Plim Documentation

0.9.11

Maxim Avanov

2015 06 22

Contents

1 Installation **3**

2 Tests **5**

3 Detailed example **7**

4 Main Documentation **9**
- 4.1 Syntax . . . 9
- 4.2 Syntactic Differences . . . 24
- 4.3 Extensions . . . 26
- 4.4 Framework Integration . . . 31
- 4.5 Command-line Interface . . . 32
- 4.6 License . . . 32
- 4.7 Authors . . . 32
- 4.8 Contributors (in chronological order of the first contribution) . . . 32
- 4.9 Related projects . . . 33
- 4.10 Changelog . . . 33

5 Indices and tables **37**

Python **39**

Plim is a Python port of Ruby's Slim template language built on top of Mako Templates. It uses Mako's preprocessor feature to translate its syntax into a valid HTML/Mako markup.

CHAPTER 1

Installation

```
pip install Plim
```

CHAPTER 2

Tests

Plim provides an extensive test suite based on nosetests. You can run the tests with the following command

```
python setup.py nosetests
```

Coverage statistics are available online.

Detailed example

```
/ base.html
--------------------------
doctype html
html = next.body()
```

```
/ helpers.html
-------------------------
-def other_headers()
    meta charset="utf-8"
    link rel="stylesheet" href="/static/css/main.css"
```

```
/ layout.html
--------------------------
-inherit base.html
-namespace name="helper" helpers.html

head
  title Plim Example
  meta name="keywords" content="template language"
  = helper.other_headers()

  script
    /* "script" and "style" blocks do not require explicit literal indicator "|"  */
    $(content).do_something();

  style
    body {
      background:#FFF;
      }

  -scss
    /* SCSS/SASS extension */
    @option compress: no;
    .selector {
      a {
        display: block;
      }
      strong {
        color: blue;
      }
    }

  -coffee
```

```
    # CoffeeScript extension
    square = (x) -> x * x

body
  h1 Markup examples
  #content.example1
    p Nest by indentation
    <div>
      p Mix raw HTML and Plim markup
    </div>

    -md
      Use Markdown
      ============

      See the syntax on [this page][1].

      [1]: http://daringfireball.net/projects/markdown/basics

    -rest
      or Use reStructuredText
      =======================

      See the syntax on `this page`_.

      .. _this page: http://docutils.sourceforge.net/docs/user/rst/quickref.html


  -if items
    table: -for item in items: tr
      td = item.name
      td = item.price
  -elif show_empty
    p No items found
  -else
    a href=request.route_url('items.add') =, _('Add items')

  -unless user.authenticated
    p Please, sign in.
  -else
    p Welcome, ${user.name}!
    ul
      --- i = 0
          limit = 5

      -while i < limit
        li#idx-${i}.up: a href='#' title="Title" == i
        --- i += 1

      -until i < 0
        li#idx-${i}.down-${i}: a href='''#''' title="""Title""" ==, i
        --- i -= 1

  #footer
    Copyright &copy; 2014.
    -include footer_links.html

= render('tracking_code')
```

CHAPTER 4

Main Documentation

4.1 Syntax

Some relevant parts of this section were copied from the official Slim documentation.

4.1.1 Line Indicators

| The pipe tells Plim to just copy the line. It essentially escapes any processing.

, The comma tells Plim to copy the line (similar to `|`), but makes sure that a single trailing space is appended.

> **:** Slim syntax instead has the single quote (`'`) line indicator.

- The dash denotes control code. Examples of control code are loops, conditionals, mako tags, and extensions.

= The equal sign tells Plim it's a Python call that produces output to add to the buffer.

=, Same as the single equal sign (=), except that it adds an explicit single trailing whitespace after the python expression.

== Same as the single equal sign (=), but adds the "n" filter to mako expression.

```
    == python_expression                        =>      ${python_expression|n}

    == python_expression | custom_filter        =>      ${python_expression |n,custom_filter}
```

==, Same as the double equal sign (==), except that it adds an explicit single trailing whitespace after the python expression.

/ Use the forward slash for code comments - anything after it won't get displayed in the final mako markup.

> **:** **There is no an equivalent syntax for Slim's "/!" html comment**. This line indicator has been considered redundant since Plim supports raw HTML tags:

```
    / You can use raw HTML comment tags, as well as all other tags
    <!-- HTML comment -->
    <div>

      / You can use Plim markup inside the raw HTML
      a href="#" = link.title

    / If you use raw HTML, you have to close all tags manually
    </div>
```

4.1.2 Indentation

Plim indentation rules are the same as of Slim: indentation matters, but it's not as strict as Haml. If you want to first indent 2 spaces, then 5 spaces, it's your choice. To nest markup you only need to indent by one space, the rest is gravy.

4.1.3 Tag Attributes

Static Attrbutes

Static tag attributes can be specified in the same form as any valid python string declaration.

For example

```
input type='text' name="username" value='''Max Power''' maxlength="""32"""
```

will be rendered as

```
<input type="text" name="username" value="Max Power" maxlength="32"/>
```

As in Python string declarations, you must take care of correct quote escaping

```
input value='It\'s simple'
input value="It's simple"
input value='''It's simple'''
input value="""It's simple"""

input value="He said \"All right!\""
input value='He said "All right!"'
input value='''He said "All right!"'''
input value="""He said "All right!\""""
```

You can omit quotes from numeric attribute values:

```
input type='text' name="measure" value=+.97 maxlength=32
```

produces

```
<input type="text" name="measure" value="+.97" maxlength="32"/>
```

Dynamic Attributes

Dynamic values can be specified in forms of:

- Mako expression

```
input type="text" name="username" value="${user.name}" maxlength=32
a href="${request.route_url('user_profile', tagname=user.login, _scheme='https')}"
```

or

```
input type="text" name="username" value=${user.name} maxlength=32
a href=${request.route_url('user_profile', tagname=user.login, _scheme='https')}
```

- Python expression

```
    input type="text" name="username" value=user.name maxlength=32
    a href=request.route_url('user_profile', tagname=user.login, _scheme='https')
```

or with parentheses

```
    input type="text" name="username" value=(user.name) maxlength=32
    a href=(request.route_url('user_profile', tagname=user.login, _scheme='https'))
```

All these examples produce the same mako markup:

```
<input type="text" name="username" value="${user.name}" maxlength="32"/>
<a href="${request.route_url('user_profile', tagname=user.login, _scheme='https')}"></a>
```

Boolean Attributes

Boolean attributes can be specified either by static or dynamic method:

Static attribute example:

```
/ Static boolean attribute "disabled"
input type="text" name="username" disabled="disabled"

/ If you wrap your attributes with parentheses, you can use
  shortcut form
input (type="text" name="username" disabled)
```

Dynamic attribute example (note the trailing question mark):

```
/ Dynamic boolean attribute "disabled"
  will be evaluated to 'disabled="disabled"' if `is_disabled`
  evaluates to True

input type="text" name="username" disabled=${is_disabled}?

/ or you can write it that way
input type="text" name="username" disabled=is_disabled?

/ or even that way
input type="text" name="username" disabled=(is_disabled or
                                            is_really_disabled)?
```

Dynamic unpacking

This feature is an equivalent to Slim's splat attributes, but the syntax was changed in order to correspond to Python's `**kwargs` semantics.

Consider the following python dictionary:

```
attrs = {
    'id': 'navbar-1',
    'class': 'navbar',
    'href': '#',
    'data-context': 'same-frame',
}
```

Now we can unpack the dictionary in order to populate tags with attributes. The following line:

```
a**attrs Link
```

will be translated to mako template which will output an equivalent to the following HTML markup

```
<a id="navbar-1" class="navbar" href="#" data-context="same-frame">Link</a>
```

Here are some other examples

```
a **attrs|Link

a **attrs **more_attrs Link

a(**attrs disabled) Disabled Link

a **function_returning_dict(
  *args, **kwargs
) Link
```

4.1.4 Attribute Wrapping

You can wrap tag attributes with parentheses `()`. Unlike Slim, Plim doesn't support square `[]` or curly `{}` braces for attributes wrapping.

```
body
  h1(id="logo" class="small tagline") = page_logo
  h2 id=(id_from_variable + '-idx') = page_tagline
```

If you wrap the attributes, you can spread them across multiple lines:

```
body

  h1 (id="logo"
    class="small tagline") = page_logo

  h2 id=(
    id_from_variable +
    '-idx'
  ) = page_tagline
```

4.1.5 Inline Tag Content

You can put content on the same line with the tag:

```
body
  h1 id="headline" Welcome to my site.
```

Or nest it. Note: use either a pipe or *Implicit literal indicators*

```
body
  h1 id="headline"

    / Explicit literal with pipe character
    | Welcome to my site.

    / Implicit literal (uppercase letter at the beginning of the line)
    Yes, Welcome to my site
```

4.1.6 Dynamic Tag Content

You can make the call on the same line

```
body
  h1 id="headline" = page_headline
```

Or nest it.

```
body
  h1 id="headline"
    = page_headline
```

4.1.7 `id` and `class` Shortcuts

You can specify the id and class attributes in the following shortcut form.

```
body

  / Static shortcuts
  h1#headline
    = page_headline
  h2#tagline.small.tagline
    = page_tagline
  .content
    = show_content
```

This is the same as:

```
body
  h1 id="headline"
    = page_headline
  h2 id="tagline" class="small tagline"
    = page_tagline
  div class="content"
    = show_content
```

In contrast to Slim, Plim allows you to insert dynamic expressions right into the shortcuts:

```
/ Dynamic shortcuts
h1#headline-${'dynamic'} = page_headline
h2#${tagline.id}.small-${tagline.cls}.${tagline.other_cls}
  = page_tagline
.${'content'}
  = show_content
```

This is the same as:

```
h1 id="headline-${'dynamic'}" = page_headline
h2 id="${tagline.id}" class="small-${tagline.cls} ${tagline.other_cls}"
  = page_tagline
div class="${'content'}"
  = show_content
```

4.1.8 Inline Tags

Sometimes you may want to be a little more compact and inline the tags.

```
ul
  li.first: a href="/a" A link
  li: a href="/b" B link
```

For readability, don't forget you can wrap the attributes.

```
ul
  li.first: a(href="/a") A link
  li: a(href="/b") B link
```

4.1.9 Inline Statements

You can inline python loops and conditional expressions in the same manner as tags.

```
ul: -for link in ['About', 'Blog', 'Sitemap']: li: a href=route_to(link) = link
```

will be rendered as

```
<ul>
%for link in ['About', 'Blog', 'Sitemap']:
<li><a href="${route_to(link)}">${link}</a></li>
%endfor
</ul>
```

4.1.10 Evaluate Python Code in Text

Use standard mako expression syntax. The text escaping depends on mako's default filters settings.

```
body
  h1 Welcome ${current_user.name} to the show.
  Explicit non-escaped ${content|n} is also possible.
```

Currently, Mako doesn't provide a simple way to escape the interpolation of expressions (i.e. render as is). You can use either the <%text> tag (or Plim's `-text` equivalent for blocks of mako syntax examples), or this trick

```
body
  h1 Welcome ${'${current_user.name}'} to the show.
```

4.1.11 Embedded Markup

You can embed plim markup right into literal blocks:

```
a href="#" Embedded `strong string` everywhere
```

is rendered as

```
<a href="#">Embedded <strong>string</strong> everywhere</a>
```

If you want to put two embedded strings next to each other, add a trailing underscore character after the first embedded string:

```
a href="#" Embedded `strong string`_`i s` everywhere
```

is rendered as

```
<a href="#">Embedded <strong>string</strong><i>s</i> everywhere</a>
```

The embedding mechanism is recursive so you can embed plim markup into embedded plim markup:

```
Another `a href="#" very ``strong funny ````i recursive``````` test
```

is rendered as

```
Another <a href="#">very <strong>funny <i>recursive</i></strong></a> test
```

4.1.12 Skip HTML Escaping

Use either a double equal sign

```
body
  h1 id="headline"
    == page_headline
```

or the explicit `| n` filter at the end of the expression

```
body
  h1 id="headline"
    = page_headline | n
```

4.1.13 Code Comments

Use forward slash `/` for code comments

```
body
  p
    / This is a comment.
      Indentation is the natural way to get block comments
```

4.1.14 Raw HTML Tags

Plim allows you to use raw HTML tag lines, and also to mix them with any available control logic. It is particularly useful in situations like this one:

```
- if edit_profile
  / Wrap interface with editable block
  <div id="edit-profile">

- include new_or_edit_interface.html

- if edit_profile
  / close wrapper tag
  </div>
```

4.1.15 Doctype Declarations

There is no default option for doctype declaration tag. Therefore, you should explicitly specify the desired doctype:

```
doctype 5
```

Here is the full list of available doctypes:

doctype html

```
<!DOCTYPE html>
```

doctype 5

```
<!DOCTYPE html>
```

doctype 1.1

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
```

doctype strict

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

doctype xml

```
<?xml version="1.0" encoding="utf-8" ?>
```

doctype transitional

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

doctype frameset

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

doctype basic

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.1//EN"
"http://www.w3.org/TR/xhtml-basic/xhtml-basic11.dtd">
```

doctype mobile

```
<!DOCTYPE html PUBLIC "-//WAPFORUM//DTD XHTML Mobile 1.2//EN"
"http://www.openmobilealliance.org/tech/DTD/xhtml-mobile12.dtd">
```

4.1.16 Control Logic

if/elif/else statements

```
-if items
  table
    -for item in items
      tr
        td = item.name
        td = item.price
-elif show_empty
  p No items found
-else
  a href=request.route_url('items.add') =, _('Add items')
```

unless statement

This is the shortcut form of the `- if not (<EXPR>)` statement.

```
-unless user.authenticated
  p Please, sign in.
-else
  p Welcome, ${user.name}!
```

for statement

```
table
  -for item in items
    tr
      td = item.name
      td = item.price
```

You can use the `-continue` and `-break` commands inside the `-for` block. See the section *Returning Early from a Template*.

while statement

```
-python
  i = 0
  limit = 5

ul
  -while i < limit
    li#idx-${i}.up: a href='#' title="Title" == i
    -py i += 1
```

You can use the `-continue` and `-break` commands inside the `-while` block. See the section *Returning Early from a Template*.

until statement

This is the shortcut form of the `- while not (<EXPR>)` statement.

```
-until i < 0
  li#idx-${i}.down-${i}: a href='''#''' title="""Title""" ==, i
  -py i -= 1
```

You can use the `-continue` and `-break` commands inside the `-until` block. See the section *Returning Early from a Template*.

with statement

The `% with` statement was introduced in Mako 0.7.0.

```
-with <EXPR> as <VAR>
  / Do some stuff
```

try/except statements

```
- try
  div = item[0]
  div = price['usd']

- except IndexError
  div IndexError

- except KeyError as e
  div = e
```

Returning Early from a Template

Plim provides a shortcut to Mako's <% return %> template directive.

```
- if not len(records)
  No records found.
  -return
```

This is the same as

```
- if not len(records)
  No records found.
  -py return
```

You can use it at any position of the template, not only inside control structures.

There are also the `-break` and `-continue` shortcuts, that can be used inside the `-for`, `-while`, and `-until` loops.

4.1.17 Literals

You can specify either explicit or implicit literal blocks. The difference is, whether you prepend a block with the explicit pipe char `|` or start it by one of the implicit indicators.

Explicit Literals

Use a pipe (`|`) or comma (`,`) literal indicators to start the escape. Each following line that is indented greater than the first one is copied over.

```
body
  p
    / Explicit literal
    | This is a test of the text block.
```

```
body
  p
    |
      This is a test of the text block.
```

The parsed result of both the above examples:

```
<body><p>This is a test of the text block.</p></body>
```

The left margin is set to the zero. Any additional spaces will be copied over.

```
body
  p
    |  This line is on the zero left margin.
        This line will have one space in front of it.
          This line will have two spaces in front of it.
            And so on...
```

Implicit Literals

Literal blocks can be implicitly specified by the following starting sequences:

- an uppercase ASCII letter;
- any non-ASCII letter;
- a digit without prefixed positive/negative signs;
- HTML-escaped character, for example ` `;
- Mako open brace sequence `${`;
- an open square brace `[`;
- an open parenthesis `(`;
- any unicode character outside the range of U0021 - U007E (ASCII 33 - 126).

```
p
  | pipe is the explicit literal indicator. It is required if your line starts with
    the non-literal character.

p
  I'm the implicit literal, because my first letter is in uppercase.

p
  1. Digits
  2. are
  3. the
  4. implicit
  5. literals
  6. too.

p
  ${raw_mako_expression} indicates the implicit literal line.

p
  If subsequent lines do not start with implicit literal indicator,
    you must indent them
  | or you can use the "explicit" pipe.
```

```
p
  / if your literal blocks are written in Russian, or any other
    language which uses non-ASCII letters, you can put even the
    lowercase letter at the beginning of the block

    Unfortunately, we cannot provide an example of this feature here,
    because current version of Sphinx Documenting tool cannot automatically
    build PDF documentation with unicode characters.
    See an explanation on
    https://groups.google.com/forum/#!topic/sphinx-dev/kUeROyCyX9w/discussion
```

4.1.18 Python Blocks

Classic Blocks

Use `-py` or `-python` to insert the <% %> mako tag.

For example

```
- python x = 1
```

or

```
-py
    x = 1
```

or even

```
- python x = 1
    y = x + 1
    if True:
        y += 1
    else:
        y -= 1
```

In latter case, the first line `x = 1` will be aligned with the second line `y = x + 1`.

New-style blocks

0.9.1 : New-style blocks were introduced for better readability of embedded python code.

Use a sequence of at least three dashes to start a new-style python block.

Here are the overwritten examples of the classic ones. The results are the same:

```
--- x = 1
```

```
-------------
    x = 1
```

```
--- x = 1
    y = x + 1
    if True:
        y += 1
    else:
        y -= 1
```

And here's an example of how we can use an inline statement for providing a block description

```
ul#userlist
    ---------- # prepare a list of users ---------------
        users = UsersService.get_many(max=100, offset=0)
        friends = UsersService.get_friends_for(users)
    ------------------------------------------------------
    -for user in users: li
        h4: a#user-${user.id} href='#' = user.name
        ul: -for friend in friends[user.id]: li
            a#friend-${friend.id} href='#' = friend.name
```

the result (indentation will be stripped out):

```
<ul id="userlist">
    <%
        # prepare a list of users
        users = UsersService.get_many(max=100, offset=0)
        friends = UsersService.get_friends_for(users)
    %>

    %for user in users:
        <li>
            <h4>
                <a href="#" id="user-${user.id}">${user.name}</a>
            </h4>
            <ul>
                %for friend in friends[user.id]:
                    <li>
                        <a href="#" id="friend-${friend.id}">${friend.name}</a>
                    </li>
                %endfor
            </ul>
        </li>
    %endfor
</ul>
```

4.1.19 Module-level Blocks

Use `-py!` or `-python!` block to insert the <%! %> mako tag.

```
-py!
    import mylib
    import re

    def filter(text):
        return re.sub(r'^@', '', text)
```

0.9.1 : The same example with a new-style syntax:

```
---! import mylib
     import re

     def filter(text):
         return re.sub(r'^@', '', text)
```

4.1.20 Mako Tags

Plim supports a complete set of Mako Tags, except the `<%doc>`. The latter has been considered deprecated, since Plim itself has built-in support of multi-line comments.

: As all mako tags start with the < char, which indicates a *raw HTML line*, they all can be written "as is". The only thing you must remember is to manually close the pair tags.

-page tag

```
-page args="x, y, z='default'"
```

produces

```
<%page args="x, y, z='default'"/>
```

See the details of what <%page> is used for in The body() Method and Caching sections of Mako Documentation.

-include tag

```
-include footer.html
```

or

```
-include file="footer.html"
```

produce the same output

```
<%include file="footer.html"/>
```

See the <%include> section of Mako Documentation to get more information about this tag.

-inherit tag

```
-inherit base.html
```

or

```
-inherit file="base.html"
```

will generate the same

```
<%inherit file="base.html"/>
```

See Mako's inheritance documentation to get more information about template inheritance.

-namespace tag

```
-namespace name="helper" helpers.html
```

or

```
-namespace file="helpers.html" name="helper"
```

produce

```
<%namespace file="helpers.html" name="helper"/>
```

See Mako's namespace documentation to get more information about template namespaces.

-def tag

```
-def account(accountname, type='regular')
```

or

```
-def name="account(accountname, type='regular')"
```

produce

```
<%def name="account(accountname, type='regular')">
</%def>
```

See Mako's defs and blocks documentation to get more information about functions and blocks.

-block tag

Unlike `-def` statements, blocks can be anonymous

```
-block
  This is an anonymous block.
```

Or they can be named as functions

```
-block name="post_prose"
  = pageargs['post'].content
```

or

```
-block post_prose
  = pageargs['post'].content
```

As in `-def`, both above examples produce the same result

```
<%block name="post_prose">
${pageargs['post'].content}</%block>
```

You can also specify other block arguments as well

```
- block filter="h"
  html this is some escaped html.
```

See Mako's defs and blocks documentation to get more information about functions and blocks.

-call tag

This statement allows you to define custom tags.

The following examples

```
-call expression="${4==4}" self:conditional
  |i'm the result

- call expression=${4==4} self:conditional
  | i'm the result

- call self:conditional
  | i'm the result
```

```
- call self:conditional
```

will produce

```
<%self:conditional expression="${4==4}">
i'm the result
</%self:conditional>

<%self:conditional expression="${4==4}">
i'm the result
</%self:conditional>

<%self:conditional>
i'm the result
</%self:conditional>

<%self:conditional>
</%self:conditional>
```

Please consult the sections <%nsname:defname> and Calling a Def with Embedded Content and/or Other Defs of Mako Documentation to get more information about custom tags.

-text tag

As it is mentioned in Mako documentation, this tag suspends the Mako lexer's normal parsing of Mako template directives, and returns its entire body contents as plain text. It is used pretty much to write documentation about Mako:

```
-text filter="h"
  here's some fake mako ${syntax}
  <%def name="x()">${x}</%def>

- text filter="h" here's some fake mako ${syntax}
  <%def name="x()">${x}</%def>

- text filter="h" = syntax
  <%def name="x()">${x}</%def>

-text
  here's some fake mako ${syntax}
  <%def name="x()">${x}</%def>

-text , here's some fake mako ${syntax}
  <%def name="x()">${x}</%def>
```

4.2 Syntactic Differences

Plim is *not the exact* port of Slim. Here is the full list of differences.

1. Slim has the ('), (='), and (==') line indicators. In Plim, single quote has been replaced by the comma char (,):

```
    , value
    =, value
    ==, value
```

The change was made in order to get rid of the syntactic ambiguities like these:

```
/ Is this an empty python string or a syntax error caused by the unclosed single quote?
=''

/ Is this a python string 'u' ('u''' is the correct python syntax) or
  a syntax error caused by the unclosed unicode docstring?
='u'''
```

Meanwhile, the comma char is not allowed at the beginning of python expression, therefore the following code samples are consistent:

```
/ Syntax error at mako runtime caused by the unclosed single quote
=,'

/ Correct and consistent. Produces an empty unicode string followed by an
  explicit trailing whitespace
=,u''
```

In addition, the comma syntax seems more natural, since in formal writing we also add a whitespace between a comma and the following word (in contrast to apostrophes, which may be written together with some parts of words - "I'm", "it's" etc).

2. Unlike Slim, Plim does not support square or curly braces for wrapping tag attributes. You can use only parentheses `()`:

```
/ For attributes wrapping we can use only parentheses
p(title="Link Title")
  h1 class=(item.id == 1 and 'one' or 'unknown') Title

  / Square and curly braces are allowed only in Python and Mako expressions
  a#idx-${item.id} href=item.get_link(
                **{'argument': 'value'}) = item.attrs['title']
```

3. In Plim, all html tags **MUST** be written in lowercase.

   This restriction was introduced to support *Implicit Litaral Blocks* feature.

```
doctype 5
html
  head
    title Page Title
  body
    p
      | Hello, Explicit Literal Block!
    p
      Hello, Implicit Literal Block!
```

4. You do not have to use the `|` (pipe) indicator in `style` and `script` tags.
5. Plim does not make distinctions between control structures and embedded filters.

   For example, in Slim you would write `-if`, `-for`, and `coffee:` (without preceding dash, but with the colon sign at the tail). But in Plim, you must write `-if`, `-for`, and `-coffee`.

6. In contrast to Slim, Plim does not support the `/!` line indicator which is used as an HTML-comment. You can use raw HTML-comments instead.

4.3 Extensions

4.3.1 Standard extensions

CoffeeScript

Plim uses Python-CoffeeScript package as a bridge to the JS CoffeeScript compiler. You can start CoffeeScript block with the `-coffee` construct.

```
- coffee
  # Assignment:
  number   = 42
  opposite = true

  # Conditions:
  number = -42 if opposite

  # Functions:
  square = (x) -> x * x

  # Arrays:
  list = [1, 2, 3, 4, 5]

  # Objects:
  math =
    root:   Math.sqrt
    square: square
    cube:   (x) -> x * square x

  # Splats:
  race = (winner, runners...) ->
    print winner, runners

  # Existence:
  alert "I knew it!" if elvis?

  # Array comprehensions:
  cubes = (math.cube num for num in list)
```

SCSS/SASS

Plim uses pyScss package to translate SCSS/SASS markup to plain CSS. You can start SCSS/SASS block with the `-scss` or `-sass` construct. The output will be wrapped with `<style></style>` tags.

For example,

```
- scss
  @option compress: no;
  .selector {
    a {
      display: block;
    }
    strong {
      color: blue;
    }
  }
```

produces

```
<style>.selector a {
  display: block;
}
.selector strong {
  color: #00f;
}</style>
```

Stylus

Plim uses stylus package to translate stylus markup to plain CSS. You can start Stylus block with the `-stylus` construct. The output will be wrapped with `<style></style>` tags.

For example,

```
- stylus
  @import 'nib'
  body
    background: linear-gradient(top, white, black)

  border-radius()
    -webkit-border-radius arguments
    -moz-border-radius arguments
    border-radius arguments

  a.button
    border-radius 5px
```

produces

```
<style>body {
  background: -webkit-gradient(linear, left top, left bottom, color-stop(0, #fff), color-stop(1, #000
  background: -webkit-linear-gradient(top, #fff 0%, #000 100%);
  background: -moz-linear-gradient(top, #fff 0%, #000 100%);
  background: -o-linear-gradient(top, #fff 0%, #000 100%);
  background: -ms-linear-gradient(top, #fff 0%, #000 100%);
  background: linear-gradient(top, #fff 0%, #000 100%);
}
a.button {
  -webkit-border-radius: 5px;
  -moz-border-radius: 5px;
  border-radius: 5px;
}</style>
```

Markdown

Plim uses python-markdown2 package for the `-markdown` (or `-md`) extension.

For example,

```
- markdown
  A First Level Header
  ====================

  A Second Level Header
  ---------------------
```

```
  Now is the time for all good men to come to
  the aid of their country. This is just a
  regular paragraph.

  The quick brown fox jumped over the lazy
  dog's back.

  ### Header 3

  > This is a blockquote.
  >
  > This is the second paragraph in the blockquote.
  >
  > ## This is an H2 in a blockquote
```

produces

```
<h1>A First Level Header</h1>

<h2>A Second Level Header</h2>

<p>Now is the time for all good men to come to
the aid of their country. This is just a
regular paragraph.</p>

<p>The quick brown fox jumped over the lazy
dog's back.</p>

<h3>Header 3</h3>

<blockquote>
    <p>This is a blockquote.</p>

    <p>This is the second paragraph in the blockquote.</p>

    <h2>This is an H2 in a blockquote</h2>
</blockquote>
```

reStructuredText

Plim uses Docutils package for both supporting the `-rest` (or `-rst`) extension and project documenting.

For example,

```
- rest
  Grid table:

  +------------+------------+-----------+
  | Header 1   | Header 2   | Header 3  |
  +============+============+===========+
  | body row 1 | column 2   | column 3  |
  +------------+------------+-----------+
  | body row 2 | Cells may span columns.|
  +------------+------------+-----------+
  | body row 3 | Cells may  | - Cells   |
  +------------+ span rows. | - contain |
  | body row 4 |            | - blocks. |
```

```
  +------------+------------+------------+
```

produces

```
<p>Grid table:</p>
<table border="1">
  <thead valign="bottom">
    <tr>
      <th>Header 1
      </th><th>Header 2
      </th><th>Header 3
    </th></tr>
  </thead>
  <tbody valign="top">
    <tr>
      <td>body row 1
      </td><td>column 2
      </td><td>column 3
    </td></tr>
    <tr>
      <td>body row 2
      </td><td colspan="2">Cells may span columns.
    </td></tr>
    <tr>
      <td>body row 3
      </td><td rowspan="2">Cells may<br>span rows.
      </td><td rowspan="2">
        <ul>
          <li>Cells
          </li><li>contain
          </li><li>blocks.
        </li></ul>
    </td></tr>
    <tr>
      <td>body row 4
    </td></tr>
</tbody></table>
```

Handlebars

Plim supports a special tag `handlebars` that is translated to a handlebars section declaration:

```
<script type="text/x-handlebars"></script>
```

This is particularly useful to developers using Ember.js.

Here is an example. The following plim document

```
html
    body
        handlebars#testapp
            .container {{outlet}}

        handlebars#about: .container {{outlet}}
```

will be rendered as

```
<html>
    <body>
        <script type="text/x-handlebars" id="testapp">
            <div class="container">{{outlet}}</div>
        </script>
        <script type="text/x-handlebars" id="about">
            <div class="container">{{outlet}}</div>
        </script>
    </body>
</html>
```

4.3.2 Extending Plim with custom parsers

0.9.2 .

It is possible to extend standard Plim markup with your own directives. This feature allows you to build your own DSL on top of Plim. For instance, the following example adds a new directive for parsing HTTP links present in a form of `http_url > title`.

```
# my_module.py
import re
from plim import preprocessor_factory


PARSE_HTTP_LINKS_RE = re.compile('(?P<url>https?://[^>]+)+\s+>\s+(?P<title>.*)')


def parse_http_link(indent_level, current_line, matched, source, syntax):
    url = matched.group('url')
    url_title = matched.group('title')
    rt = '<a href="{}">{}</a>'.format(url, url_title)
    return rt, indent_level, '', source


CUSTOM_PARSERS = [
    (PARSE_HTTP_LINKS_RE, parse_http_link)
]


custom_preprocessor = preprocessor_factory(custom_parsers=CUSTOM_PARSERS, syntax='mako')
```

The `parse_http_link()` function is defined according to the strict API.

Every parser accepts five input arguments:

1. `indent_level` - an indentation level of the current line. When the parser reaches a line which indentation is lower or equal to `indent_level`, it returns control to a top-level function.
2. `current_line` - a line which is being parsed. This is the line that has been matched by `matched` object at the previous parsing step.
3. `matched` - an instance of `re.MatchObject` of the regex associated with the current parser.
4. `source` - an instance of an enumerated object returned by `plim.lexer.enumerate_source()`.
5. `syntax` - an instance of one of `plim.syntax.BaseSyntax` children.

Every parser returns a 4-tuple of:

1. parsed_data - a string of successfully parsed data

2. tail_indent - an indentation level of the `tail line`
3. tail_line - a line which indentation level (`tail_indent`) is lower or equal to the input `indent_level`.
4. `source` - an instance of enumerated object returned by `plim.lexer.enumerate_source()` which represents the remaining (untouched) plim markup.

From now on, we can use `custom_preprocessor` in exactly the same manner as the standard `plim.preprocessor`.

Let's create a plim document with extended syntax:

```
/ hamilton.plim
---------------
html
    head:title Alexander Hamilton
    body
        h1 Alexander Hamilton
        ul
            li: http://en.wikipedia.org/wiki/Alexander_Hamilton > Wikipedia Article
            li: http://www.amazon.com/Alexander-Hamilton-Ron-Chernow/dp/0143034758 > Full-length Bio
```

Here is how we can compile this document into a valid HTML (note the `-p` argument):

```
$ plimc -H -p my_module:custom_preprocessor hamilton.plim
```

The result:

```
<html>
    <head>
        <title>Alexander Hamilton</title>
    </head>
    <body>
        <h1>Alexander Hamilton</h1>
        <ul>
            <li><a href="http://en.wikipedia.org/wiki/Alexander_Hamilton">Wikipedia Article</a></li>
            <li><a href="http://www.amazon.com/Alexander-Hamilton-Ron-Chernow/dp/0143034758">Full-le
        </ul>
    </body>
</html>
```

4.4 Framework Integration

4.4.1 Pyramid

Add `plim.adapters.pyramid_renderer` into the `pyramid.includes` list of your .ini configuration file

```
[app:main]
pyramid.includes =
    # ... (other packages)
    plim.adapters.pyramid_renderer
```

The adapter will add the `.plim` renderer for use in Pyramid. This can be overridden and more may be added via the `config.add_plim_renderer()` directive:

```
config.add_plim_renderer('.plm', mako_settings_prefix='mako.')
```

The renderer will load its configuration from a provided mako prefix in the Pyramid settings dictionary. The default prefix is 'mako.'.

4.4.2 Syntax Highlighters

At this moment, Plim doesn't have syntax highlighters.

But, at a starting point you can use Slim syntax highlighters, since most of Plim syntax is the same as of Slim.

Editors support

- vim-plim - a Plim port of vim-slim plugin.

4.5 Command-line Interface

0.7.12 .

The package provides the command-line tool `plimc` to compile plim source files into Mako templates.

```
$ plimc -h
usage: plimc [-h] [--encoding ENCODING] source target

Compile plim source files into mako files.

positional arguments:
  source                path to source plim template
  target                path to target mako template

optional arguments:
  -h, --help            show this help message and exit
  --encoding ENCODING   source file encoding
```

4.6 License

Plim source code is licensed under the MIT license.

Plim Documentation is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License.

4.7 Authors

Plim was created by Maxim Avanov.

4.8 Contributors (in chronological order of the first contribution)

- Keith Yang - https://github.com/keitheis
- iMom0 - https://github.com/imom0
- dongweiming - https://github.com/dongweiming

See also a list of Slim authors.

4.9 Related projects

There is a number of similar projects that you might be interested in.

- slimish-jinja2
- PyJade
- mint
- SHPAML
- Yammy
- PyHAML
- HamlPy

See also a list of Slim related projects.

4.10 Changelog

4.10.1 Version 0.9

- 0.9.11
  - Hotfix: Fix incorrect parsing of templates with windows-style newlines (CR+LF).
- 0.9.10
  - Hotfix: Fix `plimc`'s inability to find a custom preprocessors module in the current working dir.
- 0.9.9
  - Hotfix: Fix UnicodeEncodeError in `-def` blocks with unicode strings as default argument values.
- 0.9.8
  - Change: Stylus extension no longer depends on the `nib` package.
- 0.9.7
  - Hotfix: Include requirements.txt into the distribution.
- 0.9.6
  - Hotfix: Conditional statements parser now can handle strings containing inline tag separator sequences (#27).
- 0.9.5
  - Hotfix: Fix `plimc` unicode decoding regression introduced by the previous hotfix.
- 0.9.4
  - **Hotfix: `plimc` no longer crashes with TypeError in Python3 environments** when it writes bytes to sys.stdout.
- 0.9.3
  - Hotfix: Fix UnicodeEncodeError in `plimc` when it writes to STDOUT.
- 0.9.2
  - Feature: added support for Custom Parsers.

- 0.9.1
  - New Syntax: New-style Python Blocks.
  - New Syntax: New-style Module-level Blocks.
- 0.9.0
  - Change: Pyramid adapter now relies on Pyramid>=1.5a2 and pyramid_mako>=0.3.1.
  - Change: The package now depends on Mako>=0.9.0.
  - Change: Sass/Scss extension now requires PyScss>=1.2.0.post3.
  - **Change: Pyramid adapter's `plim.file_extension` configuration option is deprecated.** The `config.add_plim_renderer()` directive is provided instead.

4.10.2 Version 0.8

- 0.8.9
  - Bugfix: Use sys.maxsize instead of unavailable sys.maxint on Python 3.
- 0.8.8
  - Hotfix: Make Plim working with a development version of pyScss for Python-3.x setups.
- 0.8.7
  - Bugfix: Pyramid adapter is now compatible with the 1.5a2+ version of the framework.
  - **Change: default template file extension** used in pyramid bindings is changed from ".plm" to ".plim".
- 0.8.6
  - Hotfix: fixed assertion error in handlebars parser.
- 0.8.5
  - Feature: added support for Handlebars blocks.
- 0.8.4
  - Hotfix: updated links to github.
- 0.8.3
  - Hotfix: prevent lexer from parsing embedded markup inside `style` and `script` blocks.
- 0.8.2
  - Feature: added support for Embedded Markup.
  - Feature: plimc utility is now able to output plain HTML.
- 0.8.1
  - Feature: added support for Inline Statements.
- 0.8.0
  - Feature: added support for dynamic attributes unpacker (an equivalent to Slim's splat attributes).

4.10.3 Version 0.7

- 0.7.14
  - Hotfix: fixed bug with unicode handling.
- 0.7.13
  - Hotfix: fixed bug with static unicode attributes.
- 0.7.12
  - Unnecessary newline characters at the end of literal blocks have been removed.
  - Added the command-line tool `plimc`.
- 0.7.11
  - Fixed bug that had to do with incorrect parsing of multi-line dynamic class attributes.
  - Fixed bug that had to do with passing incorrect data to plim parser in babel adapter.
- 0.7.10 Fixed bug with unicode error in python block. Thanks to sqrabs@github!
- 0.7.9 Added babel message extraction plugin.
- 0.7.8 Expanded range of possible numeric values that don't require double-quoting.
- 0.7.7
  - Fixed bug with linebreaks without trailing newline character.
  - Fixed bug with missing explicit whitespace after `=,` and `==,` line indicators.
- 0.7.6 Fixed bug with incorrect parsing of static boolean attributes.
- 0.7.5 Fixed bug with comment and content blocks separated by empty lines.
- 0.7.4 Added -stylus extension.
- 0.7.3 Fix bug with literal one-liners.
- 0.7.1 Fixed installation error caused by missing README.rst.
- 0.7.0 Initial public release.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

p

plim, 33

P

plim (), 33