiest way to do it.

Compiling your blog happens in two steps:

1. `douglas-cmd compile`
2. `douglas-cmd collectstatic`

After that, you'll have a single directory with the compiled form of your blog. You can scp, ftp, rsync, unison or whatever this directory to the host for serving.

## Configuring compiling

To compile your blog, you need to set the `compiledir` setting in your `config.py` file. That tells Douglas which directory to compile your blog to. Everything else is optional and has defaults.

***base_url*** For example, if your `compiledir` were set to `/home/joe/public_html` and the url for that directory were `http://example.com/~joe/`, then you probably want to set your base_url like this:

```
py["base_url"] = "http://example.com/~joe"
```

***compiledir*** The directory to compile your blog into.

***compile_themes*** The themes to compile all pages in.

***compile_index_themes*** The themes to compile index pages in.

***day_indexes*** Whether or not to do day-based indexes.

***month_indexes*** Whether or not to do month-based indexes.

***year_indexes*** Whether or not to do year-based indexes.

***compile_urls*** List of additional urls to compile.

## Configuring collectstatic

***static_url*** The url where your static assets will be. If you're using a CDN, then this will be a complete url. Otherwise you probably want to set this to your *base_url* plus `/static`.

**static_files_dir** The list of additional directories to copy static assets from.

## Compiling your blog

To compile your blog, `cd` into your blog's directory and run:

```
$ douglas-cmd compile
```

After that, collect the static files:

```
$ douglas-cmd collectstatic
```

Once you've done both of those steps, you can copy the compiledir to your blog host.

See:

```
$ douglas-cmd compile --help
```

and:

```
$ douglas-cmd collectstatic --help
```

for options.

## Example setup

I keep my blog on my server in `/home/will/blog`. I compile it to my `/home/will/public_html` directory.

My directory layout looks like:

```
/home/will
   blog/
     |- static/
     |  |- images/
     |  |- css/
     |  \- js/
     |
     |- entries/       # all my blog entries
     |- themes/        # themes and templates
     |- plugins/       # a couple of plugins I use
     |
     |- config.py      # my config.py file
     |- compile.sh     # shell script below
```

Here's the relevant portions of my `config.py` file:

```
py["base_url"] = "http://example.com/~joe/blog"

py["compiledir"] = "/home/will/public_html/blog/"
py["compile_themes"] = ["html"]
py["compile_index_themes"] = ["html", "atom"]
py["compile_day_indexes"] = False
py["compile_month_indexes"] = False
py["compile_year_indexes"] = True

py["static_url"] = "http://example.com/~joe/blog/static"
py["static_files_dirs"] = []
```

My compile.sh file looks like this:

```
#!/bin/bash

BLOGDIR=/home/will/blog
OUTPUTDIR=/home/will/public_html/blog

# compile entire blog
douglas-cmd compile --config ${BLOGDIR}

# copy static assets
douglas-cmd collectstatic --config ${BLOGDIR}
```

## Troubleshooting

### Can't find config.py file

Use the `--config <path/to/config.py/file>` argument.

### There are all these old files in my compiledir

Both compiling everything and compiling incrementally *won't* remove outdated files. If you want old files removed, you should delete the directory, then compile and collect static files.

#### OMG! I don't want an RSS version of every page in my blog!

You probably don't want to compile an RSS or Atom version of every blog entry, so don't include those themes in `compile_themes` and instead specify the themes you want for index pages in `compile_index_themes` or the specific urls you want in `compile_urls`.

#### I want to use a CSS compiler, JS minifier, etc

Put your CSS/JS source files in your static directories, then compile them into their CSS/JS forms and then run:

```
$ douglas-cmd collectstatic
```

## Deploying Douglas with CGI

### Summary

You can run Douglas as a CGI script with many web servers. This document covers setting Douglas up as a CGI script.

### Dependencies

You need an account on a web server configured to run CGI scripts. It helps to know how to run CGI scripts on that server, too.

### Deployment

1. Copy the `douglas.cgi` file from the blog directory (the directory which you created with `douglas-cmd create ./blog/`) into your CGI directory.
2. Edit the `douglas.cgi` file.

   The top of the file looks something like this

```
#!/usr/bin/env python

# -u turns off character translation to allow transmission
# of gzip compressed content on Windows and OS/2
#!/path/to/python -u

import os, sys

# Uncomment this line to add the directory your config.py file is
# in to the python path:
#sys.path.append("/path/to/directory/")
```

   Make sure the first line points to a valid python interpreter. If you're using virtualenv, then make sure it points to the python interpreter in the virtual environment.

   Uncomment the `sys.path.append("/path/to/directory/")` line and make sure the path being appended is the directory that your `config.py` file is in.

4. Make sure the `douglas.cgi` file has the correct permissions and ownership for running a CGI script in this directory for the server that you're using.

5. Make sure your blog directory has the correct permissions for being read by the process executing your CGI script.
6. Run your `douglas.cgi` script by doing:

```
$ ./douglas.cgi test
```

   If that doesn't work, double-check to make sure you've completed the above steps, then check the trouble-shooting section below.

If that does work, then try to run the CGI script from your web browser. The url is dependent on where you put the `douglas.cgi` script and how CGI works on your web server.

## Trouble-shooting

We're going to try to break this down a bit into categories. Bear with us and keep trying things.

### Running ./douglas.cgi doesn't work

If Python is installed on your system, make sure the first line in `douglas.cgi` points to the correct Python interpreter. By default, `douglas.cgi` uses `env` to execute the Python interpreter. In some rare systems, `/usr/bin/env` doesn't exist or the system may have odd environment settings. In those cases, you may edit the first line to point to the Python interpreter directly. For example:

```
#!/usr/bin/python
```

Then try running `./douglas.cgi` again.

If Python is installed on your system and the first line of `douglas.cgi` is correct, check for permissions issues. `douglas.cgi` is a script, so it needs execute permission in order to function. If those aren't set, then fix that and try running `./douglas.cgi` again.

Check the error logs for your web server.

### I see a HTTP 404 error when I try to bring up my blog

When you try to look at your blog and you get a HTTP 404 error, then you're using the wrong URL. Here are some questions to ask yourself:

- Are you using an `.htaccess` file?
- Does your server allow you to run CGI scripts?
- Do other CGI scripts in this directory work?
- Does the URL you're trying to use to access Douglas look like other URLs that work on your system?

### I see a HTTP 500 error when I try to bring up my blog

At this point, running `./douglas.cgi` at the command prompt should work fine. If you haven't done that and you're busy trouble-shooting, go back and review the deployment instructions.

If the problem is with Douglas and not your web server, then you should see a pretty traceback that will help you figure out what the specific problem is.

If the traceback and information doesn't make any sense to you, add an issue to the issue tracker.

If you don't see a traceback, then you either have a configuration problem with your web server or a configuration problem with Python. The first thing you should do is check your web server's error logs. For Apache, look for the `error.log` file in a place like `/var/logs/apache/` or `/var/logs/httpd/`. If you don't know where your web server's error logs are, ask your system administrator.

Does the account your web server runs as have execute access to your `douglas.cgi` script? If your web server does not have the permissions to read and execute your `douglas.cgi` script, then your blog will not work.

Do you have plugins loaded? If you do, comment out the `load_plugins` setting in your `config.py` file so that Douglas isn't loading any plugins.

For example:

```
py["load_plugins"] = ['plugina', 'pluginb', ...]
```

would get changed to:

```
# commenting this out to see if it's a plugin problem
# py["load_plugins"] = ['plugina', 'pluginb', ...]
```

Check to see if the problem persists. Sometimes there are issues with plugins that only show up in certain situations.

### I have other issues

Try changing the renderer for your blog to the debug renderer. You can do this by setting the `renderer` property in your `config.py` file to `debug`. For example:

```
py["renderer"] = "debug"
```

That will show a lot more detail about your configuration, what the web server passes Douglas in environment variables, and other data about your blog that might help you figure out what your problem is.

If that doesn't help, add an issue to the issue tracker.

### UGH! My blog looks UGLY!

Check out *Renderers, Themes and Templates*.

### I hate writing in HTML!

That's ok. Douglas supports formatters and entry parsers which allow you to use a variety of markups for writing blog entries. See the documentation on Writing Entries for more information.

Check out *Categories*.

## Advanced installation

We encourage you not to try any of this until you've gotten a blog up and running.

This section covers additional advanced things you can do to your blog that will make it nicer. However, they're not necessary and they're advanced and we consider these things to be very much a "you're on your own" kind of issue.

If you ever have problems with Douglas and you ask us questions on the douglas-users or douglas-devel mailing lists, make sure you explicitly state what things you've done from this chapter. It'll go a long way in helping us to help you.

## Renaming the douglas.cgi script

In the default installation, the Douglas script is named `douglas.cgi`.

For a typical user on an Apache installation with user folders turned on, Douglas URLs could look like this:

```
http://example.com/~joe/cgi-bin/douglas.cgi
http://example.com/~joe/cgi-bin/douglas.cgi/an_entry.html
http://example.com/~joe/cgi-bin/douglas.cgi/dev/another_entry.html
```

That gets pretty long and it's not very good looking. For example, telling the URL to your mother or best friend over the phone would be challenging. It would be nice if we could shorten and simplify it.

So, we have some options:

- Change the name of the `douglas.cgi` script.
- And if that's not good enough for you, use the Apache mod_rewrite module to get URLs internally redirected to the `douglas.cgi` script.

Both methods are described here in more detail.

### Change the name of the douglas.cgi script

There's no reason that `douglas.cgi` has to be named `douglas.cgi`. Let's try changing it `blog`. Now our example URLs look like this:

```
http://example.com/~joe/cgi-bin/blog
http://example.com/~joe/cgi-bin/blog/an_entry.html
http://example.com/~joe/cgi-bin/blog/category1/another_entry.html
```

That's better looking in the example. In your specific circumstances, that may be all you need.

You might have to change the `base_url` property in your `config.py` file to match the new URL.

---

**Note:** The `base_url` value should NOT have a trailing slash.

---

If you're running on Apache, you might have to tell Apache that this is a CGI script even if it doesn't have a `.cgi` at the end of it. If you can use `.htaccess` files to override Apache settings, you might be able to do something like this:

```
# this allows execution of CGI scripts in this directory
Options ExecCGI

# if the user doesn't specify a file, then instead of doing the
# regular directory listing, we look at "blog" (which is our
# douglas.cgi script renamed)
DirectoryIndex blog

# this tells Apache that even though "blog" doesn't end in .cgi,
# it is in fact a CGI script and should be treated as such
<Files blog>
ForceType application/cgi-script
SetHandler cgi-script
</Files>
```

You may need to stop and restart Apache for your Apache changes to take effect.

### Hiding the .cgi with RewriteRule

Apache has a module for URL rewriting which allows you to convert incoming URLs to other URLs that can be handled internally. You can do URL rewriting based on all sorts of things. See the Apache manual for more details.

In our case, we want all incoming URLs pointing to `blog` to get rewritten to `cgi-bin/douglas.cgi` so they can be handled by Douglas. Then all our URLs will look like this:

```
http://example.com/~joe/blog
http://example.com/~joe/blog/an_entry.html
http://example.com/~joe/blog/category1/another_entry.html
```

To do this, we create an .htaccess file (it has to be named exactly that) in our `public_html` directory (or wherever it is that `/~joe/` points to). In that file we have the following code:

```
RewriteEngine on
RewriteRule   ^blog?(.*)$   /~joe/cgi-bin/douglas.cgi$1   [last]
```

The first line turns on the Apache mod_rewrite engine so that it will rewrite URLs.

The second line has four parts. The first part denotes the line as a RewriteRule. The second part states the regular expression that matches the part of the URL that we want to rewrite. The third part denotes what we're rewriting the URL to. The fourth part states that after this rule is applied, no future rewrite rules should be applied.

If you do URL rewriting, you may have to set the base_url property in your `config.py` accordingly. In the above example, the `base_url` would be `http://example.com/~joe/blog` with no trailing slash.

For more information on URL re-writing, see the mode_rewrite chapter in the Apache documentation for the version that you're using.

# Deploying Douglas with Paste

## Summary

Douglas supports Paste. This document covers installing and using Douglas with Paste.

This installation assumes you have some understanding of Python Paste. If this doesn't sound like you, then you can read up on Paste on the Paste website or the Wikipedia page.

## Dependencies

You'll need:

- Python Paste which can be found at http://pythonpaste.org/

Install:

```
$ pip install pastescript
```

## Deployment for testing

Create a new blog by doing:

```
$ douglas-cmd create <BLOG-DIR>
```

Then do:

```
$ cd <BLOG-DIR>
$ paster serve blog.ini
```

The `paster` script will print the URL for your blog on the command line and your blog is now available on your local machine to a browser on your local machine.

This allows you to test your blog and make sure it works.

### Paste .ini file configuration

Paste configuration is done in an `.ini` file.

Edit the `blog.ini` file that `douglas-cmd` created for you.

The `[server:main]` section dictates how Paste is serving your blog. See the Paste documentation for more details on this section.

The `[app:main]` section specifies the Douglas WSGI application function and the directory your `config.py` file is in. A sample is here:

```
[app:main]
paste.app_factory = Douglas.douglas:douglas_app_factory
configpydir = /home/joe/blog/
```

Additionally, you can override `config.py` values in your `blog.ini`. For example, this overrides the `blog_title` value:

```
[app:main]
paste.app_factory = Douglas.douglas:douglas_app_factory
configpydir = /home/joe/blog/

# Douglas config here
blog_title = Joe's Blog
```

This is really handy for testing changes to your blog infrastructure.

## Deploying Douglas with Apache and mod_wsgi

### Summary

This walks through install Douglas as an WSGI application on an Apache web server with mod_wsgi installed.

If you find any issues, please let us know.

If you can help with the documentation efforts, please let us know.

### Dependencies

- Apache
- mod_wsgi

- administrative priveliges to the server

## Deployment

1. Make sure mod_wsgi is installed correctly and working.
2. Create a blog—see the instructions for the blog directories, `config.py` setup and other bits of **Setting up a blog** in `install_cgi`.
3. Create a `douglas.wsgi` script that looks something like this:

```
# This is the douglas.wsgi script that powers the _______
# blog.

import sys

def add_to_path(d):
   if d not in sys.path:
      sys.path.insert(0, d)

# call add_to_path with the directory that your config.py lives in.
add_to_path("/home/joe/blog")

# if you have Douglas installed in a directory and NOT as a
# Python library, then call add_to_path with the directory that
# Douglas lives in.  For example, if I untar'd
# douglas-1.5.tar.gz into /home/joe/, then add like this:
# add_to_path("/home/joe/douglas-1.5/")

import Douglas.douglas
application = Douglas.douglas.DouglasWSGIApp()
```

4. In the Apache conf file, add:

```
WSGIScriptAlias /myblog /path/to/something.wsgi

<Directory /path/to>
    Order deny,allow
    Allow from all
</Directory>
```

   Change `/myblog` to the url path you want your blog to live at.

   Change `/path/to/something.wsgi` to be the absolute path to the .wsgi file set up in step 3.

   Change `/path/to` to the directory of the .wsgi file.

5. Restart the Apache web server.

**Note:** Any time you make changes to Douglas (update, add plugins, change configuration), you'll have to restart Apache.

# Configuring Douglas

You configure a Douglas blog by setting configuration variables in a Python file called `config.py`. Each Douglas blog has its own `config.py` file.

This chapter documents the `config.py` variables. Some of these are required, others are optional.

---

**Note:** Douglas comes with a sample config file. This file does **not** have everything listed below in it. If you want to use a variable that's not listed in your config file—just add it.

---

## Config variables and syntax

Each configuration variable is set with a line like:

```
py["blog_title"] = "Another douglas blog"
```

where:

- `blog_title` is the name of the configuration variable
- `"Another douglas blog"` is the value

Most configuration values are strings and must be enclosed in quotes, but some are lists, numbers or other types of values.

Examples:

```
# this has a string value
py["foo"] = "this is a string"

# this is a long string value
py["foo"] = (
    "This is a really long string value that breaks over "
    "multiple lines.  The parentheses cause Python to "
    "allow this string to span several lines."
)

# this has an integer value
py["foo"] = 4

# this is a boolean--True has a capital T
py["foo"] = True

# this is a boolean--False has a capital F
py["foo"] = False

# this is a list of strings
py["foo"] = [
    "list",
    "of",
    "strings"
]

# this is the same list of strings formatted slightly differently
py["foo"] = ["list", "of", "strings"]
```

Since `config.py` is a Python code file, it's written in Python and uses Python code conventions.

## Plugin variables

If you install any Douglas plugins those plugins may ask you to set additional variables in your `config.py` file. Those variables will be documented in the documentation that comes with the plugin or at the top of the plugin's source code file. Additional plugin variables will not be documented here.

## Personal configuration variables

You can add your own personal configuration variables to `config.py`. You can put any `py["name"] = value` statements that you want in `config.py`. You can then refer to your configuration variables further down in your `config.py` file and in your theme templates. This is useful for allowing you to centralize any configuration for your blog into your `config.py` file.

For example, you could move all your media files (JPEG images, GIF images, CSS, ...) into a directory on your server to be served by Apache and then set the config.py variable `py["media_url"]` to the directory with media files and use `$media_url` to refer to this URL in your theme templates.

## Configuration variables

**class** `douglas.settings.`**`Config`**

**`base_url`** **= Required**
Set `base_url` in your `config.py` file to the base url for your blog. If someone were to type this url into their browser, they'd see the front page of your blog.

---

**Note:** Your `base_url` property should **not** have a trailing slash.

---

**`blog_author`** **= (Optional) Default is ''**
This is the name of the author of your blog. Very often this is your name or a pseudonym.

If Joe Smith had a blog, he might set his `blog_author` to "Joe Smith":

```
py["blog_author"] = "Joe Smith"
```

If Joe Smith had a blog, but went by the pseudonym "Magic Rocks", he might set his `blog_author` to "Magic Rocks":

```
py["blog_author"] = "Magic Rocks"
```

**`blog_description`** **= (Optional) Default is ''**
This is the description or byline of your blog. Typically this is a phrase or a sentence that summarizes what your blog covers.

If you were writing a blog about restaurants in the Boston area, you might have a `blog_description` of:

```
py["blog_description"] = "Critiques of restaurants in the Boston area"
```

Or if your blog covered development on Douglas, your `blog_description` might go like this:

```
py["blog_description"] = (
    "Ruminations on the development of Douglas and "
    "related things that I discovered while working on "
    "the project")
```

**blog_email = (Optional) Default is ''**

This is the email address you want associated with your blog.

For example, say Joe Smith had an email address joe@joesmith.net and wanted that associated with his blog. Then he would set the email address as such:

```
py["blog_email"] = "joe@joesmith.net"
```

**blog_encoding = (Optional) Default is 'utf-8'**

This is the character encoding of your blog.

For example, if your blog was encoded in utf-8, then you would set the blog_encoding to:

```
py["blog_encoding"] = "utf-8"
```

---

**Note:** This value must be a valid character encoding value. In general, if you don't know what to set your encoding to then set it to utf-8.

---

This value should be in the meta section of any HTML- or XHTML-based themes and it's also in the header for any feed-based themes. An improper encoding will gummy up some/most feed readers and web-browsers.

W3C has a nice tutorial on encoding. You may refer to IANA charset registry for a complete list of encoding names.

**blog_language = (Optional) Default is ''**

This is the primary language code for your blog.

For example, English users should use en:

```
py["blog_language"] = "en"
```

This gets used in the RSS themes.

Refer to ISO 639-2 for language codes. Many systems use two-letter ISO 639-1 codes supplemented by three-letter ISO 639-2 codes when no two-letter code is applicable. Often ISO 639-2 is sufficient. If you use very special languages, you may want to refer to ISO 639-3, which is a super set of ISO 639-2 and contains languages used thousands of years ago.

**blog_rights = (Optional) Default is ''**

These are the rights you give to others in regards to the content on your blog. Generally this is the copyright information, for example:

```
py["blog_rights"] = "Copyright 2005 Joe Bobb"
```

This is used in the Atom and RSS 2.0 feeds. Leaving this blank or not filling it in correctly could result in a feed that doesn't validate.

**blog_title = (Optional) Default is 'My blog'**

This is the title of your blog. Typically this should be short and is accompanied by a longer summary of your blog which is set in blog_description.

For example, if Joe were writing a blog about cooking, he might title his blog:

```
py["blog_title"] = "Joe's blog about cooking"
```

**compile_index_themes = (Optional) Default is ['html']**

compile_index_themes is just like compile_themes except it's the themes of the index files: frontpage index, category indexes, date indexes, ...

Defaults to ["html"] which only renders the html theme.

For example:

```
py["compile_index_themes"] = ["html"]
```

If you want your index files to also be feeds, then you should add a feed theme to the list.

**compile_themes = (Optional) Default is ['html']**

The value of compile_themes should be a list of strings representing all the themes that should be rendered.

For example:

```
py["compile_themes"] = ["html"]
```

**compile_urls = (Optional) Default is []**

Any other url paths to compile. Sometimes plugins require you to add additional paths—this is where you'd do it.

For example:

```
py["compile_urls"] = [
    "/booklist"
]
```

**compiledir = (Optional) Default is ''**

This is the directory we will save all the output. The value of compiledir should be a string representing the **absolute path** of the output directory for compiling.

For example, Joe puts the output in his public_html directory of his account:

```
py["compiledir"] = "/home/joe/public_html"
```

**datadir = Required**

This is the full path to where your blog entries are kept on the file system.

For example, if you are storing your blog entries in /home/joe/blog/entries/, then you would set the datadir like this:

```
py["datadir"] = "/home/joe/blog/entries/"
```

---

**Note:** Must not end with a /.

---

**day_indexes = (Optional) Default is False**

Whether or not to generate indexes per day.

For example:

```
py["day_indexes"] = True
```

**default_theme = (Optional) Default is 'html'**

This specified the theme that will be used if the user doesn't specify a theme in the URI.

For example, if you wanted your default theme to be "joy", then you would set `default_theme` like this:

```
py["default_theme"] = "joy"
```

Doing this will cause Douglas to use the "joy" theme whenever URIs are requested that don't specify the theme.

For example, the following will all use the "joy" theme:

```
http://example.com/blog/
http://example.com/blog/index
http://example.com/blog/movies/
http://example.com/blog/movies/supermanreturns
```

**depth = (Optional) Default is 0**

The depth setting determines how many levels deep in the directory (category) tree that Douglas will display when doing indexes.

- 0 - infinite depth (aka grab everything) DEFAULT
- 1 - datadir only
- 2 - two levels
- 3 - three levels
- ...
- *n* - *n* levels deep

**entryparsers = (Optional) Default is {}**

Lets you override which file extensions are parsed by which entry parsers. The keys are the file extension. The values are the Python module path to the callable that will parse the file.

For example, by default, the blosxom_entry_parser parses files ending with `.txt`. You can also have it parse files ending in `.html`:

```
py["entryparsers"] = {
    'html': 'douglas.app:blosxom_entry_parser'
}
```

The `douglas.app` part denotes which Python module the callable is in. The `blosxom_entry_parser` part is the name of a function in the `douglas.app` module which will parse the entry.

**ignore_directories = (Optional) Default is []**

The `ignore_directories` variable allows you to specify which directories in your datadir should be ignored by Douglas.

This defaults to an empty list (i.e. Douglas will not ignore any directories).

For example, if you use CVS to manage the entries in your datadir, then you would want to ignore all CVS-related directories like this:

```
py["ignore_directories"] = ["CVS"]
```

If you were using CVS and you also wanted to store drafts of entries you need to think about some more in a drafts directory in your datadir, then you could set your `ignore_directories` like this:

```
py["ignore_directories"] = ["drafts", "CVS"]
```

This would ignore all directories named "CVS" and "drafts" in your datadir tree.

**load_plugins = (Optional) Default is []**

Specifying load_plugins causes Douglas to load only the plugins you name and in in the order you name them.

The value of load_plugins should be a list of strings where each string is the name of a Python module.

If you specify an empty list no plugins will be loaded.

---

**Note:** Douglas loads plugins in the order specified by load_plugins. This order also affects the order that callbacks are registered and later executed. For example, if plugin_a and plugin_b both implement the handle callback and you load plugin_b first, then plugin_b will execute before plugin_a when the handle callback kicks off.

Usually this isn't a big deal, however it's possible that some plugins will want to have a chance to do things before other plugins. This should be specified in the documentation that comes with those plugins.

---

**log_file = (Optional) Default is <open file '<stderr>', mode 'w'>**

This specifies the file that Douglas will log messages to.

If Douglas cannot open the file for writing, then log messages will be sent to sys.stderr.

For example, if you wanted Douglas to log messages to /home/joe/blog/logs/douglas.log, then you would set log_file to:

```
py["log_file"] = "/home/joe/blog/logs/douglas.log"
```

If you were on Windows, then you might set it to:

```
py["log_file"] = "c:/blog/logs/douglas.log"
```

---

**Note:** The web server that is executing Douglas must be able to write to the directory containing your douglas.log file.

---

**log_level = (Optional) Default is 'error'**

This is based on the Python logging module, so the levels are the same:

- critical
- error
- warning
- info
- debug

This sets the log level for logging messages.

If you set the log_level to critical, then *only* critical messages are logged.

If you set the log_level to error, then error and critical messages are logged.

If you set the log_level to warning, then warning, error, and critical messages are logged.

So on and so forth.

For "production" blogs (i.e. you're not tinkering with configuration, new plugins, new themes, or anything along those lines), then this should be set to `warning` or `error`.

For example, if you're done tinkering with your blog, you might set the `log_level` to `info` allowing you to see how requests are being processed:

```
py['log_level'] = "info"
```

**`month_indexes` = (Optional) Default is False**
Whether or not to generate indexes per month.

For example:

```
py["month_indexes"] = True
```

**`num_entries` = (Optional) Default is 10**
The `num_entries` variable specifies the number of entries that show up on your home page and other category index pages. It doesn't affect the number of entries that show up on date-based archive pages.

It defaults to 5 which means "show at most 5 entries".

If you set it to 0, then it will show all entries that it can.

For example, if you wanted to set `num_entries` to 10 so that 10 entries show on your category index pages, you sould set it like this:

```
py["num_entries"] = 10
```

**`plugin_dirs` = (Optional) Default is []**
The `plugin_dirs` variable tells Douglas which directories to look for plugins in addition to the plugins that Douglas comes with. You can list as many directories as you want.

For example, if your blog used the "paginate" plugin that comes with Douglas and a "myfancyplugin" that you wrote yourself that's in your blog's plugins directory, then you might set `plugin_dirs` like this:

```
py["plugin_dirs"] = [
    "/home/joe/blog/plugins/"
]
```

---

**Note:** Plugin directories are not searched recursively for plugins. If you have a tree of plugin directories that have plugins in them, you'll need to specify each directory in the tree.

For example, if you have plugins in `~/blog/my_plugins/` and `~/blog/phils_plugins/`, then you need to specify both directories in `plugin_dirs`:

```
py["plugin_dirs"] = [
    "/home/joe/blog/my_plugins",
    "/home/joe/blog/phils_plugins"
    ]
```

You can't just specify `~/blog/` and expect Douglas to find the plugins in the directory tree:

```
# This won't work!
py["plugin_dirs"] = [
    "/home/joe/blog"
    ]
```

---

**Note:** Plugins that come with Douglas are automatically found—you don't have to specify anything in your''plugin_dirs'' in order to use core plugins.

**renderer = (Optional) Default is 'jinjarenderer'**
The `renderer` variable lets you specify which renderer to use.

**static_files_dirs = (Optional) Default is []**
Any additional directories you want copied over to the compiledir.

For example:

```
py['static_files_dirs'] = [
    '/home/joe/blog/staticimages/',
    '/home/joe/blog/blogimages/'
]
```

**static_url = (Optional) Default is ''**
The url where your static assets will be.

If you're using a CDN, this is the CDN url.

If you're not using a CDN, this is probably the `base_url` plus `/static`.

You can use this variable in your templates. For example:

```
<link rel="stylesheet" href="{{ static_url }}/css/style.css">
```

**themedir = Required**
This is the full path to where your Douglas themes are kept.

If you do not set the `themedir`, then Douglas will look for your themes and templates in the datadir alongside your entries.

**Note:** "theme" is spelled using the British spelling and not the American one.

For example, if you want to put your entries in `/home/joe/blog/entries/` and your theme templates in `/home/joe/blog/themes/` you would set `themedir` and `datadir` like this:

```
py["datadir"] = "/home/joe/blog/entries/"
py["themedir"] = "/home/joe/blog/themes/"
```

**truncate_category = (Optional) Default is True**
Whether or not to truncate the number of entries displayed on a category-based index page to `num_entries` number of entries.

For example, this causes all entries in a category to show up in all category-based index pages:

```
py["truncate_category"] = False
```

**truncate_date = (Optional) Default is False**
Whether or not to truncate the number of entries displayed on a date-based index page to `num_entries` number of entries.

**truncate_frontpage = (Optional) Default is True**
Whether or not to truncate the number of entries displayed on teh front page to `num_entries` number of entries.

For example, this causes all entries to be displayed on your front page (which is probably a terrible idea):

```
py["truncate_frontpage"] = False
```

**year_indexes = (Optional) Default is True**
Whether or not to generate indexes per year.

For example:

```
py["year_indexes"] = True
```

### Compiling Configuration

If you are using compiling to deploy your Douglas blog you need to set some additional configuration variables in your `config.py` file, see *Deploy Douglas as a Compiled HTML Site*.

## Writing Entries

### Categories

Writing entries in Douglas is fairly straightforward. Each entry is a single text file located somewhere in the directory tree of your datadir. The directory that the entry is in is the category the entry is "filed under".

For example, if my datadir was `/home/joe/myblog/entries` and I stored an entry named `firstpost.txt` in `/home/joe/myblog/entries/status` then the category for my entry would be `/status`.

**Warning:** A warning about category names:

Be careful when you create your categories—be sure to use characters that are appropriate in directory names for the file system you're using.

**Note:** Categories are NOT the same thing as tags. An entry can only belong to ONE category. If that's not what you want, you should write or install a tags plugin.

Don't worry about making sure you have all the categories you need up front—you can add them as you need them.

### The format of an entry

Douglas entries consist of three parts: the title, the metadata, and then the body of the entry. The first line is the title of the entry. Then comes zero or more lines of metadata. After the metadata comes the body of the entry.

#### Title

The title consists of a single line of plain text. You can have whatever characters you like in the title of your entry. The title doesn't have to be the same as the entry file name.

### Metadata

The metadata section is between the title and the body of the entry. It consists of a series of lines that start with the hash mark #, then a metadata variable name, then a space, then the value of the metadata item.

Example of metadata lines:

```
#mood bored
#music The Doors - Greatest Hits Vol 1
```

The metadata variables set in the metadata section of the entry are available in your story template. So for the above example, the template variable `$(mood)` would be filled in with `bored` and `$*music)` would be filled in with `The Doors - Greatest Hits Vol 1`.

---

**Note:** Metadata is not collected in a multi-dict. If you include two pieces of metadata with the same key, the second one will overwrite the first one.

Example:

```
#mood bored
#mood happy
```

will result in `'mood'` –> `'happy'` in the metadata.

---

---

**Note:** You can provide metadata keys with no value. If you do this, then the default value is `'1'`. This seems a bit weird, but it makes it easier for plugin developers to use these as flags.

---

### Body

The body of the entry is written in HTML and comprises the rest of the entry file.

### Examples

Here's an example first post entry with a title and a body:

```
This is my first post!
<p>
  This is the body of the first post to my blog.
</p>
```

Here's a more complex example with a title and a body:

```
The rain in Spain....
<p>
  The rain
</p>
<p align="center">
  in Spain
</p>
<p align="right">
  is <font color="ff0000">mainly</font> on the plain.
</p>
```

Here's an example of a post with title, metadata, and a body:

```
The rain in Spain....
#mood bored
#music The Doors - Greatest Hits Vol 1
<p>
  The rain
</p>
<p align="center">
  in Spain
</p>
<p align="right">
  is <font color="ff0000">mainly</font> on the plain.
</p>
```

## Posting date

The posting date of the entry file is the modification time (also known as mtime) of the file itself as stored by your file system. Every time you go to edit an entry, it changes the modification time. You can see this in the following example of output:

```
willg ~/blog/entries/blosxom/site: vi testpost.txt                    [1]
willg ~/blog/entries/blosxom/site: ls -l
total 16
-rw-r--r--  1 willg willg 764 Jul 20  2003 minoradjustments.txt
-rw-r--r--  1 willg willg 524 Jul 24  2003 moreminoradjustments.txt
-rw-r--r--  1 willg willg 284 Aug 15  2004 nomorecalendar.txt
-rw-r--r--  1 willg willg  59 Mar 21 16:30 testpost.txt               [2]
willg ~/blog/entries/blosxom/site: vi testpost.txt                    [3]
willg ~/blog/entries/blosxom/site: ls -l
total 16
-rw-r--r--  1 willg willg 764 Jul 20  2003 minoradjustments.txt
-rw-r--r--  1 willg willg 524 Jul 24  2003 moreminoradjustments.txt
-rw-r--r--  1 willg willg 284 Aug 15  2004 nomorecalendar.txt
-rw-r--r--  1 willg willg  59 Mar 21 16:34 testpost.txt               [4]
```

1. I create the blog entry `testpost.txt` using `vi` (vi is a text editor). The mtime of the file will be the time I last save the file and exit out of vi.
2. Note that the mtime on the file is `Mar 21 16:30`. That's when I last saved the blog entry and exited out of vi.
3. I discover that I made a spelling mistake in my entry... So I edit it again in vi and fix the mistake. The mtime of the entry has now changed!
4. Now the mtime of the file is `Mar 21 16:34`. This is the time that will show up in my blog as the posting date.

**Warning:** A warning about mtimes:

There are some issues with this method for storing the posting date.

First, if you ever change the blog entry, the mtime will change as well. That makes updating blog entries very difficult down the line.

Second, if you move files around (backup/restore, changing the category structure, ...), you need to make sure you do so in a way that maintains the file's mtime.

### Entry parsers

Douglas supports one format for entry files by default. This format is the same format that blosxom uses and is described in previous sections.

A sample blog entry could look like this:

```
First post
<p>
  Here's the body of my first post.
</p>
```

Some people don't like writing in HTML. Other people use their entries in other places, so they need a different markup format. Some folks write a lot of material in a non-HTML markup format and would like to use that same format for blog entries. These are all very valid reasons to want to use other markup formats.

Douglas allows you to install entry parser plugins which are Douglas plugins that implement an entry parser. These entry parser plugins allow you to use other markup formats. Check the Plugin Registry on the website for other available entry parsers. Douglas comes with a restructured text entry parser.

If you don't see your favorite markup format represented, try looking at the code for other entry parsers and implement it yourself. If you need help, please ask on the douglas-devel mailing list or on IRC. Details for both of these are on the website.

Additionally, you're not locked into using a single markup across your blog. You can use any markup for an entry that you have an entry parser for.

## Renderers, Themes and Templates

### Summary

This chapter covers renderers, themes and templates.

### Renderers

Douglas supports multiple renderers and comes with two by default: debug and Jinja2.

#### debug

The debug renderer outputs your blog in a form that makes it easy to see the data generated when handling a Douglas request. This is useful for debugging plugins, working on Jinja2 themes and templates, and probably other things as well.

To set Douglas to use the debug renderer, do this in your `config.py` file:

```
py["renderer"] = "debug"
```

#### jinjarenderer

This is the default renderer and it renders the blog using Jinja2. This renderer lets you specify how your blog is rendered using themes which are composed of Jinja2 templates. This is described later in this chapter.

It's the default renderer, so if you want to use it, you don't have to do anything special.

### Other renderers

If you want your blog rendered by a different renderer, you'll need to install a plugin that implements the `renderer` callback or write your own.

## Themes and templates

The jinjarenderer lets you specify how your blog should be rendered with themes and templates.

A theme consists of at least:

- a `content_type` file which has the mimetype of the output being rendered (e.g. `text/html`)
- an `entry.<themename>` Jinja2 template file which is used when rendering a page with a single entry
- an `entry_list.<themename>` Jinja2 template file which is used when rendering a page with a bunch of entries (e.g. category list, date archive list, front page, ...)

Plugins may require additional templates. See each plugin's documentation for details.

### Example blog

Joe has this set in his `config.py` file:

```
py["themedir"] = "/home/joe/blog/themes/"
```

Joe's blog directory structure looks like this:

```
/home/joe/blog/
    |- entries/              <-- datadir
    |  |- work/              <-- work category of entries
    |  |- home/              <-- home category of entries
    |
    |- themes/
       |- html/              <-- html theme
       |  |- content_type
       |  |- entry.html
       |  |- entry_list.html
       |
       |- rss/               <-- rss theme
          |- content_type
          |- entry.rss
          |- entry_list.rss
```

---

**Note:** There's some redundancy between the theme named directory and the theme in the extension. Having the theme in the extension makes it more likely your editor will use the right syntax highlighting. So that's helpful. Having themes in separate directories means that if you have a bunch of files, they don't overlap and get all confuzzled. That's helpful, too.

However, this gets a bit irritating when you go to rename a theme and have to rename the directory as well as the extensions of all the files.

However however, given that the file extensions triggers syntax highlighting in editors, I suspect this won't happen often and that themes will be a .tar.gz file consisting of an `html/` directory with `*.html` files in it.

---

## Template writing tips

We're using Jinja2, so we reference variables using Jinja2 syntax and we can use Jinja2 blocks, built-in functions and built-in filters.

This prints a variable:

```
{{ foo }}
```

You can iterate through a list:

```
{% for entry in content %}
    {{ entry.title }}
    ...
{% endfor %}
```

Douglas has autoescaping set, so if the variable you're printing is HTML and you know that it's safe, you can use the `safe` filter:

```
{{ entry.body|safe }}
```

Templates can inherit from other templates. It's probably the case you want to have a base layout template that defines the common parts of your site, then have the entry or entry-list specific stuff in those templates.

To inherit from another template, use the `extends` tag:

```
{% extends "filename.ext" %}
```

In the "super template" you can define blocks and override those blocks in the "sub templates".

See the included html theme for an example.

**See also:**

http://jinja.pocoo.org/

http://jinja.pocoo.org/docs/templates/

http://jinja.pocoo.org/docs/templates/#template-inheritance

## Template variables

This is the list of variables that are available to your templates. Templates contain variables that are expanded when the template is rendered. Plugins may add additional variables—refer to plugin documentation for a list of which variables they add and in which templates they're available.

### Getting a complete list of variables

To get a complete list of what variables are available in your blog, use the debug renderer by changing the value of the `renderer` property in your `config.py` file to `debug` like this:

```
py["renderer"] = "debug"
```

That will tell you all kinds of stuff about the data structures involved in the request. Don't forget to change it back when you're done!

### Variables from config.py

Anything in your `config.py` file is a variable available to all of your templates. For example, these standard properties in your `config.py` file are available:

- `blog_description`
- `blog_title`
- `blog_language`
- `blog_encoding`
- `blog_author`
- `blog_email`
- `base_url`
- `static_url`
- ...

Additionally, any other properties you set in `config.py` are available in your templates. If you wanted to create a `blog_images` variable holding the base url of the directory with all your images in it:

```
py["blog_images"] = "http://example.com/~joe/images/"
```

to your `config.py` file and it would be available in all your templates.

### Calculated template variables

These template variables are available to all templates as well. They are calculated based on the request.

**`root_datadir`** The root datadir of this page?

Example: `/home/subtle/blosxom/weblogs/tools/douglas`

**`url`** The PATH_INFO to this page.

Example: `douglas/weblogs/tools/douglas`

**`theme`** The theme that's being used to render this page.

Example: `html`

**`latest_date`** The date of the most recent entry that is going to be rendered.

Example: `Tue, 15 Nov 2005`

**`latest_w3cdate`** The date of the most recent entry that is going to be rendered in w3cdate format.

Example: `2005-11-13T17:50:02Z`

**`latest_rfc822date`** The date of the most recent entry that is going to show in RFC 822 format.

Example: `Sun, 13 Nov 2005 17:50 GMT`

**`pi_yr`** The four-digit year if the request indicated a year.

Example: `2002`

**`pi_mo`** The month name if the request indicated a month.

Example: `Sep`

**pi_da** The day of the month if the request indicated a day of the month.

Example: `15`

**pi_bl** The entry the user requested to see if the request indicated a specific entry.

Example: `weblogs/tools/douglas`

**douglas_version** The version number and release date of the douglas version you're using.

Example: `1.2 3/25/2005`

### Variables available in the content entries

These template variables are available in the entries.

**title** The title of the entry.

Example: `First Post!`

**body** The text of the entry in HTML.

Example: `<p>This is my first post!</p>`

**filename** The absolute path of the blog entry file on the file system.

Example: `/home/subtle/blosxom/weblogs/tools/douglas/firstpost.txt`

**file_path** The filename and extension of the file that the entry is stored in.

Example: `firstpost.txt`

**basename** The filename without directory or file extension.

Example: `firstpost`

**absolute_path** The category/path of the entry (from the perspective of the url).

Example: `weblogs/tools/douglas`

**path** The category/path of the entry.

Example: `weblogs/tools/douglas`

**tb_id** The trackback id of the entry.

Example: `_firstpost`

**yr** The four-digit year of the mtime of this entry.

Example: `2004`

**mo** The month abbreviation of the mtime of this entry.

Example: `Jan`

**mo_num** The zero-padded month number of the mtime of this entry.

Example: `01`

**ti** The 24-hour hour and minute of the mtime of this entry.

Example: `16:40`

**date** The date string of the mtime of this entry.

Example: `Sun, 23 May 2004`

**w3cdate** The date in w3cdate format of the mtime of this entry.

Example: `2005-11-13T17:50:02Z`

**rfc822date** The date in RFC 822 format of the mtime of this entry.

Example: `Sun, 13 Nov 2005 17:50 GMT`

**fulltime** The date in YYYYMMDDHHMMSS format of the mtime of this entry.

Example: `20040523164000`

**timetuple** The time tuple (year, month, month-day, hour, minute, second, week-day, year-day, isdst) of the mtime of this entry.

Example: `(2004, 5, 23, 16, 40, 0, 6, 144, 1)`

**mtime** The mtime of this entry measured in seconds since the epoch.

Example: `1085348400.0`

**dw** The day of the week of the mtime of this entry.

Example: `Sunday`

**da** The day of the month of the mtime of this entry.

Example: `23`

Also, any variables created by plugins that are entry-centric and any variables that come from metadata in the entry are available. See those sections in this document for more details.

### Template variables from plugins

Many plugins will create additional variables that are available in templates. Refer to the documentation of the plugins that you have installed to see what variables are available and what they do.

### Template variables from entry metadata

You can add metadata to your entries on an individual basis and this metadata is available to your story templates.

For example, if I had a blog entry like this:

```
First Post!
#mood happy
#music The Doors - Break on Through to the Other Side
<p>
  This is the first post to my new Douglas blog.  I've
  also got two metadata items in it which will be available
  as variables!
</p>
```

You'll have two variables `$mood` and `$music` that will also be available in your story templates.

## Invoking a theme

The theme for a given page is specified in the extension of the file being requested. For example:

- `http://example.com/` - brings up the index in the default theme which is "html"
- `http://example.com/index.html` - brings up the index in the "html" theme

- `http://example.com/index.rss` - brings up the index in the "rss" theme (which by default is RSS 0.9.1)
- `http://example.com/2004/05/index.joy` - brings up the index for May of 2004 in the "joy" theme

Additionally, you can specify the theme by adding a `theme` variable in the query-string. Examples:

- `http://example.com/` - brings up the index in the default theme which is "html"
- `http://example.com/?theme=rss` - brings up the index in the "rss" theme
- `http://example.com/2004/05/index?theme=joy` - brings up the index for May of 2004 in the "joy" theme

### Setting default theme

You can change the default theme from `html` to some other theme in your `config.py` file with the `default_theme` property:

```
py["default_theme"] = "joy"
```

Doing this will set the default theme to use when the URI the user has used doesn't specify which theme to use.

This url doesn't specify the theme to use, so it will be rendered with the default theme:

```
http://example.com/cgi-bin/douglas.cgi/2005/03
```

This url specifies the theme, so it will be rendered with that theme:

```
http://example.com/cgi-bin/douglas.cgi/2005/03/?theme=html
```

### Order of operations to figure out which theme to use

We know that you can specify the default theme to use in the `config.py` file with the `default_theme` property. We know that the user can specify which theme to use by the file extension of the URI. We also know that the user can specify which theme to use by using the `flav` variable in the query string.

The order in which we figure out which theme to use is this:

1. Look at the URI extension: if the URI has one, then we use that.
2. Look at the `theme` querystring variable: if there is one, then we use that.
3. Look at the `default_theme` property in the `config.py` file which defaults to `html`.

## Plugins

Douglas allows you to extend and augment its base functionality with plugins. Plugins allow you to:

- create additional variables
- provide additional entry parsers, renderers, post-formatters, and pre-formatters
- create new output data types
- pull information from other non-blog sources
- create images

- and a variety of other things

Plugins hook into Douglas using callbacks which allow plugins to handle, transform, override and otherwise affect Douglas's behavior.

### Setting Douglas up to use plugins

There are two properties in your `config.py` file that affect the behavior for loading plugins: `plugin_dirs` and `load_plugins`. There's more documentation on these in *Configuration variables*.

### Finding plugins

Douglas comes with a core set of plugins. Documentation for these plugins is in *Part 2: Core plugin documentation*.

Additionally, you can write your own plugins.

### Installing plugins

When you're installing a plugin, refer to its documentation. The documentation could be in a `README` file, but more commonly it's in the plugin code itself at the top of the file. This documentation should tell you how to install the plugin, what template variables the plugin exposes, how to invoke the plugin, how to get in touch with the author should you find bugs or need help, and any additional things you should know about.

Most plugins should have a pretty easy installation method. You should be able to copy the plugin into the directory defined in your `config.py` file in the `plugin_dirs` property. Then there might be some additional properties you'll have to set in your `config.py` file to define the plugin's behavior. That should be about it. On some occasions, you may have to change the code in the plugin itself to meet your specific needs.

### Writing your own plugins

You may find that you desire functionality and there is no plugin that anyone knows about that performs that functionality. It's probably best at this point for you to ask someone to write the plugin you need or write it yourself.

Douglas plugins are fairly easy to write and can cover a lot of really different functionality. The best way to learn how to write Douglas plugins is to read through the plugins in the plugin registry. Many of them are well written and may provide insight as to how to solve your specific problem.

If you plan on writing your own plugin, check out *Writing Plugins*.

## Authors

### Douglas

Douglas is a rewrite of Pyblosxom by Will Kahn-Greene.

- Will Kahn-Greene
- Brian Fife

## Pyblosxom

Pyblosxom was originally written by Wari Wahab and is now maintained by Akai Kitsune.

Over the years, Pyblosxom has had many contributors who have helped make the project what it is today (in no particular order):

- Will Kahn-Greene
- Martin Kraft
- Joerg Wendland
- Enrico Zini
- Gpal. V
- Nathan Gray
- Alexandre Patry
- Brian Warner
- Ryan Thiessen
- Dewayne Christensen
- Joe Gregorio
- James Henstridge
- Myers Carpenter
- Scott C.
- Axel Kollmorgen
- Thenault Sylvain
- Russell Nelson
- IWS
- Joseph Reagle
- Tollef Fog Heen
- Colin Walters
- Norbert Tretkowski
- David Stanek
- FX
- Zoom Quiet
- Matej Cepl
- Andrew Kuchling
- Dieter Plaetinck
- Jordi Mallach
- Marius Gedminas
- Mikko Värri
- Robert Wall

- Ryan Barrett
- Sebastian Spaeth
- Steven Armstrong
- Ted Leung
- Wari Wahab
- Weakish Jakukyo
- Weakish Jiang
- Doug Ransom
- Abe Fettig
- Benjamin Mako Hill
- David Pashley
- David Geller
- Roberto De Almeida
- Antonio “Willy” Malara
- Sean Whitton
- Nicholas Tollervey

Many thanks for all their work and efforts!

CHAPTER 4

# Part 2: Core plugin documentation

Documentation for plugins that come with Douglas.

## archives - Builds month/year-based archives li...

### Summary

Walks through your blog root figuring out all the available monthly archives in your blogs. It generates html with this information and stores it in the `$(archivelinks)` variable which you can use in your head and foot templates.

### Install

This plugin comes with douglas. To install, do the following:

1. Add `douglas.plugins.pyarchives` to the `load_plugins` list in your `config.py` file.
2. Configure using the following configuration variables.

`archive_template`

Let's you change the format of the output for an archive link.

For example:

```
py['archive_template'] = ('<li><a href="%(base_url)s/%(Y)s/%(b)s">'
                          '%(m)s/%(y)s</a></li>')
```

This displays the archives as list items, with a month number, then a slash, then the year number.

The formatting variables available in the `archive_template` are:

```
b         'Jun'
m         '6'
```

```
Y         '1978'
y         '78'
```

These work the same as `time.strftime` in python.

Additionally, you can use variables from config and data.

---

**Note:** The syntax used here is the Python string formatting syntax—not the douglas template rendering syntax!

---

### Usage

Add `$(archivelinks)` to your head and/or foot templates.

### License

Plugin is distributed under license: MIT

## categories - Builds a list of categories....

### Summary

Walks through your blog root figuring out all the categories you have and how many entries are in each category. It generates html with this information and stores it in the `$(categorylinks)` variable which you can use in your head or foot templates.

### Install

This plugin comes with douglas. To install, do the following:

1. Add `douglas.plugins.pycategories` to the `load_plugins` list in your `config.py` file.

### Configuration

There is no configuration.

### Usage

Categories plugin provides an HTML version of the categories in a list form. You can use it in your template like this:

```
{{ categories.as_list()|safe }}
```

Alternatively, you can build the categories HTML yourself:

```
{% for cat, count in categories.categorydata %}
    ....
{% endfor %}
```

### License

Plugin is distributed under license: MIT

## draft_folder - Draft folder...

### Summary

Enables drafts for your blog.

### Install and Configure

1. Add `douglas.plugins.draft_folder` to the `load_plugins` list in your `config.py` file.
2. Set `py["draftdir"]` to the directory where your draft entries will be.

   This can't be a subdirectory of your datadir.

   Make sure to create this directory, too.
3. (optional) Set `py["draft_trigger"]` in your `config.py` file to the url path you want to show drafts in. This defaults to `drafts`.

That's it!

### License

Plugin is distributed under license: MIT

## ignore_future - Ignores entries in the future....

### Summary

Prevents blog entries published in the future from showing up on the blog.

### Install

Add `douglas.plugins.ignore_future` to the `load_plugins` list in your `config.py` file.

### License

Plugin is distributed under license: MIT

# no_old_comments - Prevent comments on entries older t...

## Summary

This plugin implements the `comment_reject` callback of the comments plugin.

If someone tries to comment on an entry that's older than 28 days, the comment is rejected.

## Install

Requires the `comments` plugin.

This plugin comes with Douglas. To install, do the following:

1. Add `Douglas.plugins.no_old_comments` to the `load_plugins` list in your `config.py` file.

## Revisions

1.0 - August 5th 2006: First released.

## License

Plugin is distributed under license: Public Domain

# pages - Allows you to include non-blog-entr...

## Summary

Blogs don't always consist solely of blog entries. Sometimes you want to add other content to your blog that's not a blog entry. For example, an "about this blog" page or a page covering a list of your development projects.

This plugin allows you to have pages served by douglas that aren't blog entries.

Additionally, this plugin allows you to have a non-blog-entry front page. This makes it easier to use douglas to run your entire website.

## Install

This plugin comes with douglas. To install, do the following:

1. add `douglas.plugins.pages` to the `load_plugins` list in your `config.py` file.
2. configure the plugin using the configuration variables below

`pagesdir`

This is the directory that holds the pages files.

For example, if you wanted your pages in `/home/foo/blog/pages/`, then you would set it to:

```
py["pagesdir"] = "/home/foo/blog/pages/"
```

If you have `blogdir` defined in your `config.py` file which holds your `datadir` and `themedir` directories, then you could set it to:

```
py["pagesdir"] = os.path.join(blogdir, "pages")
```

`pages_trigger` (optional)

Defaults to `pages`.

This is the url trigger that causes the pages plugin to look for pages.

py["pages_trigger"] = "pages"

`pages_frontpage` (optional)

Defaults to False.

If set to True, then pages will show the `frontpage` page for the front page.

This requires you to have a `frontpage` file in your pages directory. The extension for this file works the same way as blog entries. So if your blog entries end in `.txt`, then you would need a `frontpage.txt` file.

Example:

```
py["pages_frontpage"] = True
```

## Usage

Pages looks for urls that start with the trigger `pages_trigger` value as set in your `config.py` file. For example, if your `pages_trigger` was `pages`, then it would look for urls like this:

```
/pages/blah
/pages/blah.html
```

and pulls up the file `blah.txt`[1] which is located in the path specified in the config file as `pagesdir`.

If the file is not there, it kicks up a 404.

## Template

pages formats the page using the `pages` template. So you need a `pages` template in the themes that you want these pages to be rendered in. If you want your pages rendered exactly like an entry, just extend the `entry` template.

## Python code blocks

pages handles evaluating python code blocks. Enclose python code in `<%` and `%>`. The assumption is that only you can edit your pages files, so there are no restrictions (security or otherwise).

For example:

```
<%
print "testing"
%>
```

[1] The file ending (the `.txt` part) can be any file ending that's valid for entries on your blog. For example, if you have the textile entryparser installed, then `.txtl` is also a valid file ending.

```
<%
x = { "apple": 5, "banana": 6, "pear": 4 }
for mem in x.keys():
   print "<li>%s - %s</li>" % (mem, x[mem])
%>
```

The request object is available in python code blocks. Reference it by `request`. Example:

```
<%
config = request.get_configuration()
print "your datadir is: %s" % config["datadir"]
%>
```

### License

Plugin is distributed under license: MIT

## paginate - Allows navigation by page for index...

### Summary

Plugin for breaking up long index pages with many entries into pages.

### Install

This plugin comes with douglas. To install, do the following:

1. Add `douglas.plugins.paginate` to your `load_plugins` list variable in your `config.py` file.

   Make sure it's the first plugin in the `load_plugins` list so that it has a chance to operate on the entry list before other plugins.

2. (optional) Add some configuration to your `config.py` file.

### Usage

Add the following blurb where you want page navigation to your `entry_list` template:

```
{% if pager is defined %}
  {{ pager.as_list()|safe }}
{% endif %}
```

which generates HTML like this:

```
[1] 2 3 4 5 6 7 8 9 ... >>
```

Or:

```
{% if pager is defined %}
  {{ pager.as_span()|safe }}
{% endif %}
```

which generates HTMl like this:

```
Page 1 of 4 >>
```

You can also do your own pagination. The `pager` instance exposes the following helpful bits:

- `number` - the page number being shown
- `has_next()` - True if there's a next page
- `has_previous()` - True if there's a previous page
- `link(pageno)` - Builds the url for the specified page

### Configuration variables

`paginate_previous_text`

Defaults to "<<".

This is the text for the "previous page" link.

`paginate_next_text`

Defaults to ">>".

This is the text for the "next page" link.

### Note about compiling

This plugin works fine with compiling, but the urls look different. Instead of adding a `page=4` kind of thing to the querystring, this adds it to the url.

For example, say your front page was `/index.html` and you had 5 pages of entries. Then the urls would look like this:

```
/index.html           first page
/index_page2.html     second page
/index_page3.html     third page
...
```

### License

Plugin is distributed under license: MIT

## published_date - Maintain published date in file met...

### Summary

This takes a #published date/time stamp in the entry and returns that as the mtime.

Example entry:

```
My first post!
#published 2008-01-01 12:20:22
<p>
  This is my first post!
</p>
```

returns an mtime of 01-01-2008 at 12:20:22.

### Install

Add `douglas.plugins.published_date` to the `load_plugins` list of your `config.py` file.

### License

Plugin is distributed under license: MIT

## rst_parser - restructured text support for blog ...

### Summary

A reStructuredText entry formatter for douglas. reStructuredText is part of the docutils project (http://docutils.sourceforge.net/). To use, you need a *recent* version of docutils. A development snapshot (http://docutils.sourceforge.net/#development-snapshots) will work fine.

### Install

This plugin comes with douglas. To install, do the following:

1. Add `douglas.plugins.rst_parser` to the `load_plugins` list in your `config.py` file.
2. Install docutils. Instructions are at http://docutils.sourceforge.net/

### Configuration

There's two optional configuration parameter you can for additional control over the rendered HTML:

```
# To set the starting level for the rendered heading elements.
# 1 is the default.
py['reST_initial_header_level'] = 1

# Enable or disable the promotion of a lone top-level section title to
# document title (and subsequent section title to document subtitle
# promotion); disabled by default.
py['reST_transform_doctitle'] = 0
```

**Note:** If you're not seeing headings that you think should be there, try changing the `reST_initial_header_level` property to 0.

### Usage

Blog entries with a `.rst` extension will be parsed as reStructuredText.

Blog entries can have a summary. Insert a break directive at the point where the summary should end. For example:

```
First part of my blog entry....

.. break::

Second part of my blog entry after the fold.
```

Some entries don't have a summary attribute, so if you're going to show the summary, you need to make sure it's defined first.

For example, in your entry_list template, you could show the summary like this:

```
{% if entry.summary is defined %}
  {{ entry.summary|safe }}
  <p><a href="{{ entry.url }}">Read more...</a></p>
{% else %}
  {{ entry.body|safe }}
{% endif %}
```

### License

Plugin is distributed under license: MIT

## tags - Tags plugin...

### Summary

This plugin allows you to specify the tags your entry has in the metadata of the entry. It adds a new command to douglas-cmd to index all the tags data and store it in a file.

It creates a `TagManager` instance in the Jinja2 environment which you can use to iterate through and display tags data.

### Install

This plugin comes with douglas. To install, do the following:

1. Add `douglas.plugins.tags` to the `load_plugins` list in your `config.py` file.
2. Configure as documented below.

### Configuration

The following config properties define where the tags file is located, how tag metadata is formatted, and how tag lists triggered.

`tags_separator`

This defines the separator between tags in the metadata line. Defaults to ",".

After splitting on the separator, each individual tag is stripped of whitespace before and after the text.

For example:

```
Weather in Boston
#tags weather, boston
<p>
  The weather in Boston today is pretty nice.
</p>
```

returns tags `weather` and `boston`.

If the `tags_separator` is:

```
py["tags_separator"] = "::"
```

then tags could be declared in the entries like this:

```
Weather in Boston
#tags weather::boston
<p>
  The weather in Boston today is pretty nice.
</p>
```

`tags_filename`

This is the file that holds indexed tags data. Defaults to datadir + os.pardir + `tags.index`.

This file needs to be readable by the process that runs your blog. This file needs to be writable by the process that creates the index.

`tags_trigger`

This is the url trigger to indicate that the tags plugin should handle the file list based on the tag. Defaults to `tag`.

`truncate_tags`

If this is True, then tags index listings will get passed through the truncate callback. If this is False, then the tags index listing will not be truncated.

If you're using a paging plugin, then setting this to True will allow your tags index to be paged.

Example:

```
py["truncate_tags"] = True
```

Defaults to True.

## Usage in templates

The `TagManager` gets added to the context as `tags`. It has the following methods:

**`all_tags()`** Returns a list of (tag, tag_url, count) tuples.

You can iterate over this to render tag data for all the tags on your blog.

```
{{ tags.all_tags()|safe }}
```

**all_tags_div()** Generates HTML for a div of class `allTags` with `a` tags of class `tag` in it–one for each tag.

```
{{ tags.all_tags_div()|safe }}
```

**all_tags_cloud()** Generates HTML for a div of class `allTagsCloud` with `a` tags of class `tag` in it–one for each tag. The `a` tags also have one of `biggestTag`, `bigTag`, `mediumTag`, `smallTag`, or `smallestTag` depending on how "big" the tag should show up in the cloud.

```
{{ tags.all_tags_cloud()|safe }}
```

**entry_tags(entry)** Returns a list of (tag, tag_url) tuples for tags for the specified entry.

```
{% for tag in tags.entry_tags(entry) %}
  {{ tag }}
{% endfor %}
```

**entry_tags_span(entry)** Generates HTML for a span of class `entryTags` with `a` tags of class `tag` in it–one for each tag.

```
{{ tags.entry_tags_span(entry)|safe }}
```

**Note:** If you use functions that generate HTML in a Jinja2 template, you need to run them through the `|safe` filter. Otherwise the HTML will be escaped.

## Creating the tags index file

Run:

```
douglas-cmd buildtags
```

from the directory your `config.py` is in or:

```
douglas-cmd buildtags --config=/path/to/config/file
```

from anywhere.

This builds the tags index file that the tags plugin requires to generate tags-based bits for the request.

Until you rebuild the tags index file, the entry will not have its tags indexed. Thus you should either rebuild the tags file after writing or updating an entry or you should rebuild the tags file as a cron job.

**Note:** If you're compiling your blog, you need to build the tags index before you compile.

## Converting from categories to tags

This plugin has a command that goes through your entries and adds tag metadata based on the category. There are some caveats:

1. it assumes entries are in the blosxom format of title, then metadata, then the body.
2. it only operates on entries in the datadir.

It maintains the atime and mtime of the file. My suggestion is to back up your files (use tar or something that maintains file stats), then try it out and see how well it works, and figure out if that works or not.

To run the command do:

```
douglas-cmd categorytotags
```

from the directory your `config.py` is in or:

```
douglas-cmd categorytotags --config=/path/to/config/file
```

from anywhere.

### License

Plugin is distributed under license: MIT

## yeararchives - Builds year-based archives listing....

### Summary

Walks through your blog root figuring out all the available years for the archives list. Handles year-based indexes. Builds a list of years your blog has entries for which you can use in your template.

### Install

This plugin comes with Douglas. To install, do the following:

1. Add `douglas.plugins.yeararchives` to the `load_plugins` list in your `config.py` file.

### Usage

Add:

```
{{ yeararchives.as_list()|safe }}
```

to the appropriate place in your template.

When the user clicks on one of the year links (e.g. `http://example.com/2004/`), then yeararchives will display a summary page for that year.

### License

Plugin is distributed under license: MIT

CHAPTER 5

# Part 3: Developer documentation

Documentation anyone interested in hacking on Douglas, writing plugins, or things of that ilk.

## Contributing

This covers the basics you need to know for contributing to Douglas.



### Status

**December 27th, 2013**

I'm rewriting Pyblosxom fixing a lot of problems I had with it. This project is in crazy flux right now. I don't expect anyone to want to help at this stage. If you want to help anyways, see the issues in the issue tracker for what's in the queue of things to fix.

## How to clone the project

Douglas is on GitHub.

### If you have a GitHub account [Recommended]

This is the ideal way to contribute because GitHub makes things simple and convenient.

Go to the project page (https://github.com/willkg/douglas) and click on "Fork" at the top right on the screen. GitHub will create a copy of the Douglas repository in your account for you to work on.

Create a new branch off of master for any new work that you do.

When you want to send it upstream, create a pull request.

If you need help with this process, see the GitHub documentation.

### If you don't have a GitHub account

Clone the project using git:

```
$ git clone https://github.com/willkg/douglas.git
```

Set `user.name` and `user.email` git configuration:

```
$ git config user.name "your name"
$ git config user.email "your@email.address"
```

Create a new branch off of master for any new work that you do.

When you want to send it upstream, do:

```
$ git format-patch --stdout origin/master > NAME_OF_PATCH_FILE.patch
```

where `NAME_OF_PATCH_FILE` is a nice name that's short and descriptive of what the patch holds and master should be replaced with your branch name

Then attach that `.patch` file and send it to douglas-devel mailing list.

## Installing for hacking

1. Clone the project into a directory
2. Create a virtual environment and activate it
3. Install Douglas into your virtual environment in a way that's suitable for hacking:

   ```
   $ pip install -e .
   ```

4. Install development requirements:

   ```
   $ pip install -r requirements-dev.txt
   ```

Create a new blog:

```
$ douglas-cmd create [<dir>]
```

Generate "sample" entries:

```
$ douglas-cmd generate [<num_entries>]
```

Douglas comes with

## Code conventions

Follow PEP-8.

Best to run pyflakes and pep8 over your code.

Don't use l as a variable name.

## Tests

In the douglas git repository, there are two big things that have test suites:

1. the Douglas core code
2. the plugins that are in `douglas/plugins/`

Please add tests for changes you make. In general, it's best to write a test, verify that it fails, then fix the code which should make the test pass.

Tests go in `douglas/tests/`.

We use nose because it's super.

Run the tests by:

```
$ nosetests
```

The `douglas.tests` package defines helper functions, classes, and other things to make testing easier.

Writing tests is pretty easy:

1. create a file in `douglas/tests/` with a filename that starts with `test_` and ends with `.py`.
2. at the top, do:

   ```
   from douglas.tests import UnitTestBase
   ```

3. create a subclass of `UnitTestBase`
4. write some tests using pretty standard unittest/nose stuff

See `douglas/tests/` for examples testing the core as well as core plugins.

## Documentation

New features should come with appropriate changes to the documentation.

Documentation is in the `docs/` directory, written using reStructuredText, and built with Sphinx.

# Douglas Architecture

This covers the architecture for Douglas to help write plugins.



## Summary

Douglas is a blog system where each entry is a file on your file system. This allows you to use any text editor for editing blog entries.

Douglas has a callback/handler system that allows plugins to augment, override or add new functionality to Douglas allowing you to adjust Douglas to your needs.

This chapter covers Douglas's architecture.

## Starting Douglas

FIXME - write this

## Handling requests

FIXME - write this

## Callbacks

FIXME - list all callbacks here

# Writing Plugins

## Summary

This chapter covers a bunch of useful things to know when writing Douglas plugins. This chapter, moreso than the rest of this manual, is very much a work in progress.

If you need help with plugin development, write up an issue in the issue tracker.

FIXME - this needs more work

## Things that all plugins should have

This section covers things that all plugins should have. This makes plugins easier to distribute, maintain, update, and easier for users to use them.

### Example

Here's a really short example plugin named `ignore_future.py`:

```
"""
Summary
=======

Prevents blog entries published in the future from showing up on
the blog.


Install
=======

Add ``douglas.plugins.ignore_future`` to the ``load_plugins`` list in
your ``config.py`` file.

"""

__description__ = "Ignores entries in the future."
__category__ = "content"
__license__ = "MIT"

import time

from douglas.tools import filestat


def cb_entries(args):
    cfg = args['config']
    entry_files = args['entry_files']

    now = time.time()

    def check_mtime(cfg, now, path):
        mtime = time.mktime(filestat(cfg, path))
        return mtime < now

    entry_files = [path for path in entry_files
                   if check_mtime(cfg, now, path)]
    args['entry_files'] = entry_files

    return args
```

### Name

All plugins need a good name that's unique so that your plugin doesn't get confused with other plugins. Additionally, the filename for your plugin needs to be unique.

**Warning:** Make sure the filename for your plugin is unique! Douglas imports your plugin using Python import machinery which means that if your plugin has the same name as a package on your system, then depending on how `sys.path` is set up, Douglas may load the package on your system and NOT your plugin.

If you think this might be happening to you, do `douglas-cmd test` and it'll tell you the paths of what it's loading.

### Documentation

All plugins should have a docstring at the top of the file that explains what the plugin does, how to install it, how to configure it and how to use it.

### Metadata

All plugins should have the following module-level variables defined in them just after the docstring:

- `__description__` - This is a one-sentence description of what your plugin does.
- `__license__` - The license this plugin is distributed under.
- `__category__` - (Optional) A one-word category for the plugin. You only need to include this if you're planning to create a pull request to add this plugin to Douglas core plugins.
- `__url__` (Optional) The canonical url where information about this plugin is. GitHub repository, web-site, author's blog entry—whatever. Users will use this url to figure out whether their copy of the plugin is up-to-date, contact the author with issues, etc.

### Configuration, installation and verification

After that, you could have a `verify_installation` function that verifies that the plugin is configured correctly. This helps when your plugin has complex configuration since you can walk the user through misconfiguration issues rather than the user see your plugin fail inexplicably.

If your plugin doesn't require much configuration or the configuration is trivial, feel free to skip this.

Here's an example:

```
def verify_installation(request):
    cfg = request.get_configuration()

    if 'important_key' not in cfg:
        print 'You are missing important_key in your configuration!'
        return False
    return True
```

Return `False` if it fails verification.

Return `True` if it passes verification.

## How to log to the log file

The user can configure logging in their `config.py` file. If it's not configured, then logging is at the `error` level and is piped to stdout.

Douglas uses the Python logging module.

## How to implement a callback

If you want to implement a callback, you add a function corresponding to the callback name to your plugin module. For example, if you wanted to modify the Request object just before rendering, you'd implement `cb_prepare` like this:

```
def cb_prepare(args):
    pass
```

Obviously, since we have `pass` we're not actually doing anything here, but when the user sends a request and Douglas handles it, this function in your plugin will get called when Douglas runs the prepare callback.

Each callback passes in arguments through a single dictionary. Each callback passes in different arguments and expects different return values. Check the doc:*dev_architecture <architecture>* chapter for a list of all the callbacks that are available, their arguments, and how they work.

## Writing an entryparser

FIXME - write this

## Writing a renderer

FIXME - write this

## Writing a plugin that adds a commandline command

The `douglas-cmd` command allows for plugin-defined commands. This allows your plugin to do maintenance tasks (updating an index, statistics, generating content, ...) and allows the user to schedule command execution through cron or some similar system.

To write a new command, you must:

1. implement the `commandline` callback which adds the command, handler, and command summary
2. implement the command function

For example, this adds a command to print command line arguments:

```
def printargs(command, argv):
    print argv
    return 0

def cb_commandline(args):
    args["printargs"] = (printargs, "prints arguments")
    return args
```

Executing the command looks like this:

```
% douglas-cmd printargs a b c
douglas-cmd version 0.1
a b c
```

For examples, see `douglas/cmdline.py` and `douglas/plugins/tags.py`.

# Code Documentation

## Introduction

This chapter covers important functions, methods and classes in Douglas. When in doubt, read the code.

## Douglas

**__version__**
Douglas version as a string. Conforms to PEP-386.

For example `"0.1"`.

## douglas.app

**class** `douglas.app.`**Douglas**(*config*, *environ*, *data=None*)
Main class for Douglas functionality. It handles initialization, defines default behavior, and also pushes the request through all the steps until the output is rendered and we're complete.

**cleanup**()
This cleans up Douglas after a run.

This should be called when Douglas has done everything it needs to do before exiting.

**get_request**()
Returns the Request object for this Douglas instance.

**get_response**()
Returns the Response object associated with this Request.

**initialize**()
The initialize step further initializes the Request by setting additional information in the `data` dict, registering plugins, and entryparsers.

**run**(*compiling=False*)
This is the main loop for Douglas. This method will run the handle callback to allow registered handlers to handle the request. If nothing handles the request, then we use the `default_blosxom_handler`.

**Parameters** **compiling** – True if Douglas should execute in compiling mode and False otherwise.

**run_collectstatic**()
Collects static files and copies them to compiledir

**run_compile**(*incremental=False*)
Compiles the blog into an HTML site.

This will go through all possible things in the blog and compile the blog to the `compiledir` specified in the config file.

This figures out all the possible `path_info` settings and calls `self.run()` a bazillion times saving each file.

**Parameters** **incremental** – Whether (True) or not (False) to compile incrementally. If we're incrementally compiling, then only the urls that are likely to have changed get re-compiled.

**run_render_one**(*url*, *headers*)
Renders a single page from the blog.

**Parameters**

- **url** – the url to render–this has to be relative to the base url for this blog.
- **headers** – True if you want headers to be rendered and False if not.

**class** douglas.app.**EnvDict**(*request*, *env*)

Wrapper arround a dict to provide a backwards compatible way to get the `form` with syntax as:

```
request.get_http()['form']
```

instead of:

```
request.get_form()
```

**class** douglas.app.**Request**(*config*, *environ*, *data*)

This class holds the Douglas request. It holds configuration information, HTTP/CGI information, and data that we calculate and transform over the course of execution.

There should be only one instance of this class floating around and it should get created by `douglas.cgi` and passed into the Douglas instance which will do further manipulation on the Request instance.

**add_configuration**(*newdict*)

Takes in a dict and adds/overrides values in the existing configuration dict with the new values.

**add_data**(*d*)

Takes in a dict and adds/overrides values in the existing data dict with the new values.

**add_http**(*d*)

Takes in a dict and adds/overrides values in the existing http dict with the new values.

**buffer_input_stream**()

Buffer the input stream in a StringIO instance. This is done to have a known/consistent way of accessing incomming data. For example the input stream passed by mod_python does not offer the same functionality as `sys.stdin`.

**get_configuration**()

Returns the *actual* configuration dict. The configuration dict holds values that the user sets in their `config.py` file.

Modifying the contents of the dict will affect all downstream processing.

**get_data**()

Returns the *actual* data dict. Holds run-time data which is created and transformed by douglas during execution.

Modifying the contents of the dict will affect all downstream processing.

**get_form**()

Returns the form data submitted by the client. The `form` instance is created only when requested to prevent overhead and unnecessary consumption of the input stream.

**Returns** a `cgi.FieldStorage` instance.

**get_http**()

Returns the *actual* http dict. Holds HTTP/CGI data derived from the environment of execution.

Modifying the contents of the dict will affect all downstream processing.

**get_response**()

Returns the Response for this request.

**get_theme**()

Returns the user-requested theme.

**set_response**(*response*)
Sets the Response object.

**class** douglas.app.**Response**(*request*)
Response class to handle all output related tasks in one place.

This class is basically a wrapper arround a StringIO instance. It also provides methods for managing http headers.

**add_header**(*key*, *value*)
Populates the HTTP header with lines of text. Sets the status code on this response object if the given argument list containes a 'Status' header.

Example:

```
>>> resp.add_header("Content-type", "text/plain")
>>> resp.add_header("Content-Length", "10500")
```

**Raises** **ValueError** – This happens when the parameters are not correct.

**get_headers**()
Returns the headers.

**get_status**()
Returns the status code and message of this response.

**send_body**(*out*)
Send the response body to the given output stream.

**Parameters** **out** – the file-like object to print the body to.

**send_headers**(*out*)
Send HTTP Headers to the given output stream.

**Note:** This prints the headers and then the \n\n that separates headers from the body.

**Parameters** **out** – The file-like object to print headers to.

**set_status**(*status*)
Sets the status code for this response. The status should be a valid HTTP response status.

Examples:

```
>>> resp.set_status("200 OK")
>>> resp.set_status("404 Not Found")
```

**Parameters** **status** – the status string.

douglas.app.**blosxom_entry_parser**(*filename*, *request*)
Parses a .txt entry file.

The first line becomes the title of the entry. Lines after the first line that start with # are metadata lines. After the metadata lines, the remaining lines are the body of the entry.

**Parameters**

- **filename** – a filename to extract data and metadata from
- **request** – a standard request object

**Returns** dict containing parsed data and meta data with the particular file (and plugin)

douglas.app.**blosxom_file_list_handler**(*args*)
This is the default handler for getting entries. It takes the request object in and figures out which entries based on the default behavior that we want to show and generates a list of EntryBase subclass objects which it returns.

**Parameters** **args** – dict containing the incoming Request object

**Returns** the content we want to render

douglas.app.**blosxom_handler**(*request*)
This is the default blosxom handler.

It calls the renderer callback to get a renderer. If there is no renderer, it uses the blosxom renderer.

It calls the pathinfo callback to process the path_info http variable.

It calls the filelist callback to build a list of entries to display.

It calls the prepare callback to do any additional preparation before rendering the entries.

Then it tells the renderer to render the entries.

**Parameters** **request** – the request object.

douglas.app.**blosxom_process_path_info**(*args*)
Process HTTP `PATH_INFO` for URI according to path specifications, fill in data dict accordingly.

The paths specification looks like this:

- `/foo.html` and `/cat/foo.html` - file foo.* in / and /cat
- `/cat` - category
- `/2002` - category (if that's a directory)
- `/2002` - year index
- `/2002/02` - year/month index
- `/2002/02/04` - year/month/day index

**Parameters** **args** – dict containing the incoming Request object

douglas.app.**blosxom_sort_list_handler**(*args*)
Sorts the list based on `_mtime` attribute such that most recently written entries are at the beginning of the list and oldest entries are at the end.

**Parameters** **args** – args dict with `request` object and `entry_list` list of entries

**Returns** the sorted `entry_list`

douglas.app.**blosxom_truncate_list_handler**(*args*)
If `config["num_entries"]` is not 0 and `data["truncate"]` is not 0, then this truncates `args["entry_list"]` by `config["num_entries"]`.

**Parameters** **args** – args dict with `request` object and `entry_list` list of entries

**Returns** the truncated `entry_list`.

douglas.app.**douglas_app_factory**(*global_config*, ***local_config*)
App factory for paste.

**Returns** WSGI application

douglas.app.**run_cgi**(*cfg*)
Executes Douglas as a CGI script.

## douglas.tools

**exception** douglas.tools.**ConfigSyntaxErrorException**
Thrown for `convert_configini_values` syntax errors.

douglas.tools.**convert_configini_values**(*configini*)
Takes a dict containing config.ini style keys and values, converts the values, and returns a new config dict.

**Parameters** **confini** – dict containing the config.ini style keys and values

**Raises** ***ConfigSyntaxErrorException*** – when there's a syntax error

**Returns** new config dict

douglas.tools.**create_entry**(*datadir*, *category*, *filename*, *mtime*, *title*, *metadata*, *body*)
Creates a new entry in the blog.

This is primarily used by the testing system, but it could be used by scripts and other tools.

**Parameters**

- **datadir** – the datadir
- **category** – the category the entry should go in
- **filename** – the name of the blog entry (filename and extension–no directory)
- **mtime** – the mtime (float) for the entry in seconds since the epoch
- **title** – the title for the entry
- **metadata** – dict of key/value metadata pairs
- **body** – the body of the entry

**Raises** **IOError** – if the datadir + category directory exists, but isn't a directory

douglas.tools.**escape_text**(*s*)
Takes in a string and converts:

- `&` to `&`
- `>` to `>`
- `<` to `<`
- `"` to `&quot;`
- `'` to `&#x27;`
- `/` to `&#x2F;`

Note: if `s` is `None`, then we return `None`.

```
>>> escape_text(None)
None
>>> escape_text("")
""
>>> escape_text("a'b")
"a&#x27;b"
>>> escape_text('a"b')
"a&quot;b"
```

douglas.tools.**filestat**(*config*, *filename*)
Returns the filestat on a given file.

This returns the mtime of the file (same as returned by `time.localtime()`) – tuple of 9 ints.

**Parameters**

- **config** – config
- **filename** – the file name of the file to stat

**Returns** the filestat (tuple of 9 ints) on the given file

douglas.tools.**get_entries**(*cfg*, *root*, *recurse=0*)
Returns a list of all the entries in the root

This only pulls extensions that are registered with entryparsers. This uses the `entries` callback.

Allows plugins to remove and add items.

FIXME - fix docs

douglas.tools.**get_static_files**(*cfg*)
Return a list of `(root, file_path)` tuples for all static files found.

**Parameters** **cfg** – configuration

**Returns** list of `(root, file_path)` tuples

douglas.tools.**importname**(*modulename*, *name*)
Safely imports modules for runtime importing.

**Parameters**

- **modulename** – the package name of the module to import from
- **name** – the name of the module to import

**Returns** the module object or `None` if there were problems importing.

douglas.tools.**pwrap**(*s*)
Wraps the text and prints it.

douglas.tools.**pwrap_error**(*s*)
Wraps an error message and prints it to stderr.

douglas.tools.**render_url**(*cfg*, *pathinfo*, *querystring=''*)
Takes a url and a querystring and renders the page that corresponds with that by creating a Request and a Douglas object and passing it through. It then returns the resulting Response.

**Parameters**

- **cfg** – the config.py dict
- **pathinfo** – the `PATH_INFO` string; example: `/dev/douglas/firstpost.html`
- **querystring** – the querystring (if any); example: debug=yes

**Returns** a douglas `Response` object.

douglas.tools.**render_url_statically**(*cfg*, *url*, *querystring*)
Renders a url and saves the rendered output to the filesystem.

**Parameters**

- **cfg** – config dict
- **url** – url to render
- **querystring** – querystring of the url to render or “”

douglas.tools.**run_callback**(*chain*, *input*, *mappingfunc=<function <lambda>>*, *donefunc=<function <lambda>>*, *defaultfunc=None*, *cache_key=None*)

Executes a callback chain on a given piece of data. passed in is a dict of name/value pairs. Consult the documentation for the specific callback chain you're executing.

Callback chains should conform to their documented behavior. This function allows us to do transforms on data, handling data, and also callbacks.

The difference in behavior is affected by the mappingfunc passed in which converts the output of a given function in the chain to the input for the next function.

If this is confusing, read through the code for this function.

Returns the transformed input dict.

**Parameters**

- **chain** – the name of the callback chain to run
- **input** – dict with name/value pairs that gets passed as the args dict to all callback functions
- **mappingfunc** – the function that maps output arguments to input arguments for the next iteration. It must take two arguments: the original dict and the return from the previous function. It defaults to returning the original dict.
- **donefunc** – this function tests whether we're done doing what we're doing. This function takes as input the output of the most recent iteration. If this function returns True then we'll drop out of the loop. For example, if you wanted a callback to stop running when one of the registered functions returned a 1, then you would pass in: `donefunc=lambda x: x.`
- **defaultfunc** – if this is set and we finish going through all the functions in the chain and none of them have returned something that satisfies the donefunc, then we'll execute the defaultfunc with the latest version of the input dict.
- **cache_key** – If the return value for this callback with these arguments can be cached, then this is a function that takes the original input args dict and returns the cache key.

**Returns** varies

douglas.tools.**urlencode_text**(*s*)

Calls `urllib.quote` on the string `s`.

Note: if `s` is `None`, then we return `None`.

```
>>> urlencode_text(None)
None
>>> urlencode_text("")
""
>>> urlencode_text("a c")
"a%20c"
>>> urlencode_text("a&c")
"a%26c"
>>> urlencode_text("a=c")
"a%3Dc"
```

douglas.tools.**walk**(*request*, *root='.'*, *recurse=0*, *pattern=''*, *return_folders=0*)

This function walks a directory tree starting at a specified root folder, and returns a list of all of the files (and optionally folders) that match our pattern(s). Taken from the online Python Cookbook and modified to own needs.

It will look at the config "ignore_directories" for a list of directories to ignore. It uses a regexp that joins all the things you list. So the following:

```
config.py["ignore_directories"] = ["CVS", "dev/douglas"]
```

turns into the regexp:

```
.*?(CVS|dev/douglas)$
```

It will also skip all directories that start with a period.

**Parameters**

- **request** – Request
- **root** – the root directory to walk
- **recurse** – the depth of recursion; defaults to 0 which goes all the way down
- **pattern** – the regexp object for matching files; defaults to '' which causes douglas to return files with file extensions that match those the entryparsers handle
- **return_folders** – True if you want only folders, False if you want files AND folders

**Returns** a list of file paths.

douglas.tools.**what_ext**(*extensions*, *filepath*)

Takes in a filepath and a list of extensions and tries them all until it finds the first extension that works.

**Parameters**

- **extensions** – the list of extensions to test
- **filepath** – the complete file path (minus the extension) to test and find the extension for

**Returns** the extension (string) of the file or `None`.

## douglas.renderers.base

The is the base renderer module. If you were to dislike the blosxom renderer and wanted to build a renderer that used a different templating system, you would extend the RendererBase class and implement the functionality required by the other rendering system.

For examples, look at the BlosxomRenderer and the Renderer in the debug module.

**class** douglas.renderers.base.**Renderer**(*request*, *stdoutput=<open file '<stdout>', mode 'w'>*)

This is a null renderer.

**class** douglas.renderers.base.**RendererBase**(*request*, *stdoutput=<open file '<stdout>', mode 'w'>*)

Douglas core handles the Input and Process of the system and passes the result of the process to the Renderers for output. All renderers are child classes of RendererBase. RenderBase will contain the public interfaces for all Renderer onject.

**add_header**(**args*)

Populates the HTTP header with lines of text

**Parameters** **args** – Paired list of headers

**Raises** **ValueError** – This happens when the parameters are not correct

**get_content**()

Return the content field

This is exposed for blosxom callbacks.

**Returns** content

**render**(*render_headers=True*)
Do final rendering.

**Parameters** **render_headers** – whether (True) or not (False) to render the headers

**set_content**(*content*)
Sets the content. The content can be any of the following:

- dict
- list of entries

**Parameters** **content** – the content to be displayed

**show_headers**()
Updated the headers of the `Response<douglas.douglas.Response>` instance.

This is here for backwards compatibility.

**write**(*data*)
Convenience method for programs to use instead of accessing self._out.write()

Other classes can override this if there is a unique way to write out data, for example, a two stream output, e.g. one output stream and one output log stream.

Another use for this could be a plugin that writes out binary files, but because renderers and other frameworks may probably not want you to write to `stdout` directly, this method assists you nicely. For example:

```
def cb_start(args):
    req = args['request']
    renderer = req['renderer']

    if reqIsGif and gifFileExists(theGifFile):
        # Read the file
        data = open(theGifFile).read()

        # Modify header
        renderer.add_header('Content-type', 'image/gif')
        renderer.add_header('Content-Length', len(data))
        renderer.show_headers()

        # Write to output
        renderer.write(data)

        # Tell douglas not to render anymore as data is
        # processed already
        renderer.rendered = 1
```

This simple piece of pseudocode explains what you could do with this method, though I highly don't recommend this, unless douglas is running continuously.

**Parameters** **data** – Piece of string you want printed

## douglas.entries.base

This module contains the base class for all the Entry classes. The EntryBase class is essentially the API for entries in douglas. Reading through the comments for this class will walk you through building your own EntryBase derivatives.

This module also holds a generic generate_entry function which will generate a BaseEntry with data that you provide for it.

**class** douglas.entries.base.**EntryBase**(*request*)

EntryBase is the base class for all the Entry classes. Each instance of an Entry class represents a single entry in the weblog, whether it came from a file, or a database, or even somewhere off the InterWeeb.

EntryBase derivatives are dict-like.

**get_id**()

This should return an id that's unique enough for caching purposes.

Override this.

**Returns** string id

**set_time**(*timetuple*)

This takes in a given time tuple and sets all the magic metadata variables we have according to the items in the time tuple.

**Parameters** **timetuple** – the timetuple to use to set the data with–this is the same thing as the mtime/atime portions of an os.stat. This time is expected to be local time, not UTC.

douglas.entries.base.**generate_entry**(*request*, *properties*, *data*, *mtime=None*)

Takes a properties dict and a data string and generates a generic entry using the data you provided.

**Parameters**

- **request** – the Request object
- **properties** – the dict of properties for the entry
- **data** – the data content for the entry
- **mtime** – the mtime tuple (as given by time.localtime()). if you pass in None, then we'll use localtime.

## Release process

1. Update the release process
2. Update WHATSNEW
3. Update the version number and date in:
   - Douglas/_version.py
   - docs/conf.py
4. Update AUTHORS:

```
git log --pretty=format:%an | sort -u
```

5. Commit the changes
6. Tag the commit:

```
git tag -a vx.y
```

   e.g. v1.5

7. Run:

```
./maketarball.sh TAG
```

8. Push everything to `douglas`.

   git push –tags origin master

9. Build the docs:

```
cd docs/
make html
```

10. Write release blog entry
11. Push docs, tarball and blog entry to website.
12. Push website changes to douglas-web repository.
13. Update PYPI using the release tarball:

```
tar -xzvf douglas-x.y.tar.gz
cd douglas-x.y
python setup.py sdist upload
```

14. Send email to douglas-users and douglas-devel.
15. Update version numbers to next version + `.dev`.

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## d

# Index

## Symbols



## A



## B



## C



## D



## E



## F



## G