pyy Documentation

Release 2.0

Tom Flanagan, Jake Wharton

Nov 27, 2018

Contents

pyy is a python library designed to allow for rapid creation of HTML5 applications through tight integration with your server-side code.

```
>>> print html(body(div('Forget templates and binding.')))
<html>
  <body>
    <div>Forget templates and binding.</div>
  </body>
</html>
```

The foundation of the library is in eliminating the need for an intermediate templating language as well as the weak and often redundant bindings that come with it. By coupling the HTML generation as simple object creation within your code you can seamlessly create the necessary HTML for your application using the expressive python syntax.

Contents:

CHAPTER 1

Getting Started

Coming soon.

CHAPTER 2

Features

Coming very soon.

Modules:

CHAPTER 3

pyy.cgi

CHAPTER 4

pyy.html

4.1 About

pyy.html allows for creating (X)HTML markup through the use of objects. This allows you to tightly integrate (X)HTML generation into your backend without the need of using an intermediate templating language.

pyy.html also provides you with helper classes for generating and parsing (X)HTML documents.

4.2 Usage

All these examples assume you have imported the appropriate tags or entire tag set (i.e. *from pyy.html.html import* *).

4.2.1 Hello, pyy!

Constructing a "Hello, World!"-style example is as easy as this:

```
>>> print html(body(h1('Hello, pyy!')))
<html>
  <body>
    <h1>Hello, pyy!</h1>
  </body>
</html>
```

4.2.2 Complex Structures

Through the use of the += operator and the *.add()* method you can easily create more advanced structures.

Create a simple list:

```
>>> list = ul()
>>> for item in range(4):
>>>   list += li('Item #', item)
>>> print list
<ul>
  <li>Item #0</li>
  <li>Item #1</li>
  <li>Item #2</li>
  <li>Item #3</li>
</ul>
```

If you are using a database or other backend to fetch data, *pyy.html* supports iterables to help streamline your code:

```
>>> print ul(li(a(name, href=link), __inline=True) for name, link in menu_items)
<ul>
  <li><a href="/home/">Home</a></li>
  <li><a href="/about/">About</a></li>
  <li><a href="/downloads/">Downloads</a></li>
  <li><a href="/links/">Links</a></li>
</ul>
```

A simple document tree:

```
>>> _html = html()
>>> _body = _html.add(body())
>>> header  = _body.add(div(id='header'))
>>> content = _body.add(div(id='content'))
>>> footer  = _body.add(div(id='footer'))
>>> print _html
<html>
  <body>
    <div id="header"></div>
    <div id="content"></div>
    <div id="footer"></div>
  </body>
</html>
```

For clean code, the *.add()* method returns children in tuples. The above example can be cleaned up and expanded like this:

```
>>> _html = html()
>>> _head, _body = _html.add(head(title('Simple Document Tree')), body())
>>> names = ['header', 'content', 'footer']
>>> header, content, footer = _body.add(div(id=name) for name in names)
>>> print _html
<html>
  <head>
    <title>Simple Document Tree</title>
  </head>
  <body>
    <div id="header"></div>
    <div id="content"></div>
    <div id="footer"></div>
  </body>
</html>
```

You can modify the attributes of tags through a dictionary-like interface:

```
>>> header = div()
>>> header['id'] = 'header'
>>> print header
<div id="header"></div>
```

Comments can be created using objects too!

```
>>> print comment('BEGIN HEADER')
<!--BEGIN HEADER-->
>>> print comment(p('Stop using IE5!'), condition='lt IE6')
<!--[if lt IE6]>
<p>Stop using IE5!</p>
<![endif]-->
```

4.2.3 Creating Documents

Since creating the common structure of an HTML document everytime would be excessively tedious pyy.html provides a class to create an manage them for you, *document*.

When you create a new document, the basic HTML tag structure is created for you.

```
>>> d = document()
>>> print d
<html>
  <head>
    <title>PYY Page</title>
  </head>
  <body></body>
</html>
```

The *document* class also provides helpers to allow you to access the *html*, *head*, and *body* elements directly.

```
>>> d = document()
>>> d.html
<pyy.html.html.html: 0 attributes, 2 children>
>>> d.head
<pyy.html.html.head: 0 attributes, 0 children>
>>> d.body
<pyy.html.html.body: 0 attributes, 0 children>
```

You should notice that here the *head* tag contains zero children. This is because the default *title* tag is only added when the document is rendered and the *head* element already does not explicitly contain one.

The *document* class also provides helpers to allow you to directly add elements to the *body* tag.

```
>>> d = document()
>>> d += h1('Hello, World!')
>>> p += p('This is a paragraph.')
>>> print d
<html>
  <head>
    <title>PYY Page</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
    <p>This is a paragraph.</p>
```

(continues on next page)

(continued from previous page)

```
  </body>
</html>
```

4.2.4 Markup Validation

You can also set the DOCTYPE of the *document* which will validate the tag tree when it is rendered.

```
>>> d = document()
>>> d.doctype = dtd.xhtml11
>>> print d
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/
↪xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>PYY Page</title>
  </head>
  <body></body>
</html>
```

Notice how the required XHTML 1.1 attribute *xmlns* is automatically added to the <*html*> tag.

If there are any errors in the tag tree a *ValueError* will be thrown which describes the offending tag. You can check the validity of your document at any time by calling *validate()* explicity.

You may also set a *document*'s DOCTYPE with the keyword argument *doctype*.

```
>>> d = document(doctype=dtd.html5)
```

However, doing so will cause the entire document to be validated with every added tag. For production environments it is best to set the DOCTYPE after the entire tag tree has been created so it all is only validated once.

4.2.5 Parsing Documents

The *parser* class contains two methods: *parse* and *pageparse*. *parse* will take valid and mostly-valid (X)HTML input and return a tag tree. *pageparse* will take an entire document and return a *document* instance complete with the DOCTYPE (if present) and a tag tree using the (X)HTML version specified by the DOCTYPE.

parse will simply return a heirarchy of all the tags and their content that it can recognize in a string.

```
>>> parse('<p>Hello.</p>')
<pyy.html.html.p: 0 attributes, 1 child>
>>> parse('<html><head><title>test</title></head><body><h1>Hi.</h1></body></html>')
<pyy.html.html.html: 0 attributes, 2 children>
>>> parse('<div id="first"></div><div id="second"></div>')
[<pyy.html.html.div: 1 attribute, 0 children>, <pyy.html.html.div: 1 attribute, 0␣
↪children>]
```

Notice that if multiple top-level tags exist in the string *parse* will return them as an array.

pageparse also takes a string of tags and optionally a DOCTYPE and returns a *document* object.

```
>>> pageparse('<!DOCTYPE html><html><head><title>Test</title></head><body></body></
↪html>')
<pyy.html.document.document html5 "Test">
```

4.3 Developed By

- Tom Flanagan - <theknio@gmail.com>
- Jake Wharton - <jakewharton@gmail.com>

Git repository located at github.com/Knio/pyy

4.4 License

CHAPTER 5

pyy.server

CHAPTER 6

pyy.web

6.1 About

pyy.web contains useful classes and functions to aid in dealing with the HTTP protocol and serving websites.

6.2 Usage

6.2.1 Cookies

pyy.web.cookie allows for easy creation of properly formatted cookies.

The cookie name and value are the only two required fields for its simplest form:

```
>>> cookie('test', 'cookie')
<pyy.web.cookie.cookie test=cookie; path=/;>
```

You can also pass any number of the following arguments: * *expires* - Expiration date (mutually exclusive with *duration*). * *duration* - Length of cookie (mutually exclusive with *expires*). * *path* - The URL path of the cookie (default: */*). * *domain* - The cookie's domain. * *secure* - Whether or not the cookie is secure (default: *False*). * *httponly* - If the cookie applies to only HTTP (default: *False*).

While the string and *__repr__* outputs contain the rendered cookie you can also call *render()* directly which takes an optional *is_header* boolean.

```
>>> cookie('test', 'cookie').render()
'test=cookie; path=/;'
>>> cookie('test', 'cookie').render(True)
'Set-Cookie: test=cookie; path=/;'
```

6.2.2 HTTP Messages

pyy.web.httpmessage contains the classes for the *httprequest* and *httpresponse* objects. These two classes hold information on the HTTP requests and the HTTP responses, respectively, that the other modules pass between themselves.

6.2.3 Parsers

The *pyy.web.parsers* class contains four functions which aid in parsing common formats of the HTTP protocol into a more friendly and usable format.

- *parse_query(string)* - Parses *a=b&c=d* format into a dictionary. Also supports arrays.
- *parse_semi(string)* - Parses *a=b; c=d* format into a dictionary.
- *parse_user_agent(string)* - Parses a user agent string and returns a browser object containing vendor and version.
- *parse_multipart(content_type, data)* - Parses multipart encoded form data into a dictionary.

6.2.4 Resolvers

When serving dynamic web applications it is sometimes useful to obfuscate your URLs or simple clean them up to be "pretty". The *resolvers* class allows you to create logical rules or patterns for your URLs and have them easily be mapped back to approprite classes.

There are currently two methods of dynamic URL resolving: Regex-based and file heirarchy-based.

Regex-based resolving takes a list of tuples which associate a regular expression to a class. When a request is resolved in this way the tuples are iterated over and the first match is taken and returned.

urls = [(r'^/photos/', pages.photos), (r'^/videos/', pages.videos),

]

If a match is not found, an *httperror* is raised with a 404 code.

You can also use named match groups in your regular expressions to help parse the URL which will be copied into the request's *GET* mapping.

urls = [(r'^/photos/((?P<id>d+)/)?$', pages.photos)

]

Here an optional ID can be specified after the */photos/* qualifier and the *pages.photos* class can react accordingly.

When you create a directory like the following:

pages/ __init__.py home.py admin/

__init__.py users.py

you can use corresponding URLs to access those classes: / (would match in __init__.py) /home/ (would match home.py) /home/page2/ (would match home.py with 'page2' put into the request's GET) /admin/ (would match admin/__init__.py) /admin/users/ (would match admin/users.py) /admin/other/ (would match admin/__init.py with 'other' put into the request's GET)

In each of those files should be an Index class which is the default one selected in a file. If in the case of the */admin/other/* URL the *admin/__init__.py* file had an *Other* class it would be the one that was selected.

6.3 Developed By

- Tom Flanagan - <theknio@gmail.com>
- Jake Wharton - <jakewharton@gmail.com>

Git repository located at github.com/Knio/pyy

6.4 License

Python Module Index

p

Index

P