Pelican Documentation

Release 2

Alexis Métaireau

July 02, 2015

Contents

Pelican is a simple weblog generator, written in python.

- Write your weblog entries directly with your editor of choice (vim!) and directly in restructured text, or markdown.
- A simple cli-tool to (re)generate the weblog.
- Easy to interface with DVCSes and web hooks
- Completely static output, so easy to host anywhere !

CHAPTER 1

Features

Pelican currently supports:

- blog articles and simple pages
- comments, via an external service (disqus). Please notice that while it's useful, it's an external service, and you'll not manage the comments by yourself. It could potentially eat your data. (optional)
- easy theming (themes are done using jinja2)
- PDF generation of the articles/pages (optional).
- publication of articles in various languages
- RSS/Atom feeds
- wordpress/dotclear or RSS imports
- integration with various tools: twitter/google analytics (optional)

CHAPTER 2

Why the name “Pelican” ?

Heh, you didn’t noticed? “Pelican” is an anagram for “Calepin” ;)

CHAPTER 3

Source code

You can access the source code via git on http://github.com/ametaireau/pelican/

CHAPTER 4

Feedback / Contact us

If you want to see new features in Pelican, dont hesitate to tell me, to clone the repository, etc. That's open source, dude!

Contact me at "alexis at notmyidea dot org" for any request/feedback! You can also join the team at #pelican on irc.freenode.org (or if you don't have any IRC client, using the webchat) for quick feedback.

CHAPTER 5

Documentation

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

5.1 Getting started

5.1.1 Installing

You're ready? Let's go ! You can install pelican in a lot of different ways, the simpler one is via pip:

```
$ pip install pelican
```

If you have the sources, you can install pelican using the distutils command install. I recommend to do so in a virtualenv:

```
$ virtualenv pelican_venv
$ source bin/activate
$ python setup.py install
```

Dependencies

At this time, pelican is dependent of the following python packages:

- feedgenerator, to generate the ATOM feeds.
- jinja2, for templating support.

If you're not using python 2.7, you will also need *argparse*.

Optionally:

- docutils, for reST support
- pygments, to have syntactic colorization with resT input
- Markdown, for Markdown as an input format

5.1.2 Writing articles using pelican

Files metadata

Pelican tries to be smart enough to get the informations it needs from the file system (for instance, about the category of your articles), but you need to provide by hand some of those informations in your files.

You could provide the metadata in the restructured text files, using the following syntax (give your file the *.rst* extension):

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

:date: 2010-10-03 10:20
:tags: thats, awesome
:category: yeah
:author: Alexis Metaireau
```

You can also use a markdown syntax (with a file ending in *.md*):

```
Date: 2010-12-03
Title: My super title

Put you content here.
```

Note that none of those are mandatory: if the date is not specified, pelican will rely on the mtime of your file, and the category can also be determined by the directory where the rst file is. For instance, the category of *python/foobar/myfoobar.rst* is *foobar*.

Generate your blog

To launch pelican, just use the *pelican* command:

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

And... that's all! You can see your weblog generated on the *content/* folder.

This one will just generate a simple output, with the default theme. It's not really sexy, as it's a simple HTML output (without any style).

You can create your own style if you want, have a look to the help to see all the options you can use:

```
$ pelican --help
```

Kickstart a blog

You also can use the *pelican-quickstart* script to start a new blog in seconds, by just answering few questions. Just run *pelican-quickstart* and you're done! (Added in pelican 3)

Pages

If you create a folder named *pages*, all the files in it will be used to generate static pages.

Then, use the *DISPLAY_PAGES_ON_MENU* setting, which will add all the pages to the menu.

Importing an existing blog

It is possible to import your blog from dotclear, wordpress and an RSS feed 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 in your articles/pages, and to set a *DEFAULT_LANG* setting (which is en by default). Then, only articles with this default language will be listed, and each article will have a translation list.

Pelican uses the "slug" of two articles to compare if they are translations of each others. So it's possible to define (in restructured text) the slug directly.

Here is an exemple of two articles (one in english and the other one in french).

The english one:

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

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

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

And the french one:

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

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

Oui oui, foobar est toujours vivant !
```

Despite the text quality, you can see that only the slug is the same here. You're not forced to define the slug that way, and it's completely possible to have two translations with the same title (which defines the slug)

Syntactic recognition

Pelican is able to regognise the syntax you are using, and to colorize the right way your block codes. To do so, you have to use the following syntax:

```
.. code-block:: identifier

   your code goes here
```

The identifier is one of the lexers available here.

You also can use the default *::* syntax:

```
::

    your code goes here
```

It will be assumed that your code is witten in python.

Autoreload

It's possible to tell pelican to watch for your modifications, instead of manually launching it each time you need. Use the *-r* option, or *–autoreload*.

Publishing drafts

If you want to publish an article as a draft, for friends to review it for instance, you can add a `status: draft` to its metadata, it will then be available under the `drafts` folder, and not be listed under the index page nor any category page.

Viewing the generated files

The files generated by pelican are static files, so you don't actually need something special to see what's hapenning with the generated files.

You can either run your browser on the files on your disk:

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

Or run a simple web server using python:

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

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 given as the form of a python module (a file). You can have an example by looking at /samples/pelican.conf.py

All the settings identifiers must be set in caps, otherwise they will not be processed.

The settings you define in the configuration file will be passed to the templates, it allows you to use them to add site-wide contents if you need.

Here is a list of settings for pelican, regarding the different features.

5.2.1 Basic settings

Setting name (default value)	what does it do?
ARTICLE_PERMALINK_STRUCTURE (`''`)	Empty by default. Allows to render URLs in a particular way, see below.
AUTHOR	Default author (put your name)
CLEAN_URLS (`False`)	If set to *True*, the URLs will not be suffixed by *.html*, so you will have to setup URL rewriting on your web server.
DATE_FORMATS (`{}`)	If you do manage multiple languages, you can set the date formatting here. See "Date format and locales" section below for details.
DEFAULT_CATEGORY (`'misc'`)	The default category to fallback on.
DEFAULT_DATE_FORMAT (`'%a %d %B %Y'`)	The default date format you want to use.
DISPLAY_PAGES_ON_MENU (`True`)	Display or not the pages on the menu of the template. Templates can follow or not this settings.
FALLBACK_ON_FS_DATE (`True`)	If True, pelican will use the file system dates infos (mtime) if it can't get informations from the metadata
JINJA_EXTENSIONS (`[]`)	A list of any Jinja2 extensions you want to use.
DELETE_OUTPUT_DIRECTORY (`False`)	Delete the output directory and just the generated files.
LOCALE (''[1])	Change the locale. A list of locales can be provided here or a single string representing one locale. When providing a list, all the locales will be tried until one works.
MARKUP (`('rst', 'md')`)	A list of available markup languages you want to use. For the moment, only available values are *rst* and *md*.
MD_EXTENSIONS (`('codehilite','extra')`)	A list of the extensions that the markdown processor will use. Refer to the extensions chapter in the Python-Markdown documentation for a complete list of supported extensions.
OUTPUT_PATH (`'output/'`)	Where to output the generated files.
PATH (`None`)	path to look at for input files.
PDF_GENERATOR (`False`)	Set to True if you want to have PDF versions of your documents. You will need to install *rst2pdf*.
RELATIVE_URLS (`True`)	Defines if pelican should use relative urls or not.
SITENAME (`'A Pelican Blog'`)	Your site name
SITEURL	base URL of your website. Note that this is not a way to tell pelican to use relative urls or static ones. You should rather use the *RELATIVE_URL* setting for such use.
STATIC_PATHS (`['images']`)	The static paths you want to have accessible on the output path "static". By default, pelican will copy the 'images' folder to the output folder.
TIMEZONE	The timezone used in the date information, to generate atom and rss feeds. See the "timezone" section below for more info.

Article permalink structure

Allow to render articles sorted by date, in case you specify a format as specified in the example. It follows the python datetime directives:

- %Y: Year with century as a decimal number.

- %m: Month as a decimal number [01,12].
- %d: Day of the month as a decimal number [01,31].

Note: if you specify a datetime directive, it will be substituted using the date metadata field into the rest file. if the date is not specified, pelican will rely on the mtime of your file.

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

Also, you can use any metadata in the restructured text files:

- category: '%(category)s'
- author: '%(author)s'
- tags: '%(tags)s'
- date: '%(date)s'

Example usage:

- '/%Y/%m/' it will be something like '/2011/07/sample-post.html'.
- '/%Y/%(category)s/' it will be something like '/2011/life/sample-post.html'.

Timezone

If no timezone is defined, UTC is assumed. This means that the generated atom and rss feeds will have wrong 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 old versions.

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

Date format and locale

If no DATE_FORMAT is set, fallback to DEFAULT_DATE_FORMAT. If you need to maintain multiple languages with different date format, you can set this dict using language name (`lang` in your posts) as key. About available format codes, see strftime document of python :

DATE_FORMAT = { '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) tuple in dict, and this will override the LOCALE setting above:

On Unix/Linux DATE_FORMAT = {

'en': ('en_US','%a, %d %b %Y'), 'jp': ('ja_JP','%Y-%m-%d(%a)'),

}

On Windows DATE_FORMAT = {

'en': ('usa','%a, %d %b %Y'), 'jp': ('jpn','%Y-%m-%d(%a)'),

}

For available list of locales on Windows . On Unix/Linux usually you can get a list of available locales with command `locale -a`, see manpage locale(1) for help.

5.2.2 Feed settings

By default, pelican uses atom feeds. However, it is possible to use RSS feeds instead, at your covenience.

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

Setting name (default value)	what does it do?
CATEGORY_FEED ('feeds/%s.atom.xml'[2])	Where to put the atom categories feeds.
CATEGORY_FEED_RSS (`None`, i.e. no RSS)	Where to put the categories rss feeds.
FEED (`'feeds/all.atom.xml'`)	relative url to output the atom feed.
FEED_RSS (`None`, i.e. no RSS)	relative url to output the rss feed.
TAG_FEED (`None`, ie no tag feed)	relative url to output the tags atom feed. It should be defined using a "%s" matchin 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. Feeds are unrestricted by default.

5.2.3 Pagination

The default behaviour of pelican is to list all the articles titles alongside with a short description of them on the index page. While it works pretty well for little to medium blogs, it is convenient to have a way to paginate this.

You can use the following settings to configure the pagination.

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 to have a last page with very few articles.
DEFAULT_PAGINATION (False)	The maximum number of articles to include on a page, not including orphans. False to disable pagination.

5.2.4 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 tags count in the cloud.

The default theme does not support tag clouds, but it is pretty easy to add:

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

You should then also define a CSS with the appropriate classes (tag-0 to tag-N, where N matches *TAG_CLOUD_STEPS* -1.

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

5.2.5 Translations

Pelican offers a way to translate articles. See the section on getting started for more information about that.

Setting name (default value)	what does it do?
DEFAULT_LANG (`'en'`)	The default language to use.
TRANSLATION_FEED ('feeds/all-%s.atom.xml'[3])	Where to put the RSS feed for translations.

5.2.6 Ordering contents

Setting name (default value)	what does it do?
REVERSE_ARCHIVE_ORDER (`False`)	Reverse the archives order. (True makes it in descending order: the newer first)
REVERSE_CATEGORY_ORDER (`False`)	Reverse the category order. (True makes it in descending order, default is alphabetically)

5.2.7 Theming

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

Setting name (default value)	what does it do?
THEME	theme to use to produce the output. can be the complete static path to a theme folder, or chosen between the list of default themes (see below)
THEME_STATIC_PATHS (`['static']`)	Static theme paths you want to copy. Default values is *static*, but if your theme has other static paths, you can put them here.
CSS_FILE (`'main.css'`)	specify the CSS file you want to load

By default, two themes are availablee. You can specify them using the *-t* option:

- notmyidea
- simple (a synonym for "full text" :)

You can define your own theme too, and specify it's emplacement in the same way (be sure to specify the full absolute path to it).

Here is a guide on how to create your theme

You can find a list of themes at http://github.com/ametaireau/pelican-themes.

Pelican comes with pelican-themes a small script for managing themes.

The *notmyidea* theme can make good use of the following settings. I recommend to use them too in your themes.

[3] %s is the language

Setting name	what does it do ?
DISQUS_SITENAME	Pelican can handle disqus comments, specify the sitename you've filled in on disqus
GITHUB_URL	Your github URL (if you have one), it will then use it to create a github ribbon.
GOOGLE_ANALYTICS	'UA-XXXX-YYYY' to activate google analytics.
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 to add a button on the articles to tweet about them. Add you twitter username if you want this button to appear.

In addition, you can use the "wide" version of the *notmyidea* theme, by adding that in your configuration:

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

5.2.8 Example settings

```
# -*- coding: utf-8 -*-
AUTHOR = u'Alexis Métaireau'
SITENAME = u"Alexis' log"
SITEURL = 'http://blog.notmyidea.org'
TIMEZONE = "Europe/Paris"

GITHUB_URL = 'http://github.com/ametaireau/'
DISQUS_SITENAME = "blog-notmyidea"
PDF_GENERATOR = False
REVERSE_CATEGORY_ORDER = True
LOCALE = ""
DEFAULT_PAGINATION = 2

FEED_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/"),
         (u'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'),)

# static paths will be copied under the same name
STATIC_PATHS = ["pictures",]
```

```
# A list of files to copy from the source to the destination
FILES_TO_COPY = (('extra/robots.txt', 'robots.txt'),)

# 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 it's HTML output. The jinja2 syntax is really simple. If you want to create your own theme, feel free to take inspiration from the "simple" theme, which is available here

5.3.1 Structure

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

```
-- static
|   -- css
|   -- images
-- templates
    -- archives.html    // to display 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 content. It will be copied on the output *theme/static* folder then. I've put the css and image folders, but they are just examples. Put what you need here.
- *templates* contains all the templates that will be used to generate the content. I've just put the mandatory templates here, you can define your own if it helps you to organize yourself while doing the theme.

5.3.2 Templates and variables

It's using a simple syntax, that you can embbed into your html pages. This document describes which templates should exist on a theme, and which variables will be passed to each template, while generating it.

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

Common variables

All of those settings will be given to all templates.

Variable	Description
articles	That's the list of articles, ordered desc. by date all the elements are *Article* objects, so you can access their properties (e.g. title, summary, author etc.).
dates	The same list of article, but ordered by date, ascending.
tags	A dict containing each tags (keys), and the list of relative articles.
categories	A dict containing each category (keys), and the list of relative articles.
pages	The list of pages.

index.html

Home page of your blog, will finally remain at output/index.html.

If pagination is active, next pages will remain at output/index'n'.html.

Variable	Description
articles_paginator	A paginator object of article list.
articles_page	The current page of articles.
dates_paginator	A paginator object of 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, and will finally remain at output/author/*author_name*.html.

If pagination is active, next pages will remain at output/author/*author_name''n*.html.

Variable	Description
author	The name of the author being processed.
articles	Articles of this author.
dates	Articles of this author, but ordered by date, ascending.
articles_paginator	A paginator object of article list.
articles_page	The current page of articles.
dates_paginator	A paginator object of article list, ordered by date, ascending.
dates_page	The current page of articles, ordered by date, ascending.
page_name	'author/*author_name*'. Useful for pagination links.

category.html

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

If pagination is active, next pages will remain at output/category/*category_name''n*.html.

Variable	Description
category	The name of the category being processed.
articles	Articles of this category.
dates	Articles of this category, but ordered by date, ascending.
articles_paginator	A paginator object of article list.
articles_page	The current page of articles.
dates_paginator	A paginator object of article list, ordered by date, ascending.
dates_page	The current page of articles, ordered by date, ascending.
page_name	'category/*category_name*'. Useful for pagination links.

article.html

This template will be processed for each article. .html files will be output in 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 of the current article.

page.html

For each page, this template will be processed. It will create .html files in output/*page_name*.html.

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

tag.html

For each tag, this template will be processed. It will create .html files in output/tag/*tag_name*.html.

If pagination is active, next pages will remain at 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 of article list.
articles_page	The current page of articles.
dates_paginator	A paginator object of article list, ordered by date, ascending.
dates_page	The current page of articles, ordered by date, ascending.
page_name	'tag/*tag_name*'. Useful for pagination links.

5.3.3 Inheritance

Since version 3, pelican supports inheritance from the `simple` theme, so you can reuse the templates of the `simple` theme 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 of the `simple` theme is right for you, you don't have to rewrite it from scratch.

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

```
{% extends "!simple/index.html" %}   <!-- extends the ``index.html`` template of the ``simple`` theme

{% extends "index.html" %}   <!-- "regular" extending -->
```

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:

```
{% extends "!simple/base.html" %}

{% block head %}
{{ super() }}
   <link rel="stylesheet" type="text/css" href="{{ SITEURL }}/theme/css/style.css" />
{% endblock %}
```

1. On the first line, we extend the `base.html` template of 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:

```
body {
    font-family : monospace ;
    font-size : 100% ;
    background-color : white ;
    color : #111 ;
    width : 80% ;
    min-width : 400px ;
    min-height : 200px ;
    padding : 1em ;
    margin : 5% 10% ;
    border : thin solid gray ;
    border-radius : 5px ;
    display : block ;
}

a:link    { color : blue ; text-decoration : none ;      }
a:hover   { color : blue ; text-decoration : underline ; }
a:visited { color : blue ;                               }

h1 a { color : inherit !important }
```

```
h2 a { color : inherit !important }
h3 a { color : inherit !important }
h4 a { color : inherit !important }
h5 a { color : inherit !important }
h6 a { color : inherit !important }

pre {
    margin : 2em 1em 2em 4em ;
}

#menu li {
    display : inline ;
}

#post-list {
    margin-bottom : 1em ;
    margin-top : 1em ;
}
```

Download

You can download this example theme `here`.

5.4 Pelican internals

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

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

5.4.1 Overall structure

What *pelican* does, is taking a list of files, and processing them, to some sort of output. Usually, the files are restructured text and markdown files, and the output is a blog, but it can be anything you want.

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

- *writers* are responsible of all the writing process of the files. It's 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 Restructured Text for now, but the system is extensible). Given a file, they return metadata (author, tags, category etc) and content (HTML formated)
- *generators* generate the different outputs. For instance, pelican comes with *ArticlesGenerator* and *PageGenerator*, into others. Given a configurations, 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 you own theme. The syntax is *jinja2*, and, trust me, really easy to learn, so don't hesitate a second.

5.4.2 How to implement a new reader ?

There is an awesome markup language you want to add to pelican ? Well, the only thing you have to do is to create a class that have a *read* method, that is returning an HTML content and some metadata.

Take a look to the Markdown reader:

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

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

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

Simple isn't it ?

If your new reader requires additional Python dependencies then you should wrap their *import* statements in *try...except*. 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 all the additional formats they don't use.

5.4.3 How to implement a new generator ?

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

- *generate_context*, that is called in a first place, 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 find all the pages, transform them into objects, and populate the context with them. Be careful to *not* output anything using this context at this stage, as it is likely to change by the effect of others generators.
- *generate_output* is then called. And guess what is it made for ? Oh, generating the output :) That's here that you may want to look at the context and call the methods of the *writer* object, that is passed at 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.

5.5 pelican-themes

5.5.1 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

5.5.2 Examples

Listing the installed themes

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

```
$ pelican-themes -l
notmyidea
two-column@
simple
$ pelican-themes --list
notmyidea
two-column@
simple
```

In this example, we can see there is 3 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 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 a more verbose output, like this:

```
$ pelican-themes -v -l
/usr/local/lib/python2.6/dist-packages/pelican-2.6.0-py2.6.egg/pelican/themes/notmyidea
/usr/local/lib/python2.6/dist-packages/pelican-2.6.0-py2.6.egg/pelican/themes/two-column (symbolic li
/usr/local/lib/python2.6/dist-packages/pelican-2.6.0-py2.6.egg/pelican/themes/simple
```

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:

```
# pelican-themes --install ~/Dev/Python/pelican-themes/notmyidea-cms --verbose
```

```
# pelican-themes --install ~/Dev/Python/pelican-themes/notmyidea-cms\
                           ~/Dev/Python/pelican-themes/martyalchin \
                           --verbose
```

```
# pelican-themes -vi ~/Dev/Python/pelican-themes/two-column
```

Removing themes

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

```
# pelican-themes --remove two-column
```

```
# pelican-themes -r martyachin notmyidea-cmd -v
```

Creating symbolic links

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

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

```
# pelican-themes --symlink ~/Dev/Python/pelican-themes/two-column
```

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:

```
$ sudo pelican-themes -s ~/Dev/Python/pelican-themes/two-column
$ pelican ~/Blog/content -o /tmp/out -t two-column
$ firefox /tmp/out/index.html
$ vim ~/Dev/Pelican/pelican-themes/two-coumn/static/css/main.css
$ pelican ~/Blog/content -o /tmp/out -t two-column
$ cp /tmp/bg.png ~/Dev/Pelican/pelican-themes/two-coumn/static/img/bg.png
$ pelican ~/Blog/content -o /tmp/out -t two-column
$ vim ~/Dev/Pelican/pelican-themes/two-coumn/templates/index.html
$ pelican ~/Blog/content -o /tmp/out -t two-column
```

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:

```
# pelican-themes --remove notmyidea-cms two-column \
                 --install ~/Dev/Python/pelican-themes/notmyidea-cms-fr \
                 --symlink ~/Dev/Python/pelican-themes/two-column \
                 --verbose
```

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

5.5.3 See also

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

5.6 Import from other blog software

5.6.1 Description

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

- Wordpress XML export
- Dotclear export
- RSS/ATOM feed

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

Usage

pelican-import [-h] [–wpfile] [–dotclear] [–feed] [-o OUTPUT]
[–dir-cat]
input

Optional arguments:

-h, --help	show this help message and exit
--wpfile	Wordpress XML export
--dotclear	Dotclear export
--feed	Feed to parse
-o OUTPUT, --output OUTPUT	Output path
--dir-cat	Put files in directories with categories name

5.6.2 Examples

for Wordpress:

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

for Dotclear:

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

5.6.3 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

5.7 Frequently Asked Questions (FAQ)

Here is a summary of the frequently asked questions for pelican.

5.7.1 Is it mandatory to have a configuration file ?

No, it's not. Configurations files are just an easy way to configure pelican. For the basic operations, it's possible to specify options while invoking pelican with the command line (see *pelican –help* for more informations about that)

5.7.2 I'm creating my own theme, how to use pygments ?

Pygment add some classes to the generated content, so the theming of your theme will be done thanks to a css file. You can have a look to the one proposed by default on the project website

5.7.3 How do I create my own theme ?

Please refer yourself to *How to create themes for pelican*.

5.7.4 How can I help ?

You have different options to help. First, you can use pelican, and report any idea or problem you have on the bugtracker.

If you want to contribute, please have a look to the git repository, fork it, add your changes and do a pull request, I'll review them as soon as possible.

You can also contribute by creating themes, and making the documentation better.

5.7.5 I want to use markdown, but I got an error

Markdown is not a hard dependency for pelican, so you will need to install it by yourself. You can do so by typing:

```
$ (sudo) pip install markdown
```

In case you don't have pip installed, consider installing it by doing:

```
$ (sudo) easy_install pip
```

5.8 Tips

Here are some tips about pelican, which you might find useful.

5.8.1 Publishing to github

Github comes with an interesting "pages" feature: you can upload things there and it will be available directly from their servers. As pelican is a static file generator, we can take advantage of this.

The excellent ghp-import makes this eally easy. You would have to install it:

```
$ pip install ghp-import
```

Then, considering a repository containing your articles, you would simply have to run pelican and upload the output to github:

```
$ pelican -s pelican.conf.py .
$ ghp-import output
$ git push origin gh-pages
```

And that's it.

If you want you can put that directly into a post commit hook, so each time you commit, your blog is up to date on github!

Put the following into *.git/hooks/post-commit*:

```
pelican -s pelican.conf.py . && ghp-import output && git push origin
gh-pages
```

5.9 How to contribute ?

There are many ways to contribute to pelican. You can enhance the documentation, add missing features, fix bugs or just report them.

Don't hesitate to fork and make a pull request on github.

5.9.1 Set up the development environment

You're free to setup up the environment in any way you like. Here is a way using virtualenv and virtualenvwrapper. If you don't have them, you can install them using:

```
$ pip install virtualenvwrapper
```

Virtual environments allow you to work on an installation of python which is not the one installed on your system. Especially, it will install the different projects under a different location.

To create the virtualenv environment, you have to do:

```
$ mkvirtualenv pelican --no-site-package
```

Then you would have to install all the dependencies:

```
$ pip install -r dev_requirements.txt
$ python setup.py develop
```

5.9.2 Running the test suite

Each time you add a feature, there are two things to do regarding tests: checking that the tests run in a right way, and be sure that you add tests for the feature you are working on or the bug you're fixing.

The tests leaves under "pelican/tests" and you can run them using the "discover" feature of unittest2:

```
$ unit2 discover
```

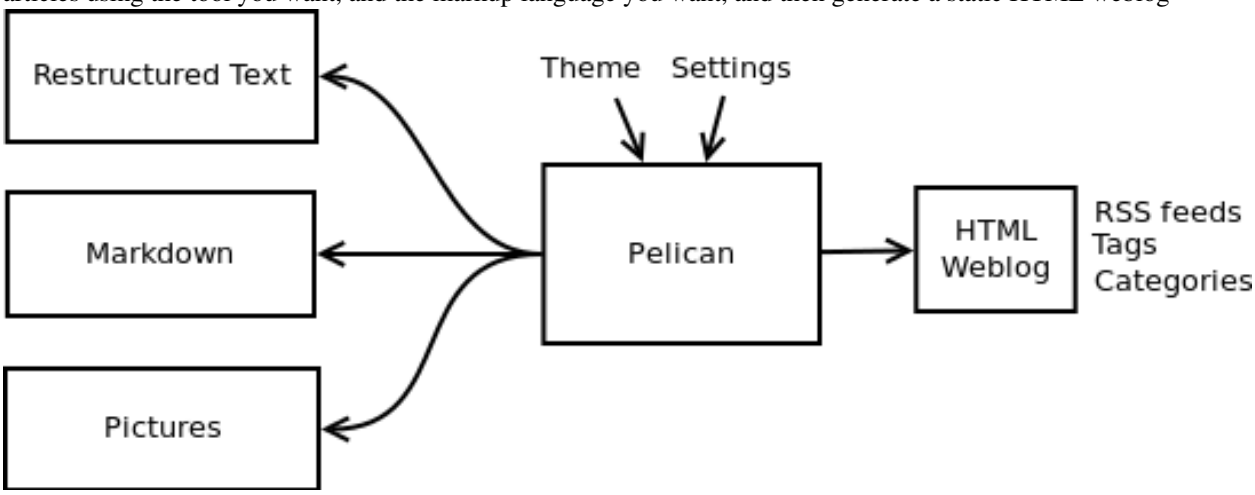
5.10 Some history about pelican

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 restructured text for now), and generate a 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.

5.10.1 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 have a template support, so you can easily write you own themes if you want to.

5.10.2 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 have created the content I wanted to parse (the restructured text 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.

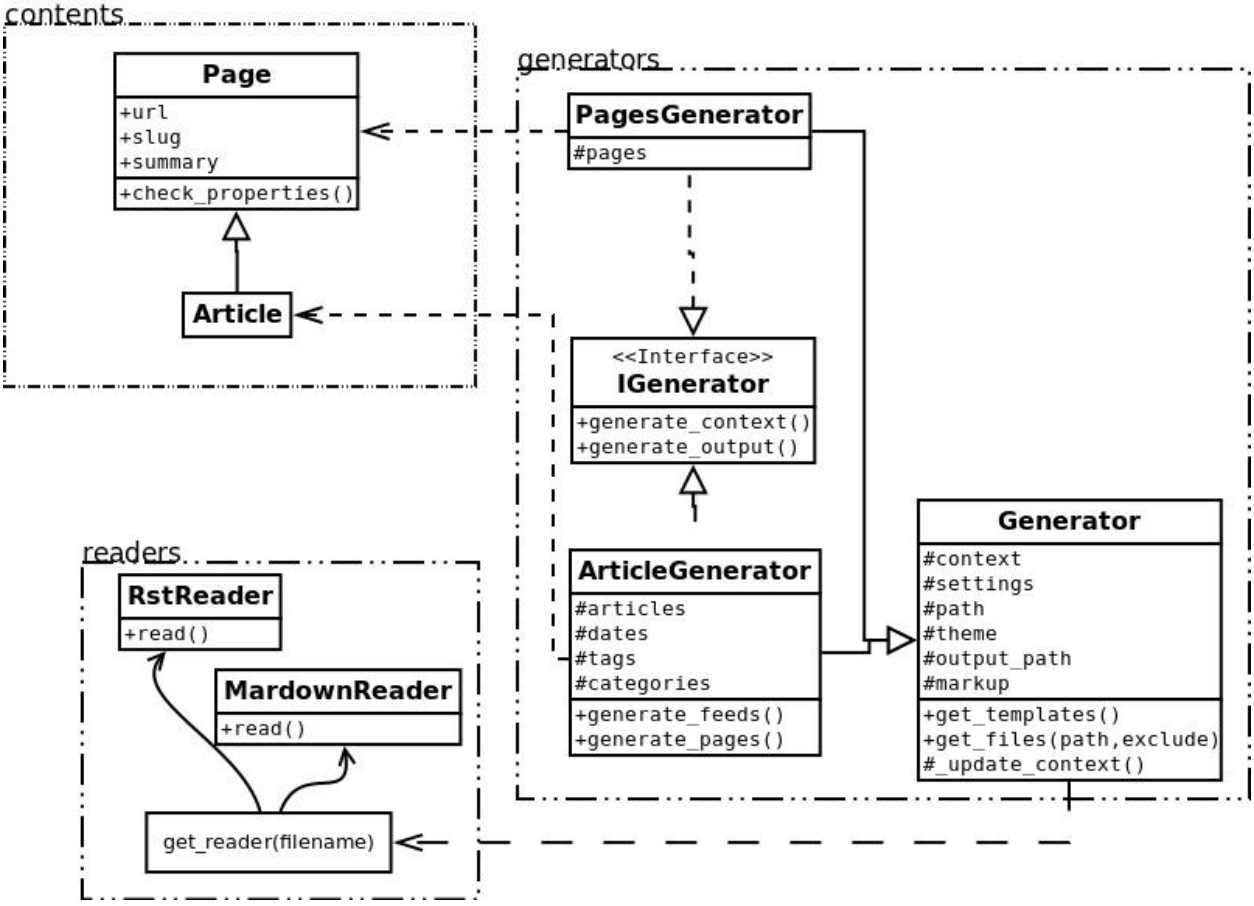
I've separated the logic in different classes and concepts:

- *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 Restructured Text for now, but the system is extensible). Given a file, they return metadata (author, tags, category etc) and content (HTML formated).
- *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.

5.10.3 In more details

Here is an overview of the classes involved in pelican.



The interface do not really exists, 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 are 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