Pelican Documentation

Release 3.3.0

Alexis Métaireau

December 25, 2013

Contents

Pelican is a static site generator, written in Python.

- Write your content directly with your editor of choice (vim!) in reStructuredText, Markdown, or AsciiDoc formats
- Includes a simple CLI tool to (re)generate your site
- Easy to interface with distributed version control systems and web hooks
- Completely static output is easy to host anywhere

CHAPTER 1

Features

Pelican 3 currently supports:

- Articles (e.g., blog posts) and pages (e.g., “About”, “Projects”, “Contact”)
- Comments, via an external service (Disqus). (Please note that while useful, Disqus is an external service, and thus the comment data will be somewhat outside of your control and potentially subject to data loss.)
- Theming support (themes are created using Jinja2 templates)
- Publication of articles in multiple languages
- Atom/RSS feeds
- Code syntax highlighting
- Import from WordPress, Dotclear, or RSS feeds
- Integration with external tools: Twitter, Google Analytics, etc. (optional)

CHAPTER 2

Why the name “Pelican”?

“Pelican” is an anagram for *calepin*, which means “notebook” in French. ;)

CHAPTER 3

Source code

You can access the source code at: https://github.com/getpelican/pelican

Feedback / Contact us

If you want to see new features in Pelican, don't hesitate to offer suggestions, clone the repository, etc. There are many ways to *contribute*. That's open source, dude!

Send a message to "authors at getpelican dot com" with any requests/feedback. For a more immediate response, you can also join the team via IRC at #pelican on Freenode — if you don't have an IRC client handy, use the webchat for quick feedback. If you ask a question via IRC and don't get an immediate response, don't leave the channel! It may take a few hours because of time zone differences, but f you are patient and remain in the channel, someone will almost always respond to your inquiry.

CHAPTER 5

Documentation

A French version of the documentation is available at `fr/index`.

5.1 Getting started

5.1.1 Installing Pelican

Pelican currently runs best on Python 2.7.x; earlier versions of Python are not supported. There is provisional support for Python 3.3, although there may be rough edges, particularly with regards to optional 3rd-party components.

You can install Pelican via several different methods. The simplest is via pip:

```
$ pip install pelican
```

If you don't have `pip` installed, an alternative method is `easy_install`:

```
$ easy_install pelican
```

(Keep in mind that operating systems will often require you to prefix the above commands with `sudo` in order to install Pelican system-wide.)

While the above is the simplest method, the recommended approach is to create a virtual environment for Pelican via virtualenv before installing Pelican. Assuming you have virtualenv installed, you can then open a new terminal session and create a new virtual environment for Pelican:

```
$ virtualenv ~/virtualenvs/pelican
$ cd ~/virtualenvs/pelican
$ . bin/activate
```

Once the virtual environment has been created and activated, Pelican can be be installed via `pip install pelican` as noted above. Alternatively, if you have the project source, you can install Pelican using the distutils method:

```
$ cd path-to-Pelican-source
$ python setup.py install
```

If you have Git installed and prefer to install the latest bleeding-edge version of Pelican rather than a stable release, use the following command:

```
$ pip install -e git+https://github.com/getpelican/pelican.git#egg=pelican
```

If you plan on using Markdown as a markup format, you'll need to install the Markdown library as well:

```
$ pip install Markdown
```

If you want to use AsciiDoc you need to install it from source or use your operating system's package manager.

Basic usage

Once Pelican is installed, you can use it to convert your Markdown or reST content into HTML via the `pelican` command, specifying the path to your content and (optionally) the path to your settings file:

```
$ pelican /path/to/your/content/ [-s path/to/your/settings.py]
```

The above command will generate your site and save it in the `output/` folder, using the default theme to produce a simple site. The default theme consists of very simple HTML without styling and is provided so folks may use it as a basis for creating their own themes.

You can also tell Pelican to watch for your modifications, instead of manually re-running it every time you want to see your changes. To enable this, run the `pelican` command with the `-r` or `--autoreload` option.

Pelican has other command-line switches available. Have a look at the help to see all the options you can use:

```
$ pelican --help
```

Continue reading below for more detail, and check out the Pelican wiki's Tutorials page for links to community-published tutorials.

Viewing the generated files

The files generated by Pelican are static files, so you don't actually need anything special to view them. You can use your browser to open the generated HTML files directly:

```
firefox output/index.html
```

Because the above method may have trouble locating your CSS and other linked assets, running a simple web server using Python will often provide a more reliable previewing experience:

```
cd output && python -m SimpleHTTPServer
```

Once the `SimpleHTTPServer` has been started, you can preview your site at http://localhost:8000/

Upgrading

If you installed a stable Pelican release via `pip` or `easy_install` and wish to upgrade to the latest stable release, you can do so by adding `--upgrade` to the relevant command. For pip, that would be:

```
$ pip install --upgrade pelican
```

If you installed Pelican via distutils or the bleeding-edge method, simply perform the same step to install the most recent version.

Dependencies

When Pelican is installed, the following dependent Python packages should be automatically installed without any action on your part:

- feedgenerator, to generate the Atom feeds
- jinja2, for templating support
- pygments, for syntax highlighting
- docutils, for supporting reStructuredText as an input format
- pytz, for timezone definitions
- blinker, an object-to-object and broadcast signaling system
- unidecode, for ASCII transliterations of Unicode text
- six, for Python 2 and 3 compatibility utilities
- MarkupSafe, for a markup safe string implementation

If you want the following optional packages, you will need to install them manually via `pip`:

- markdown, for supporting Markdown as an input format
- typogrify, for typographical enhancements

5.1.2 Kickstart your site

Once Pelican has been installed, you can create a skeleton project via the `pelican-quickstart` command, which begins by asking some questions about your site:

```
$ pelican-quickstart
```

Once you finish answering all the questions, your project will consist of the following hierarchy (except for "pages", which you can optionally add yourself if you plan to create non-chronological content):

```
yourproject/
-- content
|   -- (pages)
-- output
-- develop_server.sh
-- fabfile.py
-- Makefile
-- pelicanconf.py       # Main settings file
-- publishconf.py       # Settings to use when ready to publish
```

The next step is to begin to adding content to the *content* folder that has been created for you. (See the **Writing content using Pelican** section below for more information about how to format your content.)

Once you have written some content to generate, you can use the `pelican` command to generate your site, which will be placed in the output folder.

5.1.3 Automation tools

While the `pelican` command is the canonical way to generate your site, automation tools can be used to streamline the generation and publication flow. One of the questions asked during the `pelican-quickstart` process described above pertains to whether you want to automate site generation and publication. If you answered "yes" to that question, a `fabfile.py` and `Makefile` will be generated in the root of your project. These files, pre-populated with certain information gleaned from other answers provided during the `pelican-quickstart` process, are meant as a starting point and should be customized to fit your particular needs and usage patterns. If you find one or both of these automation tools to be of limited utility, these files can deleted at any time and will not affect usage of the canonical `pelican` command.

Following are automation tools that "wrap" the `pelican` command and can simplify the process of generating, previewing, and uploading your site.

Fabric

The advantage of Fabric is that it is written in Python and thus can be used in a wide range of environments. The downside is that it must be installed separately. Use the following command to install Fabric, prefixing with `sudo` if your environment requires it:

```
$ pip install Fabric
```

Take a moment to open the `fabfile.py` file that was generated in your project root. You will see a number of commands, any one of which can be renamed, removed, and/or customized to your liking. Using the out-of-the-box configuration, you can generate your site via:

```
$ fab build
```

If you'd prefer to have Pelican automatically regenerate your site every time a change is detected (which is handy when testing locally), use the following command instead:

```
$ fab regenerate
```

To serve the generated site so it can be previewed in your browser at http://localhost:8000/:

```
$ fab serve
```

If during the `pelican-quickstart` process you answered "yes" when asked whether you want to upload your site via SSH, you can use the following command to publish your site via rsync over SSH:

```
$ fab publish
```

These are just a few of the commands available by default, so feel free to explore `fabfile.py` and see what other commands are available. More importantly, don't hesitate to customize `fabfile.py` to suit your specific needs and preferences.

Make

A `Makefile` is also automatically created for you when you say "yes" to the relevant question during the `pelican-quickstart` process. The advantage of this method is that the `make` command is built into most POSIX systems and thus doesn't require installing anything else in order to use it. The downside is that non-POSIX systems (e.g., Windows) do not include `make`, and installing it on those systems can be a non-trivial task.

If you want to use `make` to generate your site, run:

```
$ make html
```

If you'd prefer to have Pelican automatically regenerate your site every time a change is detected (which is handy when testing locally), use the following command instead:

```
$ make regenerate
```

To serve the generated site so it can be previewed in your browser at http://localhost:8000/:

```
$ make serve
```

Normally you would need to run `make regenerate` and `make serve` in two separate terminal sessions, but you can run both at once via:

```
$ make devserver
```

The above command will simultaneously run Pelican in regeneration mode as well as serve the output at http://localhost:8000. Once you are done testing your changes, you should stop the development server via:

```
$ ./develop_server.sh stop
```

When you're ready to publish your site, you can upload it via the method(s) you chose during the `pelican-quickstart` questionnaire. For this example, we'll use rsync over ssh:

```
$ make rsync_upload
```

That's it! Your site should now be live.

5.1.4 Writing content using Pelican

Articles and pages

Pelican considers "articles" to be chronological content, such as posts on a blog, and thus associated with a date.

The idea behind "pages" is that they are usually not temporal in nature and are used for content that does not change very often (e.g., "About" or "Contact" pages).

File metadata

Pelican tries to be smart enough to get the information it needs from the file system (for instance, about the category of your articles), but some information you need to provide in the form of metadata inside your files.

If you are writing your content in reStructuredText format, you can provide this metadata in text files via the following syntax (give your file the `.rst` extension):

```
My super title
##############

:date: 2010-10-03 10:20
:tags: thats, awesome
:category: yeah
:slug: my-super-post
:author: Alexis Metaireau
:summary: Short version for index and feeds
```

Pelican implements an extension to reStructuredText to enable support for the `abbr` HTML tag. To use it, write something like this in your post:

```
This will be turned into :abbr:`HTML (HyperText Markup Language)`.
```

You can also use Markdown syntax (with a file ending in `.md`, `.markdown`, `.mkd`, or `.mdown`). Markdown generation requires that you first explicitly install the `Markdown` package, which can be done via `pip install Markdown`. Metadata syntax for Markdown posts should follow this pattern:

```
Title: My super title
Date: 2010-12-03 10:20
Category: Python
Tags: pelican, publishing
Slug: my-super-post
Author: Alexis Metaireau
Summary: Short version for index and feeds

This is the content of my super blog post.
```

Conventions for AsciiDoc posts, which should have an `.asc` extension, can be found on the AsciiDoc site.

Pelican can also process HTML files ending in `.html` and `.htm`. Pelican interprets the HTML in a very straightforward manner, reading metadata from `meta` tags, the title from the `title` tag, and the body out from the `body` tag:

```
<html>
    <head>
        <title>My super title</title>
        <meta name="tags" content="thats, awesome" />
        <meta name="date" content="2012-07-09 22:28" />
        <meta name="category" content="yeah" />
        <meta name="author" content="Alexis Métaireau" />
        <meta name="summary" content="Short version for index and feeds" />
    </head>
    <body>
        This is the content of my super blog post.
    </body>
</html>
```

With HTML, there is one simple exception to the standard metadata: `tags` can be specified either via the `tags` metadata, as is standard in Pelican, or via the `keywords` metadata, as is standard in HTML. The two can be used interchangeably.

Note that, aside from the title, none of this article metadata is mandatory: if the date is not specified and `DEFAULT_DATE` is set to `fs`, Pelican will rely on the file's "mtime" timestamp, and the category can be determined by the directory in which the file resides. For example, a file located at `python/foobar/myfoobar.rst` will have a category of `foobar`. If you would like to organize your files in other ways where the name of the subfolder would not be a good category name, you can set the setting `USE_FOLDER_AS_CATEGORY` to `False`. When parsing dates given in the page metadata, Pelican supports the W3C's suggested subset ISO 8601.

If you do not explicitly specify summary metadata for a given post, the `SUMMARY_MAX_LENGTH` setting can be used to specify how many words from the beginning of an article are used as the summary.

You can also extract any metadata from the filename through a regular expression to be set in the `FILENAME_METADATA` setting. All named groups that are matched will be set in the metadata object. The default value for the `FILENAME_METADATA` setting will only extract the date from the filename. For example, if you would like to extract both the date and the slug, you could set something like: `'(?P<date>\d{4}-\d{2}-\d{2})_(?P<slug>.*)'`

Please note that the metadata available inside your files takes precedence over the metadata extracted from the filename.

Pages

If you create a folder named `pages` inside the content folder, all the files in it will be used to generate static pages, such as **About** or **Contact** pages. (See example filesystem layout below.)

You can use the `DISPLAY_PAGES_ON_MENU` setting to control whether all those pages are displayed in the primary navigation menu. (Default is `True`.)

If you want to exclude any pages from being linked to or listed in the menu then add a `status: hidden` attribute to its metadata. This is useful for things like making error pages that fit the generated theme of your site.

Linking to internal content

From Pelican 3.1 onwards, it is now possible to specify intra-site links to files in the *source content* hierarchy instead of files in the *generated* hierarchy. This makes it easier to link from the current post to other posts and images that may be sitting alongside the current post (instead of having to determine where those resources will be placed after site generation).

To link to internal content (files in the `content` directory), use the following syntax: `{filename}path/to/file`:

```
website/
-- content
|   -- article1.rst
|   -- cat/
|   |   -- article2.md
|   -- pages
|       -- about.md
-- pelican.conf.py
```

In this example, `article1.rst` could look like:

```
The first article
#################

:date: 2012-12-01 10:02

See below intra-site link examples in reStructuredText format.

`a link relative to content root <{filename}/cat/article2.rst>`_
`a link relative to current file <{filename}cat/article2.rst>`_
```

and `article2.md`:

```
Title: The second article
Date: 2012-12-01 10:02

See below intra-site link examples in Markdown format.

[a link relative to content root]({filename}/article1.md)
[a link relative to current file]({filename}../article1.md)
```

Embedding non-article or non-page content is slightly different in that the directories need to be specified in `pelicanconf.py` file. The `images` directory is configured for this by default but others will need to be added manually:

```
content
-- images
|   -- han.jpg
-- misc
    -- image-test.md
```

And `image-test.md` would include:

```
![Alt Text]({filename}/images/han.jpg)
```

Any content can be linked in this way. What happens is that the `images` directory gets copied to `output/` during site generation because Pelican includes `images` in the `STATIC_PATHS` setting's list by default. If you want to have another directory, say `pdfs`, copied from your content to your output during site generation, you would need to add the following to your settings file:

```
STATIC_PATHS = ['images', 'pdfs']
```

After the above line has been added, subsequent site generation should copy the `content/pdfs/` directory to `output/pdfs/`.

You can also link to categories or tags, using the `{tag}tagname` and `{category}foobar` syntax.

For backward compatibility, Pelican also supports bars (`||`) in addition to curly braces (`{}`). For example: `|filename|an_article.rst`, `|tag|tagname`, `|category|foobar`. The syntax was changed from `||` to `{}` to avoid collision with Markdown extensions or reST directives.

Importing an existing blog

It is possible to import your blog from Dotclear, WordPress, and RSS feeds using a simple script. See *Import from other blog software*.

Translations

It is possible to translate articles. To do so, you need to add a `lang` meta attribute to your articles/pages and set a `DEFAULT_LANG` setting (which is English [en] by default). With those settings in place, only articles with the default language will be listed, and each article will be accompanied by a list of available translations for that article.

Pelican uses the article's URL "slug" to determine if two or more articles are translations of one another. The slug can be set manually in the file's metadata; if not set explicitly, Pelican will auto-generate the slug from the title of the article.

Here is an example of two articles, one in English and the other in French.

The English article:

```
Foobar is not dead
##################

:slug: foobar-is-not-dead
:lang: en

That's true, foobar is still alive!
```

And the French version:

```
Foobar n'est pas mort !
#######################

:slug: foobar-is-not-dead
:lang: fr

Oui oui, foobar est toujours vivant !
```

Post content quality notwithstanding, you can see that only item in common between the two articles is the slug, which is functioning here as an identifier. If you'd rather not explicitly define the slug this way, you must then instead ensure that the translated article titles are identical, since the slug will be auto-generated from the article title.

If you do not want the original version of one specific article to be detected by the `DEFAULT_LANG` setting, use the `translation` metadata to specify which posts are translations:

```
Foobar is not dead
##################

:slug: foobar-is-not-dead
:lang: en
:translation: true

That's true, foobar is still alive!
```

Syntax highlighting

Pelican is able to provide colorized syntax highlighting for your code blocks. To do so, you have to use the following conventions inside your content files.

For reStructuredText, use the code-block directive:

```
.. code-block:: identifier

   <indented code block goes here>
```

For Markdown, include the language identifier just above the code block, indenting both the identifier and code:

```
A block of text.

    :::identifier
    <code goes here>
```

The specified identifier (e.g. `python`, `ruby`) should be one that appears on the list of available lexers.

When using reStructuredText the following options are available in the code-block directive:

Option	Valid values	Description
anchor-linenos	N/A	If present wrap line numbers in <a> tags.
classprefix	string	String to prepend to token class names
hl_lines	numbers	List of lines to be highlighted.
lineanchors	string	Wrap each line in an anchor using this string and -linenumber.
linenos	string	If present or set to “table” output line numbers in a table, if set to “inline” output them inline. “none” means do not output the line numbers for this table.
linenospecial	number	If set every nth line will be given the ‘special’ css class.
linenostart	number	Line number for the first line.
linenostep	number	Print every nth line number.
lineseparator	string	String to print between lines of code, ‘n’ by default.
linespans	string	Wrap each line in a span using this and -linenumber.
nobackground	N/A	If set do not output background color for the wrapping element
nowrap	N/A	If set do not wrap the tokens at all.
tagsfile	string	ctags file to use for name definitions.
tagurlformat	string	format for the ctag links.

Note that, depending on the version, your Pygments module might not have all of these options available. Refer to the *HtmlFormatter* section of the Pygments documentation for more details on each of the options.

For example, the following code block enables line numbers, starting at 153, and prefixes the Pygments CSS classes with *pgcss* to make the names more unique and avoid possible CSS conflicts:

```
.. code-block:: identifier
    :classprefix: pgcss
    :linenos: table
    :linenostart: 153

   <indented code block goes here>
```

It is also possible to specify the `PYGMENTS_RST_OPTIONS` variable in your Pelican settings file to include options that will be automatically applied to every code block.

For example, if you want to have line numbers displayed for every code block and a CSS prefix you would set this variable to:

```
PYGMENTS_RST_OPTIONS = {'classprefix': 'pgcss', 'linenos': 'table'}
```

If specified, settings for individual code blocks will override the defaults in your settings file.

Publishing drafts

If you want to publish an article as a draft (for friends to review before publishing, for example), you can add a `Status:  draft` attribute to its metadata. That article will then be output to the `drafts` folder and not listed on the index page nor on any category or tag page.

5.2 Settings

Pelican is configurable thanks to a configuration file you can pass to the command line:

```
$ pelican -s path/to/your/settingsfile.py path
```

Settings are configured in the form of a Python module (a file). You can see an example by looking at /samples/pelican.conf.py

All the setting identifiers must be set in all-caps, otherwise they will not be processed. Setting values that are numbers (5, 20, etc.), booleans (True, False, None, etc.), dictionaries, or tuples should *not* be enclosed in quotation marks. All other values (i.e., strings) *must* be enclosed in quotation marks.

Unless otherwise specified, settings that refer to paths can be either absolute or relative to the configuration file.

The settings you define in the configuration file will be passed to the templates, which allows you to use your settings to add site-wide content.

Here is a list of settings for Pelican:

5.2.1 Basic settings

Setting name (default value)	**What does it do?**
AUTHOR	Default author (put your name)
DATE_FORMATS (`{}`)	If you manage multiple languages, you c
USE_FOLDER_AS_CATEGORY (`True`)	When you don’t specify a category in yo
DEFAULT_CATEGORY (`'misc'`)	The default category to fall back on.
DEFAULT_DATE_FORMAT (`'%a %d %B %Y'`)	The default date format you want to use.
DISPLAY_PAGES_ON_MENU (`True`)	Whether to display pages on the menu of
DISPLAY_CATEGORIES_ON_MENU (`True`)	Whether to display categories on the men
DEFAULT_DATE (`None`)	The default date you want to use. If `fs`,
DEFAULT_METADATA (`()`)	The default metadata you want to use for
FILENAME_METADATA (`'(?P<date>\d{4}-\d{2}-\d{2}).*'`)	The regexp that will be used to extract an
PATH_METADATA (”)	Like `FILENAME_METADATA`, but parse
EXTRA_PATH_METADATA (`{}`)	Extra metadata dictionaries keyed by rela
DELETE_OUTPUT_DIRECTORY (`False`)	Delete the output directory, and **all** of its
OUTPUT_RETENTION (`()`)	A tuple of filenames that should be retain
JINJA_EXTENSIONS (`[]`)	A list of any Jinja2 extensions you want
JINJA_FILTERS (`{}`)	A list of custom Jinja2 filters you want to
LOCALE (“[1])	Change the locale. A list of locales can b
READERS (`{}`)	A dictionary of file extensions / Reader c
IGNORE_FILES (`['.#*']`)	A list of file globbing patterns to match a
MD_EXTENSIONS (`['codehilite(css_class=highlight)','extra']`)	A list of the extensions that the Markdow
OUTPUT_PATH (`'output/'`)	Where to output the generated files.
PATH (`None`)	Path to content directory to be processed
PAGE_DIR (`'pages'`)	Directory to look at for pages, relative to
PAGE_EXCLUDES (`()`)	A list of directories to exclude when look
ARTICLE_DIR (”)	Directory to look at for articles, relative t

[1] Default is the system locale.

Setting name (default value)	What does it do?
ARTICLE_EXCLUDES: `(('pages',))`	A list of directories to exclude when lool
OUTPUT_SOURCES (`False`)	Set to True if you want to copy the articl
OUTPUT_SOURCES_EXTENSION (`.text`)	Controls the extension that will be used l
RELATIVE_URLS (`False`)	Defines whether Pelican should use docu
PLUGINS (`[]`)	The list of plugins to load. See *Plugins*.
SITENAME (`'A Pelican Blog'`)	Your site name
SITEURL	Base URL of your website. Not defined
TEMPLATE_PAGES (`None`)	A mapping containing template pages th
STATIC_PATHS (`['images']`)	The static paths you want to have access
TIMEZONE	The timezone used in the date informatic
TYPOGRIFY (`False`)	If set to True, several typographical impr
DIRECT_TEMPLATES (`('index', 'tags', 'categories', 'archives')`)	List of templates that are used directly to
PAGINATED_DIRECT_TEMPLATES (`('index',)`)	Provides the direct templates that should
SUMMARY_MAX_LENGTH (`50`)	When creating a short summary of an art
EXTRA_TEMPLATES_PATHS (`[]`)	A list of paths you want Jinja2 to search
ASCIIDOC_OPTIONS (`[]`)	A list of options to pass to AsciiDoc. See
WITH_FUTURE_DATES (`True`)	If disabled, content with dates in the futu
INTRASITE_LINK_REGEX (`'[{|](?P<what>.*?)[|}]'`)	Regular expression that is used to parse i
PYGMENTS_RST_OPTIONS (`[]`)	A list of default Pygments settings for yo

URL settings

The first thing to understand is that there are currently two supported methods for URL formation: *relative* and *absolute*. Relative URLs are useful when testing locally, and absolute URLs are reliable and most useful when publishing. One method of supporting both is to have one Pelican configuration file for local development and another for publishing. To see an example of this type of setup, use the `pelican-quickstart` script as described at the top of the *Getting Started* page, which will produce two separate configuration files for local development and publishing, respectively.

You can customize the URLs and locations where files will be saved. The `*_URL` and `*_SAVE_AS` variables use Python's format strings. These variables allow you to place your articles in a location such as `{slug}/index.html` and link to them as `{slug}` for clean URLs. These settings give you the flexibility to place your articles and pages anywhere you want.

Note: If you specify a `datetime` directive, it will be substituted using the input files' date metadata attribute. If the date is not specified for a particular file, Pelican will rely on the file's `mtime` timestamp.

Check the Python datetime documentation at http://bit.ly/cNcJUC for more information.

Also, you can use other file metadata attributes as well:

- slug
- date
- lang
- author
- category

Example usage:

- ARTICLE_URL = `'posts/{date:%Y}/{date:%b}/{date:%d}/{slug}/'`

- ARTICLE_SAVE_AS = `'posts/{date:%Y}/{date:%b}/{date:%d}/{slug}/index.html'`

This would save your articles in something like `/posts/2011/Aug/07/sample-post/index.html`, and the URL to this would be `/posts/2011/Aug/07/sample-post/`.

Pelican can optionally create per-year, per-month, and per-day archives of your posts. These secondary archives are disabled by default but are automatically enabled if you supply format strings for their respective `_SAVE_AS` settings. Period archives fit intuitively with the hierarchical model of web URLs and can make it easier for readers to navigate through the posts you've written over time.

Example usage:

- YEAR_ARCHIVE_SAVE_AS = `'posts/{date:%Y}/index.html'`
- MONTH_ARCHIVE_SAVE_AS = `'posts/{date:%Y}/{date:%b}/index.html'`

With these settings, Pelican will create an archive of all your posts for the year at (for instance) `posts/2011/index.html` and an archive of all your posts for the month at `posts/2011/Aug/index.html`.

Note: Period archives work best when the final path segment is `index.html`. This way a reader can remove a portion of your URL and automatically arrive at an appropriate archive of posts, without having to specify a page name.

Setting name (default value)	What does it do?
ARTICLE_URL (`'{slug}.html'`)	The URL to refer to an ARTICLE.
ARTICLE_SAVE_AS (`'{slug}.html'`)	The place where we will save an article.
ARTICLE_LANG_URL (`'{slug}-{lang}.html'`)	The URL to refer to an ARTICLE which doesn't use the default language.
ARTICLE_LANG_SAVE_AS (`'{slug}-{lang}.html'`)	The place where we will save an article which doesn't use the default language.
PAGE_URL (`'pages/{slug}.html'`)	The URL we will use to link to a page.
PAGE_SAVE_AS (`'pages/{slug}.html'`)	The location we will save the page. This value has to be the same as PAGE_URL or you need to use a rewrite in your server config.
PAGE_LANG_URL (`'pages/{slug}-{lang}.html'`)	The URL we will use to link to a page which doesn't use the default language.
PAGE_LANG_SAVE_AS (`'pages/{slug}-{lang}.html'`)	The location we will save the page which doesn't use the default language.
CATEGORY_URL (`'category/{slug}.html'`)	The URL to use for a category.
CATEGORY_SAVE_AS (`'category/{slug}.html'`)	The location to save a category.
TAG_URL (`'tag/{slug}.html'`)	The URL to use for a tag.
TAG_SAVE_AS (`'tag/{slug}.html'`)	The location to save the tag page.
TAGS_URL (`'tags.html'`)	The URL to use for the tag list.
TAGS_SAVE_AS (`'tags.html'`)	The location to save the tag list.
AUTHOR_URL (`'author/{slug}.html'`)	The URL to use for an author.
AUTHOR_SAVE_AS (`'author/{slug}.html'`)	The location to save an author.
AUTHORS_URL (`'authors.html'`)	The URL to use for the author list.
AUTHORS_SAVE_AS (`'authors.html'`)	The location to save the author list.
<DIRECT_TEMPLATE_NAME>_SAVE_AS	The location to save content generated from direct templates. Where <DIRECT_TEMPLATE_NAME> is the upper case template name.
ARCHIVES_SAVE_AS (`'archives.html'`)	The location to save the article archives page.
YEAR_ARCHIVE_SAVE_AS (False)	The location to save per-year archives of your posts.
MONTH_ARCHIVE_SAVE_AS (False)	The location to save per-month archives of your posts.
DAY_ARCHIVE_SAVE_AS (False)	The location to save per-day archives of your posts.
SLUG_SUBSTITUTIONS (`()`)	Substitutions to make prior to stripping out non-alphanumerics when generating slugs. Specified as a list of 2-tuples of `(from, to)` which are applied in order.

Note: If you do not want one or more of the default pages to be created (e.g., you are the only author on your site and thus do not need an Authors page), set the corresponding `*_SAVE_AS` setting to `False` to prevent the relevant page from being generated.

Timezone

If no timezone is defined, UTC is assumed. This means that the generated Atom and RSS feeds will contain incorrect date information if your locale is not UTC.

Pelican issues a warning in case this setting is not defined, as it was not mandatory in previous versions.

Have a look at the wikipedia page to get a list of valid timezone values.

Date format and locale

If no DATE_FORMATS are set, Pelican will fall back to DEFAULT_DATE_FORMAT. If you need to maintain multiple languages with different date formats, you can set the DATE_FORMATS dictionary using the language name (lang metadata in your post content) as the key. Regarding available format codes, see strftime document of python :

```
DATE_FORMATS = {
    'en': '%a, %d %b %Y',
    'jp': '%Y-%m-%d(%a)',
}
```

You can set locale to further control date format:

```
LOCALE = ('usa', 'jpn',   # On Windows
    'en_US', 'ja_JP'      # On Unix/Linux
    )
```

Also, it is possible to set different locale settings for each language. If you put (locale, format) tuples in the dict, this will override the LOCALE setting above:

```
# On Unix/Linux
DATE_FORMATS = {
    'en': ('en_US','%a, %d %b %Y'),
    'jp': ('ja_JP','%Y-%m-%d(%a)'),
}

# On Windows
DATE_FORMATS = {
    'en': ('usa','%a, %d %b %Y'),
    'jp': ('jpn','%Y-%m-%d(%a)'),
}
```

This is a list of available locales on Windows . On Unix/Linux, usually you can get a list of available locales via the `locale -a` command; see manpage locale(1) for more information.

5.2.2 Template pages

If you want to generate custom pages besides your blog entries, you can point any Jinja2 template file with a path pointing to the file and the destination path for the generated file.

For instance, if you have a blog with three static pages — a list of books, your resume, and a contact page — you could have:

```
TEMPLATE_PAGES = {'src/books.html': 'dest/books.html',
                  'src/resume.html': 'dest/resume.html',
                  'src/contact.html': 'dest/contact.html'}
```

5.2.3 Path metadata

Not all metadata needs to be embedded in source file itself. For example, blog posts are often named following a `YYYY-MM-DD-SLUG.rst` pattern, or nested into `YYYY/MM/DD-SLUG` directories. To extract metadata from the filename or path, set `FILENAME_METADATA` or `PATH_METADATA` to regular expressions that use Python's group name notation `(?P<name>...)`. If you want to attach additional metadata but don't want to encode it in the path, you can set `EXTRA_PATH_METADATA`:

```
EXTRA_PATH_METADATA = {
    'relative/path/to/file-1': {
        'key-1a': 'value-1a',
        'key-1b': 'value-1b',
        },
    'relative/path/to/file-2': {
        'key-2': 'value-2',
        },
    }
```

This can be a convenient way to shift the installed location of a particular file:

```
# Take advantage of the following defaults
# STATIC_SAVE_AS = '{path}'
# STATIC_URL = '{path}'
STATIC_PATHS = [
    'extra/robots.txt',
    ]
EXTRA_PATH_METADATA = {
    'extra/robots.txt': {'path': 'robots.txt'},
    }
```

5.2.4 Feed settings

By default, Pelican uses Atom feeds. However, it is also possible to use RSS feeds if you prefer.

Pelican generates category feeds as well as feeds for all your articles. It does not generate feeds for tags by default, but it is possible to do so using the `TAG_FEED_ATOM` and `TAG_FEED_RSS` settings:

Setting name (default value)	What does it do?
FEED_DOMAIN (`None`, i.e. base URL is "/")	The domain prepended to feed URLs. Since feed URLs should always be absolute, it is highly recommended to define this (e.g., "http://feeds.example.com"). If you have already explicitly defined SITEURL (see above) and want to use the same domain for your feeds, you can just set: `FEED_DOMAIN = SITEURL`.
FEED_ATOM (`None`, i.e. no Atom feed)	Relative URL to output the Atom feed.
FEED_RSS (`None`, i.e. no RSS)	Relative URL to output the RSS feed.
FEED_ALL_ATOM (`'feeds/all.atom.xml'`)	Relative URL to output the all posts Atom feed: this feed will contain all posts regardless of their language.
FEED_ALL_RSS (`None`, i.e. no all RSS)	Relative URL to output the all posts RSS feed: this feed will contain all posts regardless of their language.
CATEGORY_FEED_ATOM ('feeds/%s.atom.xml'[2])	Where to put the category Atom feeds.
CATEGORY_FEED_RSS (`None`, i.e. no RSS)	Where to put the category RSS feeds.
TAG_FEED_ATOM (`None`, i.e. no tag feed)	Relative URL to output the tag Atom feed. It should be defined using a "%s" match in the tag name.
TAG_FEED_RSS (`None`, ie no RSS tag feed)	Relative URL to output the tag RSS feed
FEED_MAX_ITEMS	Maximum number of items allowed in a feed. Feed item quantity is unrestricted by default.

If you don't want to generate some or any of these feeds, set the above variables to `None`.

FeedBurner

If you want to use FeedBurner for your feed, you will likely need to decide upon a unique identifier. For example, if your site were called "Thyme" and hosted on the www.example.com domain, you might use "thymefeeds" as your unique identifier, which we'll use throughout this section for illustrative purposes. In your Pelican settings, set the *FEED_ATOM* attribute to "thymefeeds/main.xml" to create an Atom feed with an original address of *http://www.example.com/thymefeeds/main.xml*. Set the *FEED_DOMAIN* attribute to *http://feeds.feedburner.com*, or *http://feeds.example.com* if you are using a CNAME on your own domain (i.e., FeedBurner's "MyBrand" feature).

There are two fields to configure in the FeedBurner interface: "Original Feed" and "Feed Address". In this example, the "Original Feed" would be *http://www.example.com/thymefeeds/main.xml* and the "Feed Address" suffix would be *thymefeeds/main.xml*.

5.2.5 Pagination

The default behaviour of Pelican is to list all the article titles along with a short description on the index page. While this works well for small-to-medium sites, sites with a large quantity of articles will probably benefit from paginating this list.

You can use the following settings to configure the pagination.

[2] %s is the name of the category.

Setting name (default value)	What does it do?
DEFAULT_ORPHANS (0)	The minimum number of articles allowed on the last page. Use this when you don't want the last page to only contain a handful of articles.
DEFAULT_PAGINATION (False)	The maximum number of articles to include on a page, not including orphans. False to disable pagination.
PAGINATION_PATTERNS	A set of patterns that are used to determine advanced pagination output.

Using Pagination Patterns

The `PAGINATION_PATTERNS` setting can be used to configure where subsequent pages are created. The setting is a sequence of three element tuples, where each tuple consists of:

```
(minimum page, URL setting, SAVE_AS setting,)
```

For example, if you wanted the first page to just be `/`, and the second (and subsequent) pages to be `/page/2/`, you would set `PAGINATION_PATTERNS` as follows:

```
PAGINATION_PATTERNS = (
    (1, '{base_name}/', '{base_name}/index.html'),
    (2, '{base_name}/page/{number}/', '{base_name}/page/{number}/index.html'),
)
```

This would cause the first page to be written to `{base_name}/index.html`, and subsequent ones would be written into `page/{number}` directories.

5.2.6 Tag cloud

If you want to generate a tag cloud with all your tags, you can do so using the following settings.

Setting name (default value)	What does it do?
TAG_CLOUD_STEPS (4)	Count of different font sizes in the tag cloud.
TAG_CLOUD_MAX_ITEMS (100)	Maximum number of tags in the cloud.

The default theme does not include a tag cloud, but it is pretty easy to add one:

```
<ul class="tagcloud">
    {% for tag in tag_cloud %}
        <li class="tag-{{ tag.1 }}"><a href="{{ SITEURL }}/{{ tag.0.url }}">{{ tag.0 }}</a></li>
    {% endfor %}
</ul>
```

You should then also define CSS styles with appropriate classes (tag-0 to tag-N, where N matches `TAG_CLOUD_STEPS` -1), tag-0 being the most frequent, and define a `ul.tagcloud` class with appropriate list-style to create the cloud. For example:

```
ul.tagcloud {
  list-style: none;
    padding: 0;
}

ul.tagcloud li {
    display: inline-block;
}

li.tag-0 {
    font-size: 150%;
}

li.tag-1 {
    font-size: 120%;
}

...
```

5.2.7 Translations

Pelican offers a way to translate articles. See the *Getting Started* section for more information.

Setting name (default value)	What does it do?
DEFAULT_LANG (`'en'`)	The default language to use.
TRANSLATION_FEED_ATOM ('feeds/all-%s.atom.xml'[3])	Where to put the Atom feed for translations.
TRANSLATION_FEED_RSS (`None`, i.e. no RSS)	Where to put the RSS feed for translations.

5.2.8 Ordering content

Setting name (default value)	What does it do?
NEWEST_FIRST_ARCHIVES (`True`)	Order archives by newest first by date. (False: orders by date with older articles first.)
REVERSE_CATEGORY_ORDER (`False`)	Reverse the category order. (True: lists by reverse alphabetical order; default lists alphabetically.)

5.2.9 Themes

Creating Pelican themes is addressed in a dedicated section (see *How to create themes for Pelican*). However, here are the settings that are related to themes.

Setting name (default value)	What does it do?
THEME	Theme to use to produce the output. Can be a relative or absolute path to a theme folder, or the name of a default theme or a theme installed via `pelican-themes` (see below).
THEME_STATIC_DIR (`'theme'`)	Destination directory in the output path where Pelican will place the files collected from *THEME_STATIC_PATHS*. Default is *theme*.
THEME_STATIC_PATHS (`['static']`)	Static theme paths you want to copy. Default value is *static*, but if your theme has other static paths, you can put them here. If files or directories with the same names are included in the paths defined in this settings, they will be progressively overwritten.
CSS_FILE (`'main.css'`)	Specify the CSS file you want to load.

By default, two themes are available. You can specify them using the *THEME* setting or by passing the `-t` option to the `pelican` command:

- notmyidea

[3] %s is the language

- simple (a synonym for "plain text" :)

There are a number of other themes available at http://github.com/getpelican/pelican-themes. Pelican comes with *pelican-themes*, a small script for managing themes.

You can define your own theme, either by starting from scratch or by duplicating and modifying a pre-existing theme. Here is *a guide on how to create your theme*.

Following are example ways to specify your preferred theme:

```
# Specify name of a built-in theme
THEME = "notmyidea"
# Specify name of a theme installed via the pelican-themes tool
THEME = "chunk"
# Specify a customized theme, via path relative to the settings file
THEME = "themes/mycustomtheme"
# Specify a customized theme, via absolute path
THEME = "~/projects/mysite/themes/mycustomtheme"
```

The built-in `notmyidea` theme can make good use of the following settings. Feel free to use them in your themes as well.

Setting name	What does it do ?
SITESUBTITLE	A subtitle to appear in the header.
DISQUS_SITENAME	Pelican can handle Disqus comments. Specify the Disqus sitename identifier here.
GITHUB_URL	Your GitHub URL (if you have one). It will then use this information to create a GitHub ribbon.
GOOGLE_ANALYTICS	'UA-XXXX-YYYY' to activate Google Analytics.
GOSQUARED_SITENAME	'XXX-YYYYYY-X' to activate GoSquared.
MENUITEMS	A list of tuples (Title, URL) for additional menu items to appear at the beginning of the main menu.
PIWIK_URL	URL to your Piwik server - without 'http://' at the beginning.
PIWIK_SSL_URL	If the SSL-URL differs from the normal Piwik-URL you have to include this setting too. (optional)
PIWIK_SITE_ID	ID for the monitored website. You can find the ID in the Piwik admin interface > settings > websites.
LINKS	A list of tuples (Title, URL) for links to appear on the header.
SOCIAL	A list of tuples (Title, URL) to appear in the "social" section.
TWITTER_USERNAME	Allows for adding a button to articles to encourage others to tweet about them. Add your Twitter username if you want this button to appear.

In addition, you can use the "wide" version of the `notmyidea` theme by adding the following to your configuration:

```
CSS_FILE = "wide.css"
```

5.2.10 Example settings

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

AUTHOR = 'Alexis Métaireau'
SITENAME = "Alexis' log"
SITEURL = 'http://blog.notmyidea.org'
TIMEZONE = "Europe/Paris"

# can be useful in development, but set to False when you're ready to publish
```

```
RELATIVE_URLS = True

GITHUB_URL = 'http://github.com/ametaireau/'
DISQUS_SITENAME = "blog-notmyidea"
PDF_GENERATOR = False
REVERSE_CATEGORY_ORDER = True
LOCALE = "C"
DEFAULT_PAGINATION = 4
DEFAULT_DATE = (2012, 3, 2, 14, 1, 1)

FEED_ALL_RSS = 'feeds/all.rss.xml'
CATEGORY_FEED_RSS = 'feeds/%s.rss.xml'

LINKS = (('Biologeek', 'http://biologeek.org'),
         ('Filyb', "http://filyb.info/"),
         ('Libert-fr', "http://www.libert-fr.com"),
         ('N1k0', "http://prendreuncafe.com/blog/"),
         ('Tarek Ziadé', "http://ziade.org/blog"),
         ('Zubin Mithra', "http://zubin71.wordpress.com/"),)

SOCIAL = (('twitter', 'http://twitter.com/ametaireau'),
          ('lastfm', 'http://lastfm.com/user/akounet'),
          ('github', 'http://github.com/ametaireau'),)

# global metadata to all the contents
DEFAULT_METADATA = (('yeah', 'it is'),)

# path-specific metadata
EXTRA_PATH_METADATA = {
    'extra/robots.txt': {'path': 'robots.txt'},
    }

# static paths will be copied without parsing their contents
STATIC_PATHS = [
    'pictures',
    'extra/robots.txt',
    ]

# custom page generated with a jinja2 template
TEMPLATE_PAGES = {'pages/jinja2_template.html': 'jinja2_template.html'}

# code blocks with line numbers
PYGMENTS_RST_OPTIONS = {'linenos': 'table'}

# foobar will not be used, because it's not in caps. All configuration keys
# have to be in caps
foobar = "barbaz"
```

5.3 How to create themes for Pelican

Pelican uses the great Jinja2 templating engine to generate its HTML output. Jinja2 syntax is really simple. If you want to create your own theme, feel free to take inspiration from the "simple" theme.

5.3.1 Structure

To make your own theme, you must follow the following structure:

```
-- static
|   -- css
|   -- images
-- templates
    -- archives.html         // to display archives
    -- period_archives.html  // to display time-period archives
    -- article.html          // processed for each article
    -- author.html           // processed for each author
    -- authors.html          // must list all the authors
    -- categories.html       // must list all the categories
    -- category.html         // processed for each category
    -- index.html            // the index. List all the articles
    -- page.html             // processed for each page
    -- tag.html              // processed for each tag
    -- tags.html             // must list all the tags. Can be a tag cloud.
```

- *static* contains all the static assets, which will be copied to the output *theme* folder. The above filesystem layout includes CSS and image folders, but those are just examples. Put what you need here.
- *templates* contains all the templates that will be used to generate the content. The template files listed above are mandatory; you can add your own templates if it helps you keep things organized while creating your theme.

5.3.2 Templates and variables

The idea is to use a simple syntax that you can embed into your HTML pages. This document describes which templates should exist in a theme, and which variables will be passed to each template at generation time.

All templates will receive the variables defined in your settings file, as long as they are in all-caps. You can access them directly.

Common variables

All of these settings will be available to all templates.

Variable	Description
output_file	The name of the file currently being generated. For instance, when Pelican is rendering the homepage, output_file will be "index.html".
articles	The list of articles, ordered descending by date. All the elements are *Article* objects, so you can access their attributes (e.g. title, summary, author etc.). Sometimes this is shadowed (for instance in the tags page). You will then find info about it in the *all_articles* variable.
dates	The same list of articles, but ordered by date, ascending.
tags	A list of (tag, articles) tuples, containing all the tags.
categories	A list of (category, articles) tuples, containing all the categories and corresponding articles (values)
pages	The list of pages

Sorting

URL wrappers (currently categories, tags, and authors), have comparison methods that allow them to be easily sorted by name:

```
{% for tag, articles in tags|sort %}
```

If you want to sort based on different criteria, Jinja's sort command has a number of options.

Date Formatting

Pelican formats the date according to your settings and locale (`DATE_FORMATS`/`DEFAULT_DATE_FORMAT`) and provides a `locale_date` attribute. On the other hand, the `date` attribute will be a datetime object. If you need custom formatting for a date different than your settings, use the Jinja filter `strftime` that comes with Pelican. Usage is same as Python strftime format, but the filter will do the right thing and format your date according to the locale given in your settings:

```
{{ article.date|strftime('%d %B %Y') }}
```

index.html

This is the home page of your blog, generated at output/index.html.

If pagination is active, subsequent pages will reside in output/index'n'.html.

Variable	Description
articles_paginator	A paginator object for the list of articles
articles_page	The current page of articles
dates_paginator	A paginator object for the article list, ordered by date, ascending.
dates_page	The current page of articles, ordered by date, ascending.
page_name	'index' – useful for pagination links

author.html

This template will be processed for each of the existing authors, with output generated at output/author/*author_name*.html.

If pagination is active, subsequent pages will reside as defined by setting AUTHOR_SAVE_AS (*Default:* output/author/*author_name'n'*.html).

Variable	Description
author	The name of the author being processed
articles	Articles by this author
dates	Articles by this author, but ordered by date, ascending
articles_paginator	A paginator object for the list of articles
articles_page	The current page of articles
dates_paginator	A paginator object for the article list, ordered by date, ascending.
dates_page	The current page of articles, ordered by date, ascending.
page_name	AUTHOR_URL where everything after *{slug}* is removed – useful for pagination links

category.html

This template will be processed for each of the existing categories, with output generated at output/category/*category_name*.html.

If pagination is active, subsequent pages will reside as defined by setting CATEGORY_SAVE_AS (*Default:* output/category/*category_name'n'*.html).

Variable	Description
category	The name of the category being processed
articles	Articles for this category
dates	Articles for this category, but ordered by date, ascending
articles_paginator	A paginator object for the list of articles
articles_page	The current page of articles
dates_paginator	A paginator object for the list of articles, ordered by date, ascending
dates_page	The current page of articles, ordered by date, ascending
page_name	CATEGORY_URL where everything after *{slug}* is removed – useful for pagination links

article.html

This template will be processed for each article, with .html files saved as output/*article_name*.html. Here are the specific variables it gets.

Variable	Description
article	The article object to be displayed
category	The name of the category for the current article

page.html

This template will be processed for each page, with corresponding .html files saved as output/*page_name*.html.

Variable	Description
page	The page object to be displayed. You can access its title, slug, and content.

tag.html

This template will be processed for each tag, with corresponding .html files saved as output/tag/*tag_name*.html.

If pagination is active, subsequent pages will reside as defined in setting TAG_SAVE_AS (*Default:* output/tag/*tag_name'n'*.html).

Variable	Description
tag	The name of the tag being processed
articles	Articles related to this tag
dates	Articles related to this tag, but ordered by date, ascending
articles_paginator	A paginator object for the list of articles
articles_page	The current page of articles
dates_paginator	A paginator object for the list of articles, ordered by date, ascending
dates_page	The current page of articles, ordered by date, ascending
page_name	TAG_URL where everything after *{slug}* is removed – useful for pagination links

5.3.3 Feeds

The feed variables changed in 3.0. Each variable now explicitly lists ATOM or RSS in the name. ATOM is still the default. Old themes will need to be updated. Here is a complete list of the feed variables:

```
FEED_ATOM
FEED_RSS
FEED_ALL_ATOM
FEED_ALL_RSS
CATEGORY_FEED_ATOM
```

```
CATEGORY_FEED_RSS
TAG_FEED_ATOM
TAG_FEED_RSS
TRANSLATION_FEED_ATOM
TRANSLATION_FEED_RSS
```

5.3.4 Inheritance

Since version 3.0, Pelican supports inheritance from the `simple` theme, so you can re-use the `simple` theme templates in your own themes.

If one of the mandatory files in the `templates/` directory of your theme is missing, it will be replaced by the matching template from the `simple` theme. So if the HTML structure of a template in the `simple` theme is right for you, you don't have to write a new template from scratch.

You can also extend templates from the `simple` themes in your own themes by using the `{% extends %}` directive as in the following example:

Example

With this system, it is possible to create a theme with just two files.

base.html

The first file is the `templates/base.html` template:

1. On the first line, we extend the `base.html` template from the `simple` theme, so we don't have to rewrite the entire file.
2. On the third line, we open the `head` block which has already been defined in the `simple` theme.
3. On the fourth line, the function `super()` keeps the content previously inserted in the `head` block.
4. On the fifth line, we append a stylesheet to the page.
5. On the last line, we close the `head` block.

This file will be extended by all the other templates, so the stylesheet will be linked from all pages.

style.css

The second file is the `static/css/style.css` CSS stylesheet:

Download

You can download this example theme `here`.

5.4 Plugins

Beginning with version 3.0, Pelican supports plugins. Plugins are a way to add features to Pelican without having to directly modify the Pelican core.

5.4.1 How to use plugins

To load plugins, you have to specify them in your settings file. There are two ways to do so. The first method is to specify strings with the path to the callables:

```
PLUGINS = ['package.myplugin',]
```

Alternatively, another method is to import them and add them to the list:

```
from package import myplugin
PLUGINS = [myplugin,]
```

If your plugins are not in an importable path, you can specify a `PLUGIN_PATH` in the settings. `PLUGIN_PATH` can be an absolute path or a path relative to the settings file:

```
PLUGIN_PATH = "plugins"
PLUGINS = ["list", "of", "plugins"]
```

5.4.2 Where to find plugins

We maintain a separate repository of plugins for people to share and use. Please visit the pelican-plugins repository for a list of available plugins.

Please note that while we do our best to review and maintain these plugins, they are submitted by the Pelican community and thus may have varying levels of support and interoperability.

5.4.3 How to create plugins

Plugins are based on the concept of signals. Pelican sends signals, and plugins subscribe to those signals. The list of signals are defined in a subsequent section.

The only rule to follow for plugins is to define a `register` callable, in which you map the signals to your plugin logic. Let's take a simple example:

```
from pelican import signals

def test(sender):
    print "%s initialized !!" % sender

def register():
    signals.initialized.connect(test)
```

5.4.4 List of signals

Here is the list of currently implemented signals:

Signal	Arguments	Description
initialized	pelican object	
finalized	pelican object	invoked after all the generators are executed and just before pelican exits usefull for custom post processing actions, such as: - minifying js/css assets. - notify/ping search engines with an updated sitemap.
generator_init	generator	invoked in the Generator.__init__
readers_init	readers	invoked in the Readers.__init__
article_generator_context	article_generator, metadata	
article_generator_preread	article_generator	invoked before a article is read in ArticlesGenerator.generate_context; use if code needs to do something before every article is parsed
article_generator_init	article_generator	invoked in the ArticlesGenerator.__init__
article_generator_finalized	article_generator	invoked at the end of ArticlesGenerator.generate_context
get_generators	generators	invoked in Pelican.get_generator_classes, can return a Generator, or several generator in a tuple or in a list.
page_generate_context	page_generator, metadata	
page_generator_init	page_generator	invoked in the PagesGenerator.__init__
page_generator_finalized	page_generator	invoked at the end of PagesGenerator.generate_context
content_object_init	content_object	invoked at the end of Content.__init__ (see note below)
content_written	path, context	invoked each time a content file is written.

The list is currently small, so don't hesitate to add signals and make a pull request if you need them!

Note: The signal `content_object_init` can send a different type of object as the argument. If you want to register only one type of object then you will need to specify the sender when you are connecting to the signal.

```
from pelican import signals
from pelican import contents

def test(sender, instance):
        print "%s : %s content initialized !!" % (sender, instance)

def register():
        signals.content_object_init.connect(test, sender=contents.Article)
```

Note: After Pelican 3.2, signal names were standardized. Older plugins may need to be updated to use the new names:

Old name	New name
article_generate_context	article_generator_context
article_generate_finalized	article_generator_finalized
article_generate_preread	article_generator_preread
pages_generate_context	page_generator_context
pages_generate_preread	page_generator_preread
pages_generator_finalized	page_generator_finalized
pages_generator_init	page_generator_init
static_generate_context	static_generator_context
static_generate_preread	static_generator_preread

5.4.5 Recipes

We eventually realised some of the recipes to create plugins would be best shared in the documentation somewhere, so here they are!

How to create a new reader

One thing you might want is to add support for your very own input format. While it might make sense to add this feature in Pelican core, we wisely chose to avoid this situation and instead have the different readers defined via plugins.

The rationale behind this choice is mainly that plugins are really easy to write and don’t slow down Pelican itself when they’re not active.

No more talking — here is an example:

```python
from pelican import signals
from pelican.readers import BaseReader

# Create a new reader class, inheriting from the pelican.reader.BaseReader
class NewReader(BaseReader):
    enabled = True  # Yeah, you probably want that :-)

    # The list of file extensions you want this reader to match with.
    # If multiple readers were to use the same extension, the latest will
    # win (so the one you're defining here, most probably).
    file_extensions = ['yeah']

    # You need to have a read method, which takes a filename and returns
    # some content and the associated metadata.
    def read(self, filename):
        metadata = {'title': 'Oh yeah',
                    'category': 'Foo',
                    'date': '2012-12-01'}

        parsed = {}
        for key, value in metadata.items():
            parsed[key] = self.process_metadata(key, value)

        return "Some content", parsed

def add_reader(readers):
    readers.reader_classes['yeah'] = NewReader
```

```
# This is how pelican works.
def register():
    signals.readers_init.connect(add_reader)
```

Adding a new generator

Adding a new generator is also really easy. You might want to have a look at *Pelican internals* for more information on how to create your own generator.

```
def get_generators(generators):
    # define a new generator here if you need to
    return generators

signals.get_generators.connect(get_generators)
```

5.5 Pelican internals

This section describe how Pelican works internally. As you'll see, it's quite simple, but a bit of documentation doesn't hurt. :)

You can also find in the *Some history about Pelican* section an excerpt of a report the original author wrote with some software design information.

5.5.1 Overall structure

What Pelican does is take a list of files and process them into some sort of output. Usually, the input files are reStructuredText, Markdown and AsciiDoc files, and the output is a blog, but both input and output can be anything you want.

The logic is separated into different classes and concepts:

- **Writers** are responsible for writing files: .html files, RSS feeds, and so on. Since those operations are commonly used, the object is created once and then passed to the generators.
- **Readers** are used to read from various formats (AsciiDoc, HTML, Markdown and reStructuredText for now, but the system is extensible). Given a file, they return metadata (author, tags, category, etc.) and content (HTML-formatted).
- **Generators** generate the different outputs. For instance, Pelican comes with `ArticlesGenerator` and `PageGenerator`. Given a configuration, they can do whatever they want. Most of the time, it's generating files from inputs.
- Pelican also uses templates, so it's easy to write your own theme. The syntax is Jinja2 and is very easy to learn, so don't hesitate to jump in and build your own theme.

5.5.2 How to implement a new reader?

Is there an awesome markup language you want to add to Pelican? Well, the only thing you have to do is to create a class with a `read` method that returns HTML content and some metadata.

Take a look at the Markdown reader:

```
class MarkdownReader(BaseReader):
    enabled = bool(Markdown)

    def read(self, source_path):
        """Parse content and metadata of markdown files"""
        text = pelican_open(source_path)
        md = Markdown(extensions = ['meta', 'codehilite'])
        content = md.convert(text)

        metadata = {}
        for name, value in md.Meta.items():
            name = name.lower()
            meta = self.process_metadata(name, value[0])
            metadata[name] = meta
        return content, metadata
```

Simple, isn't it?

If your new reader requires additional Python dependencies, then you should wrap their `import` statements in a `try...except` block. Then inside the reader's class, set the `enabled` class attribute to mark import success or failure. This makes it possible for users to continue using their favourite markup method without needing to install modules for formats they don't use.

5.5.3 How to implement a new generator?

Generators have two important methods. You're not forced to create both; only the existing ones will be called.

- `generate_context`, that is called first, for all the generators. Do whatever you have to do, and update the global context if needed. This context is shared between all generators, and will be passed to the templates. For instance, the `PageGenerator` `generate_context` method finds all the pages, transforms them into objects, and populates the context with them. Be careful *not* to output anything using this context at this stage, as it is likely to change by the effect of other generators.
- `generate_output` is then called. And guess what is it made for? Oh, generating the output. :) It's here that you may want to look at the context and call the methods of the `writer` object that is passed as the first argument of this function. In the `PageGenerator` example, this method will look at all the pages recorded in the global context and output a file on the disk (using the writer method `write_file`) for each page encountered.

5.6 pelican-themes

5.6.1 Description

`pelican-themes` is a command line tool for managing themes for Pelican.

Usage

pelican-themes [-h] [-l] [-i theme path [theme path ...]]
[-r theme name [theme name ...]]
[-s theme path [theme path ...]] [-v] [–version]

Optional arguments:

-h, --help	Show the help an exit
-l, --list	Show the themes already installed
-i theme_path, --install theme_path	One or more themes to install
-r theme_name, --remove theme_name	One or more themes to remove
-s theme_path, --symlink theme_path	Same as "–install", but create a symbolic link instead of copying the theme. Useful for theme development
-v, --verbose	Verbose output
--version	Print the version of this script

5.6.2 Examples

Listing the installed themes

With `pelican-themes`, you can see the available themes by using the `-l` or `--list` option:

In this example, we can see there are three themes available: `notmyidea`, `simple`, and `two-column`.

`two-column` is prefixed with an `@` because this theme is not copied to the Pelican theme path, but is instead just linked to it (see Creating symbolic links for details about creating symbolic links).

Note that you can combine the `--list` option with the `-v` or `--verbose` option to get more verbose output, like this:

Installing themes

You can install one or more themes using the `-i` or `--install` option. This option takes as argument the path(s) of the theme(s) you want to install, and can be combined with the verbose option:

Removing themes

The `pelican-themes` command can also remove themes from the Pelican themes path. The `-r` or `--remove` option takes as argument the name(s) of the theme(s) you want to remove, and can be combined with the `--verbose` option.

Creating symbolic links

`pelican-themes` can also install themes by creating symbolic links instead of copying entire themes into the Pelican themes path.

To symbolically link a theme, you can use the `-s` or `--symlink`, which works exactly as the `--install` option:

In this example, the `two-column` theme is now symbolically linked to the Pelican themes path, so we can use it, but we can also modify it without having to reinstall it after each modification.

This is useful for theme development:

Doing several things at once

The `--install`, `--remove` and `--symlink` option are not mutually exclusive, so you can combine them in the same command line to do more than one operation at time, like this:

In this example, the theme `notmyidea-cms` is replaced by the theme `notmyidea-cms-fr`

5.6.3 See also

- http://docs.notmyidea.org/alexis/pelican/
- `/usr/share/doc/pelican/` if you have installed Pelican using the APT repository

5.7 Import from other blog software

5.7.1 Description

`pelican-import` is a command-line tool for converting articles from other software to reStructuredText or Markdown. The supported import formats are:

- WordPress XML export
- Dotclear export
- Posterous API
- Tumblr API
- RSS/Atom feed

The conversion from HTML to reStructuredText or Markdown relies on Pandoc. For Dotclear, if the source posts are written with Markdown syntax, they will not be converted (as Pelican also supports Markdown).

5.7.2 Dependencies

`pelican-import` has some dependencies not required by the rest of Pelican:

- *BeautifulSoup4* and *lxml*, for WordPress and Dotclear import. Can be installed like any other Python package (`pip install BeautifulSoup4 lxml`).
- *Feedparser*, for feed import (`pip install feedparser`).
- *Pandoc*, see the Pandoc site for installation instructions on your operating system.

5.7.3 Usage

```
pelican-import [-h] [--wpfile] [--dotclear] [--posterous] [--tumblr] [--feed] [-o OUTPUT]
               [-m MARKUP] [--dir-cat] [--dir-page] [--strip-raw] [--disable-slugs]
               [-e EMAIL] [-p PASSWORD] [-b BLOGNAME]
               input|api_token|api_key
```

Positional arguments

input The input file to read api_token [Posterous only] api_token can be obtained from http://posterous.com/api/ api_key [Tumblr only] api_key can be obtained from http://www.tumblr.com/oauth/apps

Optional arguments

-h, --help	Show this help message and exit
--wpfile	WordPress XML export (default: False)
--dotclear	Dotclear export (default: False)
--posterous	Posterous API (default: False)
--tumblr	Tumblr API (default: False)
--feed	Feed to parse (default: False)
-o OUTPUT, --output OUTPUT	Output path (default: output)
-m MARKUP, --markup MARKUP	Output markup format (supports rst & markdown) (default: rst)
--dir-cat	Put files in directories with categories name (default: False)
--dir-page	Put files recognised as pages in "pages/" sub- directory (wordpress import only) (default: False)
--filter-author	Import only post from the specified author.
--strip-raw	Strip raw HTML code that can't be converted to markup such as flash embeds or iframes (wordpress import only) (default: False)
--disable-slugs	Disable storing slugs from imported posts within output. With this disabled, your Pelican URLs may not be consistent with your original posts. (default: False)
-e EMAIL, --email=EMAIL	Email used to authenticate Posterous API
-p PASSWORD, --password=PASSWORD	Password used to authenticate Posterous API
-b BLOGNAME, --blogname=BLOGNAME	Blog name used in Tumblr API

5.7.4 Examples

For WordPress:

```
$ pelican-import --wpfile -o ~/output ~/posts.xml
```

For Dotclear:

```
$ pelican-import --dotclear -o ~/output ~/backup.txt
```

for Posterous:

```
$ pelican-import --posterous -o ~/output --email=<email_address> --password=<password> <api_token>
```

For Tumblr:

```
$ pelican-import --tumblr -o ~/output --blogname=<blogname> <api_token>
```

5.7.5 Tests

To test the module, one can use sample files:

- for WordPress: http://wpcandy.com/made/the-sample-post-collection
- for Dotclear: http://themes.dotaddict.org/files/public/downloads/lorem-backup.txt

5.8 Frequently Asked Questions (FAQ)

Here are some frequently asked questions about Pelican.

5.8.1 What's the best way to communicate a problem, question, or suggestion?

If you have a problem, question, or suggestion, please start by striking up a conversation on #pelican on Freenode. Those who don't have an IRC client handy can jump in immediately via IRC webchat. Because of differing time zones, you may not get an immediate response to your question, but please be patient and stay logged into IRC — someone will almost always respond if you wait long enough (it may take a few hours).

If you're unable to resolve your issue or if you have a feature request, please refer to the issue tracker.

5.8.2 How can I help?

There are several ways to help out. First, you can report any Pelican suggestions or problems you might have via IRC or the issue tracker. If submitting an issue report, please first check the existing issue list (both open and closed) in order to avoid submitting a duplicate issue.

If you want to contribute, please fork the git repository, create a new feature branch, make your changes, and issue a pull request. Someone will review your changes as soon as possible. Please refer to the *How to Contribute* section for more details.

You can also contribute by creating themes and improving the documentation.

5.8.3 Is it mandatory to have a configuration file?

Configuration files are optional and are just an easy way to configure Pelican. For basic operations, it's possible to specify options while invoking Pelican via the command line. See `pelican --help` for more information.

5.8.4 I'm creating my own theme. How do I use Pygments for syntax highlighting?

Pygments adds some classes to the generated content. These classes are used by themes to style code syntax highlighting via CSS. Specifically, you can customize the appearance of your syntax highlighting via the `.highlight pre` class in your theme's CSS file. To see how various styles can be used to render Django code, for example, use the style selector drop-down at top-right on the Pygments project demo site.

You can use the following example commands to generate a starting CSS file from a Pygments built-in style (in this case, "monokai") and then copy the generated CSS file to your new theme:

```
pygmentize -S monokai -f html -a .highlight > pygment.css
cp pygment.css path/to/theme/static/css/
```

Don't forget to import your `pygment.css` file from your main CSS file.

5.8.5 How do I create my own theme?

Please refer to *How to create themes for Pelican*.

5.8.6 I want to use Markdown, but I got an error.

Markdown is not a hard dependency for Pelican, so you will need to explicitly install it. You can do so by typing the following command, prepending `sudo` if permissions require it:

```
pip install markdown
```

If you don't have `pip` installed, consider installing it via:

```
easy_install pip
```

5.8.7 Can I use arbitrary metadata in my templates?

Yes. For example, to include a modified date in a Markdown post, one could include the following at the top of the article:

```
Modified: 2012-08-08
```

For reStructuredText, this metadata should of course be prefixed with a colon:

```
:Modified: 2012-08-08
```

This metadata can then be accessed in templates such as `article.html` via:

```
{% if article.modified %}
Last modified: {{ article.modified }}
{% endif %}
```

If you want to include metadata in templates outside the article context (e.g., `base.html`), the `if` statement should instead be:

```
{% if article and article.modified %}
```

5.8.8 How do I assign custom templates on a per-page basis?

It's as simple as adding an extra line of metadata to any page or article that you want to have its own template. For example, this is how it would be handled for content in reST format:

```
:template: template_name
```

For content in Markdown format:

```
Template: template_name
```

Then just make sure your theme contains the relevant template file (e.g. `template_name.html`).

5.8.9 How can I override the generated URL of a specific page or article?

Include `url` and `save_as` metadata in any pages or articles that you want to override the generated URL. Here is an example page in reST format:

```
Override url/save_as page
#########################

:url: override/url/
:save_as: override/url/index.html
```

With this metadata, the page will be written to `override/url/index.html` and Pelican will use url `override/url/` to link to this page.

5.8.10 How can I use a static page as my home page?

The override feature mentioned above can be used to specify a static page as your home page. The following Markdown example could be stored in `content/pages/home.md`:

```
Title: Welcome to My Site
URL:
save_as: index.html

Thank you for visiting. Welcome!
```

5.8.11 What if I want to disable feed generation?

To disable feed generation, all feed settings should be set to `None`. All but three feed settings already default to `None`, so if you want to disable all feed generation, you only need to specify the following settings:

```
FEED_ALL_ATOM = None
CATEGORY_FEED_ATOM = None
TRANSLATION_FEED_ATOM = None
```

Please note that `None` and `''` are not the same thing. The word `None` should not be surrounded by quotes.

5.8.12 I'm getting a warning about feeds generated without SITEURL being set properly

RSS and Atom feeds require all URL links to be absolute. In order to properly generate links in Pelican you will need to set `SITEURL` to the full path of your site.

Feeds are still generated when this warning is displayed, but links within may be malformed and thus the feed may not validate.

5.8.13 My feeds are broken since I upgraded to Pelican 3.x

Starting in 3.0, some of the FEED setting names were changed to more explicitly refer to the Atom feeds they inherently represent (much like the FEED_RSS setting names). Here is an exact list of the renamed settings:

```
FEED -> FEED_ATOM
TAG_FEED -> TAG_FEED_ATOM
CATEGORY_FEED -> CATEGORY_FEED_ATOM
```

Starting in 3.1, the new feed `FEED_ALL_ATOM` has been introduced: this feed will aggregate all posts regardless of their language. This setting generates `'feeds/all.atom.xml'` by default and `FEED_ATOM` now defaults to `None`. The following feed setting has also been renamed:

```
TRANSLATION_FEED -> TRANSLATION_FEED_ATOM
```

Older themes that referenced the old setting names may not link properly. In order to rectify this, please update your theme for compatibility by changing the relevant values in your template files. For an example of complete feed headers and usage please check out the `simple` theme.

5.8.14 Is Pelican only suitable for blogs?

No. Pelican can be easily configured to create and maintain any type of static site. This may require a little customization of your theme and Pelican configuration. For example, if you are building a launch site for your product and do not need tags on your site, you could remove the relevant HTML code from your theme. You can also disable generation of tag-related pages via:

```
TAGS_SAVE_AS = ''
TAG_SAVE_AS = ''
```

5.9 Tips

Here are some tips about Pelican that you might find useful.

5.9.1 Publishing to GitHub

GitHub Pages offer an easy and convenient way to publish Pelican sites. There are two types of GitHub Pages: *Project Pages* and *User Pages*. Pelican sites can be published as both Project Pages and User Pages.

Project Pages

To publish a Pelican site as a Project Page you need to *push* the content of the `output` dir generated by Pelican to a repository's `gh-pages` branch on GitHub.

The excellent ghp-import, which can be installed with `easy_install` or `pip`, makes this process really easy.

For example, if the source of your Pelican site is contained in a GitHub repository, and if you want to publish that Pelican site in the form of Project Pages to this repository, you can then use the following:

```
$ pelican content -o output -s pelicanconf.py
$ ghp-import output
$ git push origin gh-pages
```

The `ghp-import output` command updates the local `gh-pages` branch with the content of the `output` directory (creating the branch if it doesn't already exist). The `git push origin gh-pages` command updates the remote `gh-pages` branch, effectively publishing the Pelican site.

Note: The `github` target of the Makefile created by the `pelican-quickstart` command publishes the Pelican site as Project Pages, as described above.

User Pages

To publish a Pelican site in the form of User Pages, you need to *push* the content of the `output` dir generated by Pelican to the `master` branch of your `<username>.github.io` repository on GitHub.

Again, you can take advantage of `ghp-import`:

```
$ pelican content -o output -s pelicanconf.py
$ ghp-import output
$ git push git@github.com:elemoine/elemoine.github.io.git gh-pages:master
```

The `git push` command pushes the local `gh-pages` branch (freshly updated by the `ghp-import` command) to the `elemoine.github.io` repository's `master` branch on GitHub.

Note: To publish your Pelican site as User Pages, feel free to adjust the `github` target of the Makefile.

Extra Tips

Tip #1:

To automatically update your Pelican site on each commit, you can create a post-commit hook. For example, you can add the following to `.git/hooks/post-commit`:

```
pelican content -o output -s pelicanconf.py && ghp-import output && git push origin gh-pages
```

Tip #2:

To use a custom domain with GitHub Pages, you need to put the domain of your site (e.g., `blog.example.com`) inside a `CNAME` file at the root of your site. To do this, create the `content/extras/` directory and add a `CNAME` file to it. Then use the `STATIC_PATHS` setting to tell Pelican to copy this file to your output directory. For example:

```
STATIC_PATHS = ['images', 'extra/CNAME']
EXTRA_PATH_METADATA = {'extra/CNAME': {'path': 'CNAME'},}
```

5.9.2 How to add YouTube or Vimeo Videos

The easiest way is to paste the embed code of the video from these sites directly into your source content.

Alternatively, you can also use Pelican plugins like `liquid_tags`, `pelican_youtube`, or `pelican_vimeo` to embed videos in your content.

5.10 How to contribute

There are many ways to contribute to Pelican. You can improve the documentation, add missing features, and fix bugs (or just report them). You can also help out by reviewing and commenting on existing issues.

Don't hesitate to fork Pelican and submit a pull request on GitHub. When doing so, please adhere to the following guidelines.

5.10.1 Contribution submission guidelines

- Consider whether your new feature might be better suited as a plugin. Folks are usually available in the #pelican IRC channel if help is needed to make that determination.

- Create a new git branch specific to your change (as opposed to making your commits in the master branch).
- **Don't put multiple fixes/features in the same branch / pull request.** For example, if you're hacking on a new feature and find a bugfix that doesn't *require* your new feature, **make a new distinct branch and pull request** for the bugfix.
- Adhere to PEP8 coding standards whenever possible.
- Check for unnecessary whitespace via `git diff --check` before committing.
- **Add docs and tests for your changes**.
- Run all the tests **on both Python 2.7 and 3.3** to ensure nothing was accidentally broken.
- First line of your commit message should start with present-tense verb, be 50 characters or less, and include the relevant issue number(s) if applicable. *Example:* `Ensure proper PLUGIN_PATH behavior. Refs #428.` If the commit *completely fixes* an existing bug report, please use `Fixes #585` or `Fix #585` syntax (so the relevant issue is automatically closed upon PR merge).
- After the first line of the commit message, add a blank line and then a more detailed explanation (when relevant).
- If you have previously filed a GitHub issue and want to contribute code that addresses that issue, **please use** `hub pull-request` instead of using GitHub's web UI to submit the pull request. This isn't an absolute requirement, but makes the maintainers' lives much easier! Specifically: install hub and then run hub pull-request to turn your GitHub issue into a pull request containing your code.

Check out our Git Tips page or ask on the #pelican IRC channel if you need assistance or have any questions about these guidelines.

5.10.2 Setting up the development environment

While there are many ways to set up one's development environment, following is a method that uses virtualenv. If you don't have `virtualenv` installed, you can install it via:

```
$ pip install virtualenv
```

Virtual environments allow you to work on Python projects which are isolated from one another so you can use different packages (and package versions) with different projects.

To create and activate a virtual environment, use the following syntax:

```
$ virtualenv ~/virtualenvs/pelican
$ cd ~/virtualenvs/pelican
$ . bin/activate
```

To clone the Pelican source:

```
$ git clone https://github.com/getpelican/pelican.git src/pelican
```

To install the development dependencies:

```
$ cd src/pelican
$ pip install -r dev_requirements.txt
```

To install Pelican and its dependencies:

```
$ python setup.py develop
```

Or using `pip`:

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

5.10.3 Coding standards

Try to respect what is described in the PEP8 specification when making contributions. This can be eased via the pep8 or flake8 tools, the latter of which in particular will give you some useful hints about ways in which the code/formatting can be improved.

5.10.4 Building the docs

If you make changes to the documentation, you should preview your changes before committing them:

```
$ pip install sphinx
$ cd src/pelican/docs
$ make html
```

Open `_build/html/index.html` in your browser to preview the documentation.

5.10.5 Running the test suite

Each time you add a feature, there are two things to do regarding tests: check that the existing tests pass, and add tests for the new feature or bugfix.

The tests live in `pelican/tests` and you can run them using the "discover" feature of `unittest`:

```
$ python -m unittest discover
```

After making your changes and running the tests, you may see a test failure mentioning that "some generated files differ from the expected functional tests output." If you have made changes that affect the HTML output generated by Pelican, and the changes to that output are expected and deemed correct given the nature of your changes, then you should update the output used by the functional tests. To do so, you can use the following two commands:

```
$ LC_ALL=en_US.utf8 pelican -o pelican/tests/output/custom/ \
    -s samples/pelican.conf.py samples/content/
$ LC_ALL=en_US.utf8 pelican -o pelican/tests/output/basic/ \
    samples/content/
```

Testing on Python 2 and 3

Testing on Python 3 currently requires some extra steps: installing Python 3-compatible versions of dependent packages and plugins.

Tox is a useful tool to run tests on both versions. It will install the Python 3-compatible version of dependent packages.

5.10.6 Python 3 development tips

Here are some tips that may be useful when doing some code for both Python 2.7 and Python 3 at the same time:

- Assume every string and literal is unicode (import unicode_literals):
  - Do not use prefix `u'`.

  - Do not encode/decode strings in the middle of sth. Follow the code to the source (or target) of a string and encode/decode at the first/last possible point.
  - In other words, write your functions to expect and to return unicode.
  - Encode/decode strings if e.g. the source is a Python function that is known to handle this badly, e.g. strftime() in Python 2.
- Use new syntax: print function, "except ... *as* e" (not comma) etc.
- Refactor method calls like `dict.iteritems()`, `xrange()` etc. in a way that runs without code change in both Python versions.
- Do not use magic method `__unicode()__` in new classes. Use only `__str()__` and decorate the class with `@python_2_unicode_compatible`.
- Do not start int literals with a zero. This is a syntax error in Py3k.
- Unfortunately I did not find an octal notation that is valid in both Pythons. Use decimal instead.
- use six, e.g.:
  - `isinstance(.., basestring) -> isinstance(.., six.string_types)`
  - `isinstance(.., unicode) -> isinstance(.., six.text_type)`
- `setlocale()` in Python 2 bails when we give the locale name as unicode, and since we are using `from __future__ import unicode_literals`, we do that everywhere! As a workaround, I enclosed the localename with `str()`; in Python 2 this casts the name to a byte string, in Python 3 this should do nothing, because the locale name already had been unicode.
- Kept range() almost everywhere as-is (2to3 suggests list(range())), just changed it where I felt necessary.
- Changed xrange() back to range(), so it is valid in both Python versions.

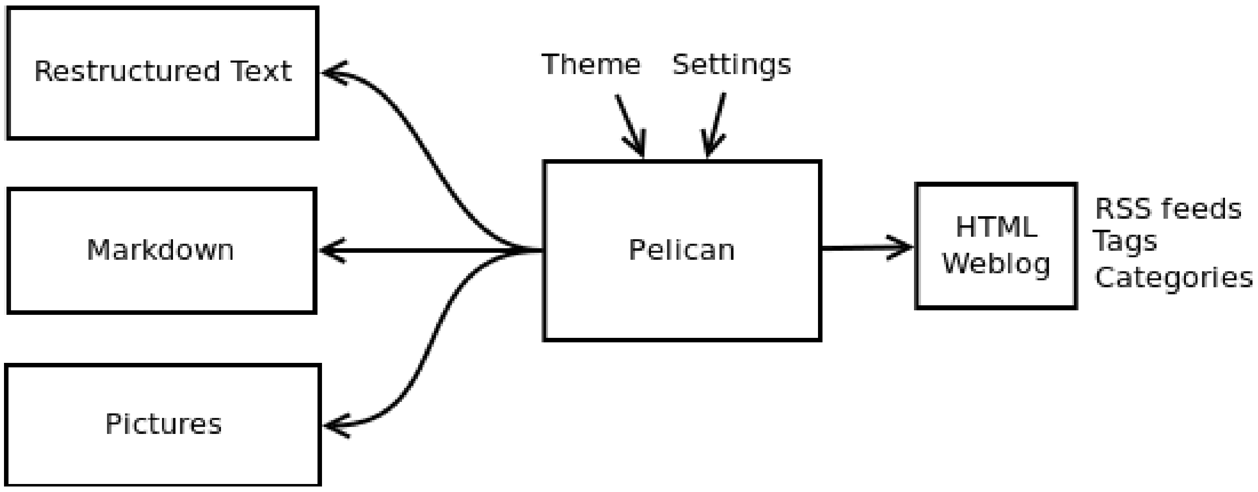
5.11 Some history about Pelican

Warning: This page comes from a report the original author (Alexis Métaireau) wrote right after writing Pelican, in December 2010. The information may not be up-to-date.

Pelican is a simple static blog generator. It parses markup files (Markdown or reStructuredText for now) and generates an HTML folder with all the files in it. I've chosen to use Python to implement Pelican because it seemed to be simple and to fit to my needs. I did not wanted to define a class for each thing, but still wanted to keep my things loosely coupled. It turns out that it was exactly what I wanted. From time to time, thanks to the feedback of some users, it took me a very few time to provide fixes on it. So far, I've re-factored the Pelican code by two times; each time took less than 30 minutes.

5.11.1 Use case

I was previously using WordPress, a solution you can host on a web server to manage your blog. Most of the time, I prefer using markup languages such as Markdown or reStructuredText to type my articles. To do so, I use vim. I think it is important to let the people choose the tool they want to write the articles. In my opinion, a blog manager should just allow you to take any kind of input and transform it to a weblog. That's what Pelican does. You can write your articles using the tool you want, and the markup language you want, and then generate a static HTML weblog.



To be flexible enough, Pelican has template support, so you can easily write your own themes if you want to.

5.11.2 Design process

Pelican came from a need I have. I started by creating a single file application, and I have make it grow to support what it does by now. To start, I wrote a piece of documentation about what I wanted to do. Then, I created the content I wanted to parse (the reStructuredText files) and started experimenting with the code. Pelican was 200 lines long and contained almost ten functions and one class when it was first usable.

I have been facing different problems all over the time and wanted to add features to Pelican while using it. The first change I have done was to add the support of a settings file. It is possible to pass the options to the command line, but can be tedious if there is a lot of them. In the same way, I have added the support of different things over time: Atom feeds, multiple themes, multiple markup support, etc. At some point, it appears that the "only one file" mantra was not good enough for Pelican, so I decided to rework a bit all that, and split this in multiple different files.

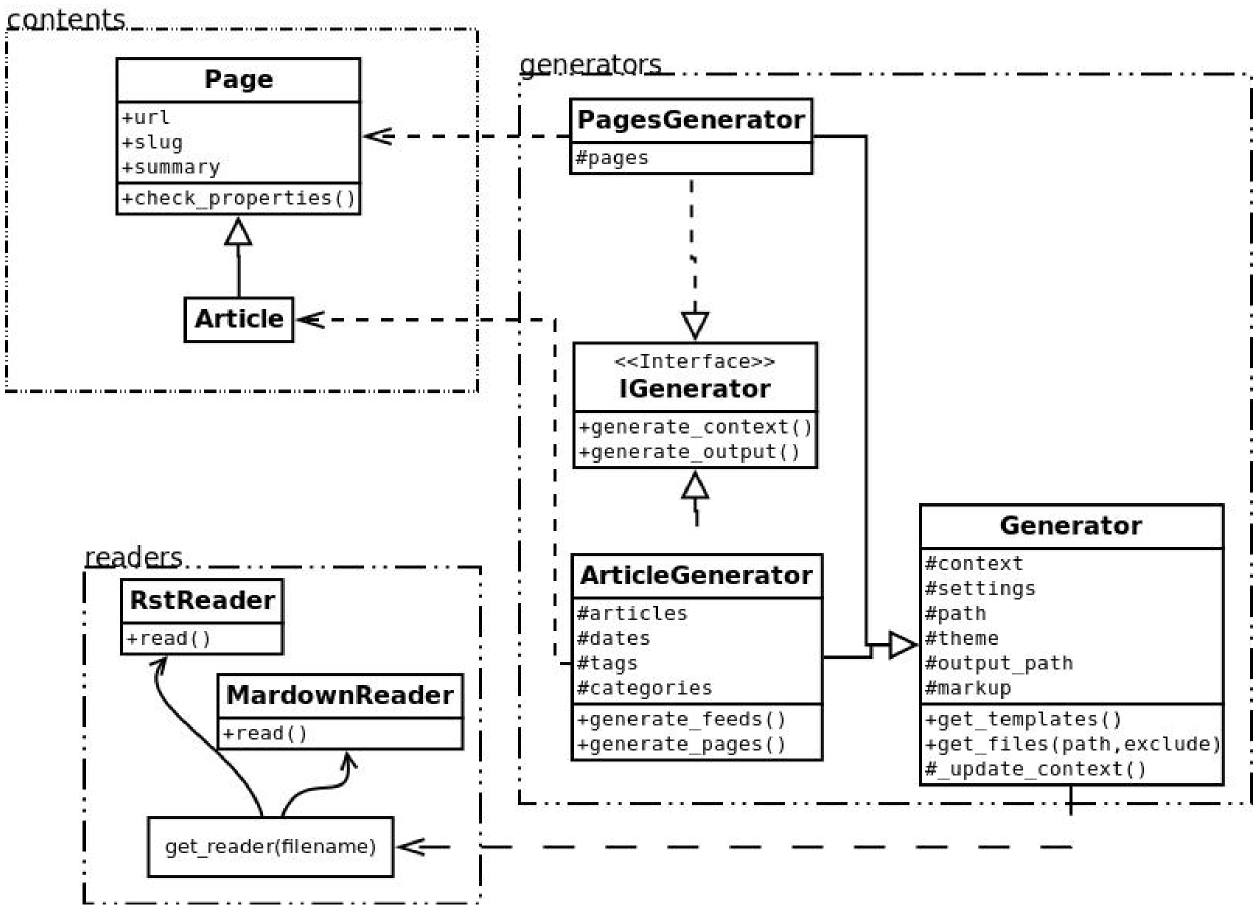
I've separated the logic in different classes and concepts:

- *writers* are responsible of all the writing process of the files. They are responsible of writing .html files, RSS feeds and so on. Since those operations are commonly used, the object is created once, and then passed to the generators.
- *readers* are used to read from various formats (Markdown and reStructuredText for now, but the system is extensible). Given a file, they return metadata (author, tags, category, etc) and content (HTML formatted).
- *generators* generate the different outputs. For instance, Pelican comes with an ArticlesGenerator and PagesGenerator, into others. Given a configuration, they can do whatever you want them to do. Most of the time it's generating files from inputs (user inputs and files).

I also deal with contents objects. They can be `Articles`, `Pages`, `Quotes`, or whatever you want. They are defined in the `contents.py` module and represent some content to be used by the program.

5.11.3 In more detail

Here is an overview of the classes involved in Pelican.



The interface does not really exist, and I have added it only to clarify the whole picture. I do use duck typing and not interfaces.

Internally, the following process is followed:

- First of all, the command line is parsed, and some content from the user is used to initialize the different generator objects.
- A `context` is created. It contains the settings from the command line and a settings file if provided.
- The `generate_context` method of each generator is called, updating the context.
- The writer is created and given to the `generate_output` method of each generator.

I make two calls because it is important that when the output is generated by the generators, the context will not change. In other words, the first method `generate_context` should modify the context, whereas the second `generate_output` method should not.

Then, it is up to the generators to do what the want, in the `generate_context` and `generate_content` method. Taking the `ArticlesGenerator` class will help to understand some others concepts. Here is what happens when calling the `generate_context` method:

- Read the folder "path", looking for restructured text files, load each of them, and construct a content object (`Article`) with it. To do so, use `Reader` objects.
- Update the `context` with all those articles.

Then, the `generate_content` method uses the `context` and the `writer` to generate the wanted output.

5.12 Release history

5.12.1 3.3.0 (2013-09-24)

- Drop Python 3.2 support in favor of Python 3.3
- Add `Fabfile` so Fabric can be used for workflow automation instead of Make
- `OUTPUT_RETENTION` setting can be used to preserve metadata (e.g., VCS data such as `.hg` and `.git`) from being removed from output directory
- Tumblr import
- Improve logic and consistency when cleaning output folder
- Improve documentation versioning and release automation
- Improve pagination flexibility
- Rename signals for better consistency (some plugins may need to be updated)
- Move metadata extraction from generators to readers; metadata extraction no longer article-specific
- Deprecate `FILES_TO_COPY` in favor of `STATIC_PATHS` and `EXTRA_PATH_METADATA`
- Summaries in Markdown posts no longer include footnotes
- Remove unnecessary whitespace in output via `lstrip_blocks` Jinja parameter
- Move PDF generation from core to plugin
- Replace `MARKUP` setting with `READERS`
- Add warning if img tag is missing `alt` attribute
- Add support for `{}` in relative links syntax, besides `||`
- Add support for `{tag}` and `{category}` relative links
- Add a `content_written` signal

5.12.2 3.2.1 and 3.2.2

- Facilitate inclusion in FreeBSD Ports Collection

5.12.3 3.2 (2013-04-24)

- Support for Python 3!
- Override page save-to location from meta-data (enables using a static page as the site's home page, for example)
- Time period archives (per-year, per-month, and per-day archives of posts)
- Posterous blog import
- Improve WordPress blog import
- Migrate plugins to separate repository
- Improve HTML parser
- Provide ability to show or hide categories from menu using `DISPLAY_CATEGORIES_ON_MENU` option
- Auto-regeneration can be told to ignore files via `IGNORE_FILES` setting

- Improve post-generation feedback to user
- For multilingual posts, use meta-data to designate which is the original and which is the translation
- Add `.mdown` to list of supported Markdown file extensions
- Document-relative URL generation (`RELATIVE_URLS`) is now off by default

5.12.4 3.1 (2012-12-04)

- Importer now stores slugs within files by default. This can be disabled with the `--disable-slugs` option.
- Improve handling of links to intra-site resources
- Ensure WordPress import adds paragraphs for all types of line endings in post content
- Decode HTML entities within WordPress post titles on import
- Improve appearance of LinkedIn icon in default theme
- Add GitHub and Google+ social icons support in default theme
- Optimize social icons
- Add `FEED_ALL_ATOM` and `FEED_ALL_RSS` to generate feeds containing all posts regardless of their language
- Split `TRANSLATION_FEED` into `TRANSLATION_FEED_ATOM` and `TRANSLATION_FEED_RSS`
- Different feeds can now be enabled/disabled individually
- Allow for blank author: if `AUTHOR` setting is not set, author won't default to `${USER}` anymore, and a post won't contain any author information if the post author is empty
- Move LESS and Webassets support from Pelican core to plugin
- The `DEFAULT_DATE` setting now defaults to `None`, which means that articles won't be generated unless date metadata is specified
- Add `FILENAME_METADATA` setting to support metadata extraction from filename
- Add `gzip_cache` plugin to compress common text files into a `.gz` file within the same directory as the original file, preventing the server (e.g. Nginx) from having to compress files during an HTTP call
- Add support for AsciiDoc-formatted content
- Add `USE_FOLDER_AS_CATEGORY` setting so that feature can be toggled on/off
- Support arbitrary Jinja template files
- Restore basic functional tests
- New signals: `generator_init`, `get_generators`, and `article_generate_preread`

5.12.5 3.0 (2012-08-08)

- Refactored the way URLs are handled
- Improved the English documentation
- Fixed packaging using `setuptools` entrypoints
- Added `typogrify` support
- Added a way to disable feed generation

- Added support for `DIRECT_TEMPLATES`
- Allow multiple extensions for content files
- Added LESS support
- Improved the import script
- Added functional tests
- Rsync support in the generated Makefile
- Improved feed support (easily pluggable with Feedburner for instance)
- Added support for `abbr` in reST
- Fixed a bunch of bugs :-)

5.12.6 2.8 (2012-02-28)

- Dotclear importer
- Allow the usage of Markdown extensions
- Themes are now easily extensible
- Don’t output pagination information if there is only one page
- Add a page per author, with all their articles
- Improved the test suite
- Made the themes easier to extend
- Removed Skribit support
- Added a `pelican-quickstart` script
- Fixed timezone-related issues
- Added some scripts for Windows support
- Date can be specified in seconds
- Never fail when generating posts (skip and continue)
- Allow the use of future dates
- Support having different timezones per language
- Enhanced the documentation

5.12.7 2.7 (2011-06-11)

- Use `logging` rather than echoing to stdout
- Support custom Jinja filters
- Compatibility with Python 2.5
- Added a theme manager
- Packaged for Debian
- Added draft support

5.12.8 2.6 (2011-03-08)

- Changes in the output directory structure
- Makes templates easier to work with / create
- Added RSS support (was Atom-only)
- Added tag support for the feeds
- Enhance the documentation
- Added another theme (brownstone)
- Added translations
- Added a way to use cleaner URLs with a rewrite url module (or equivalent)
- Added a tag cloud
- Added an autoreloading feature: the blog is automatically regenerated each time a modification is detected
- Translate the documentation into French
- Import a blog from an RSS feed
- Pagination support
- Added Skribit support

5.12.9 2.5 (2010-11-20)

- Import from Wordpress
- Added some new themes (martyalchin / wide-notmyidea)
- First bug report!
- Linkedin support
- Added a FAQ
- Google Analytics support
- Twitter support
- Use relative URLs, not static ones

5.12.10 2.4 (2010-11-06)

- Minor themes changes
- Add Disqus support (so we have comments)
- Another code refactoring
- Added config settings about pages
- Blog entries can also be generated in PDF

5.12.11 2.3 (2010-10-31)

- Markdown support

5.12.12 2.2 (2010-10-30)

- Prettify output
- Manages static pages as well

5.12.13 2.1 (2010-10-30)

- Make notmyidea the default theme

5.12.14 2.0 (2010-10-30)

- Refactoring to be more extensible
- Change into the setting variables

5.12.15 1.2 (2010-09-28)

- Added a debug option
- Added per-category feeds
- Use filesystem to get dates if no metadata is provided
- Add Pygments support

5.12.16 1.1 (2010-08-19)

- First working version