yawrap Documentation

Release 0.4.0

Michal Kaczmarczyk

Jul 12, 2018

Contents

Yawrap is a powerful, lightweight, pythonic *pseudo-static HTML* builder that works with:

- `python2.7`,
- `python3.4-python.3.6`
- `pypy`.

The name comes from something like *Yet Another Wrapper (of HTML code)*.

Repo:	https://github.com/kamichal/yarap
OldRepo:	https://bitbucket.org/gandowin/yarap (goes obsolete)
Docs:	http://yawrap.readthedocs.io
Pypi:	https://pypi.python.org/pypi/yawrap
Author:	Michał Kaczmarczyk from Poland
Email:	*michal.skaczmarczy.k at gmail.com*

CHAPTER 1

Features

- **Very nice syntax**

  No more headache caused by closing and indentation of HTML elements! Just write python code.

  Yawrap reflects python scopes in HTML perfectly - with no mistakes and indents it natively for free.

- ***CSS* & *JS* support** Handle it how you like. JS and CSS can be sourced either:
  - from local file
  - from url
  - from python string

  And it can be placed either:
  - as internal content
  - as external file
  - as linked resource.

  From single "All in one" HTML file to multi-page documents sharing CSS&JS resources. Yawrap takes care for handling them properly.

- **SVG support** Don't care about defining SVG structure, just write its main contents. Yawrap will take care about the whole rest. Also typical SVG attributes which are problematic from python keyword-arguments point of view have it's convenience feature.

- **Linking local files** You can reference local files by passing its absolute path on python side and it will appear under links relative to the current document. And you don't have to *calculate the paths*.

- ***Defining page style and scripts on python class level*** Page styles can be defined by deriving Yawrap classes. This makes possibility to get the styles shared / inherited / override in well known *pythonic* way.

- ***Multi-page structure*** Define multiple pages (even in complex directory structure) and don't care about the paths. Not existing directories will be automatically created, you just define the path of target file.

- **Automatic navigation** That's ultra-nice. **Probably the cutest yawrap's feature.** Create multiple pages and see how yawrap joins them in navigation panel created for each generated page. Everything happens behind the curtains.

  The only one thing you need to do is to care for the navigation's `CSS` style (if you don't like the default navigation style provided by Yawrap).

- **Bookmarking** Create intra-page bookmarks with just one command during document definition and see how they are inserted in correct subsection of the page navigation.

CHAPTER 2

Usage Examples

Because a good example is worth more than thousand words.

2.1 Basic Usage

Passing target file location to Yawrap constructor is mandatory, although the file will be not generated until `Yawrap.render()` is called. Title is optional.

```
from yawrap import Yawrap


def hello_yawrap():
    doc = Yawrap("not/revelant/path.html", "The super title")

    with doc.tag("div", id="content", klass="main"):
        with doc.tag('h2'):
            doc.text('Hello yawrap!')

        with doc.tag('p'):
            doc.text('Could it be simpler?')

    doc.comment("that's it")
    result = doc.getvalue()

    assert result == """\
<!doctype html><html lang='en-US'>
  <head>
    <meta charset='UTF-8' />
    <title>The super title</title>
  </head>
  <body>
    <div class='main'>
      <h2>Hello yawrap!</h2>
```

(continues on next page)

(continued from previous page)

```
      <p>Could it be simpler?</p>
    </div>
    <!-- that's it -->
  </body>
</html>"""
```

Will create a page that looks like this:

2.2 Basic JS usage

It's an adaptation of following example:

https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_fadetoggle

```
from yawrap import Yawrap, ExternalJs, EmbedCss, EmbedJs, BODY_END


class MyPageTemplate(Yawrap):
    resources = [
        ExternalJs("https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js
↪"),
        EmbedCss("""\
        body {
            padding: 12px;
            font-family: helvetica, sans-serif;
            font-size: 14px;
        }
        .box {
            display: inline-block;
            height: 80px;
            margin: 8px;
            padding: 8px;
            width: 80px;
        }"""),
        EmbedJs("""\
        $("button").click(function(){
            $("#red-box").fadeToggle();
            $("#green-box").fadeToggle("slow");
            $("#blue-box").fadeToggle(3000);
        });""", placement=BODY_END),
    ]

    def _create_box(self, name, color):
        style = "background-color:{};".format(color)
        with self.tag('div', id=name, klass="box", style=style):
            self.text(name)

    def fill_with_content(self):
        with self.tag("h2"):
            self.text("Demonstrate a simple JavaScript.")

        with self.tag('p'):
            self.text("fadeToggle() operating with different speed parameters.")
```

(continues on next page)

(continued from previous page)

```
        with self.tag('button'):
            self.text("Click to fade in/out boxes")

        with self.tag("div"):
            self._create_box('red-box', "red")
            self._create_box('green-box', "#0f0")
            self._create_box('blue-box', "#0a1cf0")


def create_page(output_file_path):
    doc = MyPageTemplate(output_file_path, "Name of the page")
    doc.fill_with_content()
    doc.render()

# that's it
```

Will create a page as below. Note that the `placement=BODY_END` in `EmbedJs` call caused the script to be placed at the end. Withouth that argument it would get into `head` section.

That looks like this:

2.3 Ways to feed it with CSS / JS resources

There are several possibilities to place a CSS and JS in html page. Each has some advantages. Yawrap helps handling probably all of them.

```
from yawrap import Yawrap


class PlainPage(Yawrap):
    resources = []


doc = PlainPage("/tmp/test.html")
doc.render()
```

Will create a page that has no style sheet (and no content) defined. It's just to show the *Tabula rasa* case, when the resources list is empty.

OK, but of course you would like to use the styles and scripts.

2.3.1 External resources

This is how to reference resources hosted on external server.

```
from yawrap import ExternalCss, ExternalJs
```

(continues on next page)

(continued from previous page)

```
class PageWithExternalCss(Yawrap):
    resources = [
        ExternalCss.from_url("https://www.w3schools.com/w3css/4/w3.css"),
        ExternalJs.from_url("https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/
↪jquery.min.js"),
    ]
```

That gives:

Disadvantage of such approach is of course - dependency on other web resource, . . . but Yawrap doesn't care it's your business.

2.3.2 Auto-Linked resources

Yawrap can download the sheet or script given by URL (while page build), save it as a local file and manage its relative path calculation.

Just replace `ExternalCss` with `LinkCss` and `ExternalJs` with `LinkJs` like this:

```
from yawrap import LinkCss, LinkJs


class LinkedCssPage(Yawrap):
    resources = [
        LinkCss.from_url("https://www.w3schools.com/w3css/4/w3.css"),
        LinkJs.from_url("https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.
↪min.js"),
    ]
```

With such a result:

See how these link point to local files in `resources` directory? It's a default location, next to output html file. Yawrap created it while build.

Let's say it once again *Yawrap downloads and saves the files with each run.*

If you allways want to get a fresh copy from these urls, it's ok. But it's probably eaiser to download it manually, save it close to your yawrap generator code. While each run, yawrap will source its contents from local files, instead of the web. Rest (i.e. saving to target resources dir) is the same as before.

Such a code will source the linked code from given files:

```
class LocalSources(Yawrap):
    resources = [
        LinkCss.from_file("path/to/source_of/w3.css"),
        LinkJs.from_file("path/to/source_of/jquery.min.js"),
    ]
```

2.3.3 Changing default resources path

The default `resources` output directory can be changed like this:

```
class LinkCssInMyStyle(LinkCss):
    resource_subdir = "the_super_directory"
    # for nested structure, do:
#   # resource_subdir = os.path.join("the", "super", "directory")


class LinkJsInMyStyle(LinkJs):
    resource_subdir = ""
    # empty string will same dir, as parent html


class MyStyleOfLinking(Yawrap):
    resources = [
        LinkCssInMyStyle.from_url("https://www.w3schools.com/w3css/4/w3.css"),
        LinkJsInMyStyle.from_url("https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/
↪jquery.min.js"),
    ]
```

This code will generate exactly the same HTML, just paths differ.

Note that JS and CSS will go to separate dirs.

Note: Relative path management makes sense when several pages from different paths share same CSS/JS file. E.g. when `NavedYawrap` is in use. Then the `resources` directory path is threated as relative to the top level of output directory structure (relative to root, the first page).

2.3.4 Defining CSS / JS in python code

Sometimes you would like to keep definition of the CSS or JS inside a python module. Especially if you use python for generating it (e.g. calculate CSS colors, margin sizes, etc..)

Then there are

If you are interested with getting single-file page (all-in-one, no linked CSS nor JS), where everything is embedded in single html, then use such an pattern:

Replacing a *resource sourcer* `LinkCss` with `EmbedCss` as here:

```
LinkCss.from_url("https://www.w3schools.com/w3css/4/w3.css")    # with
EmbedCss.from_url("https://www.w3schools.com/w3css/4/w3.css")
```

will cause downloading the w3 style sheet and placing it directly to the `/html/head/style` section. The download happens with each call of `Yawrap.render()`.

It also embeds additional style for body defined as a python string.

The most useful in my opinion is to create own class derived from `Yawrap` (or `NavedYawrap` or whatever from the family).

CHAPTER 3

Contents

3.1 CSS Support

Yawrap has it's basic support of cascade style sheets. During creation of html page you can append styles definition as strings or dictionaries. It will appear in `/html/head/style` section of current yawrap document. Also linking CSS as external file is possible.

Note: The `class` keyword, widely used in `html/css` will cause python's syntax error if used as keyword argument, so you can define it by typing `klass` instead. Yattag will convert it to `class` automatically.

3.1.1 Internal CSS as str

```
from yawrap import Yawrap, EmbedCss
out_file = '/tmp/css_1.html'
jawrap = Yawrap(out_file)

with jawrap.tag('div', klass='content'):
    with jawrap.tag('span'):
        jawrap.text('CSS in yawrap.')

jawrap.add(EmbedCss('''
.content { margin: 2px; }
.content:hover {
   background-color: #DAF;
}'''))
jawrap.render()

print(open(out_file, 'rt').read())
```

That outputs:

```
<!doctype html><html lang='en-US'>
  <head>
    <meta charset='UTF-8' />
    <style>
.content { margin: 2px; }
.content:hover {
   background-color: #DAF;
}</style>
  </head>
  <body><div class='content'><span>CSS in yawrap.</span></div></body>
</html>
```

The method `add()` called with action `EmbedCss()` appends CSS definitions to page head section. It can be called several times, output CSS will be sorted and nicely joined.

The passed CSS can be a string without any formatting. It will be reformatted during page generation.

3.1.2 Internal CSS as python's dict

The argument passed to `add_css()` can be a regular python dictionary definins css.

```
css_dict = {
  '.content': {
    'color': '#321',
    'padding': '1px 16px'
  },
  'span': {
    'border': '1px solid black'
  }
}
# reusing jawrap instance from subsection above.
jawrap.add(EmbedCss(css_dict))
jawrap.render()

print(open(out_file, 'rt').read())
```

Will give:

```
<!doctype html><html lang='en-US'>
  <head>
    <meta charset='UTF-8' />
    <style>
.content { margin: 2px; }
.content:hover {
   background-color: #DAF;
}</style>
    <style>

  .content {
    color: #321;
    padding: 1px 16px;
  }
  span {
    border: 1px solid black;
  }
    </style>
```

(continues on next page)

(continued from previous page)

```
  </head>
  <body><div class='content'><span>CSS in yawrap.</span></div></body>
</html>
```

Note the previous `.content` selector's definition is overwritten with new one.

3.1.3 External CSS from local file

It's also possible to link style sheet from local file. It's source can be placed anywhere as long as it's accessible for build process. Yawrap will copy it and place in default `resources` directory, next to target file (or next to root document):

```
from yawrap import Yawrap, LinkCss
out_file = '/tmp/css_2.html'

jawrap = Yawrap(out_file)
jawrap.text('CSS from local file.')
jawrap.add(LinkCss.from_file('/tmp/files/my.css'))
jawrap.render()
```

3.1.4 External CSS from web

Using global CSS from some resources can be obtained by calling `add()` with `ExternalCss` object.

```
from yawrap import Yawrap, ExternalCss
out_file = '/tmp/css_3.html'

jawrap = Yawrap(out_file)
jawrap.text('CSS from web.')
jawrap.add(ExternalCss("https://www.w3schools.com/w3css/4/w3.css"))

html = jawrap.getvalue()
print(html)
```

```
<!doctype html><html lang='en-US'>
  <head>
    <meta charset='UTF-8' />
    <link type='text/css' href='https://www.w3schools.com/w3css/4/w3.css' rel=
↪'stylesheet' />
  </head>
  <body>CSS from web.</body>
</html>
```

3.1.5 CSS defined on class level

You can derive own class from `Yawrap` or `Navrap` class and define its CSS that will be inherited in its subclasses. You have to define *css* class attribute either as a string or a dictionary.

```
from yawrap import Yawrap, EmbedCss
out_file = '/tmp/css_4.html'

```

(continues on next page)

(continued from previous page)

```
class MyStyledPage(Yawrap):
    resources = [EmbedCss('''\
        body {
          margin: 0px;
          padding: 13px 14px;
        }
        .content {
           color: #BAC;
           margin: 2px;
        }''')
]

myStyled = MyStyledPage(out_file)
with myStyled.tag('div', klass='content'):
    myStyled.text('Deriving CSS.')

myStyled.render()

print(open(out_file, 'rt').read())
```

Should give:

```
<!doctype html><html lang='en-US'>
  <head>
    <meta charset='UTF-8' />
    <style>
        body {
          margin: 0px;
          padding: 13px 14px;
        }
        .content {
           color: #BAC;
           margin: 2px;
        }
    </style>
  </head>
  <body><div class='content'>Deriving CSS.</div></body>
</html>
```

Adding CSS is still possible, but to instance of the derived class (to `myStyled` above), not to the class definition (here `MyStyledPage`), so the appended CSS will not be inherited.

3.2 JavaScript support

Yattag does not modify nor even analyze added `js` sources. You provide them as strings and these are rendered in `/html/head/script` section. Each addition places the content in separate `<script>` section.

Of course there is also possibility to link external `js` source file.

3.2.1 Internal JS

Appending js to `Yawrap` or `Navrap` instances can be done appending string-code to Class

Note: Yawrap has a class-level atribute being a list named `js`. It is supposed to contain JavaScript code as strings. Similary as with CSS, you can derive from `Yawrap` class and also define the class-level JS that will be inherited by its subclasses unless you override it.

3.2.2 Using jQuery

It's an adaptation of following example: https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_fadetoggle

```
from yawrap import Yawrap, ExternalJs, EmbedCss, EmbedJs, BODY_END


class MyPageTemplate(Yawrap):
    resources = [
        ExternalJs("https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js
↪"),
        EmbedCss("""\
        body {
            padding: 12px;
            font-family: helvetica, sans-serif;
            font-size: 14px;
        }
        .box {
            display: inline-block;
            height: 80px;
            margin: 8px;
            padding: 8px;
            width: 80px;
        }"""),
        EmbedJs("""\
        $("button").click(function(){
            $("#red-box").fadeToggle();
            $("#green-box").fadeToggle("slow");
            $("#blue-box").fadeToggle(3000);
        });""", placement=BODY_END),
    ]

    def _create_box(self, name, color):
        style = "background-color:{};".format(color)
        with self.tag('div', id=name, klass="box", style=style):
            self.text(name)

    def fill_with_content(self):
        with self.tag("h2"):
            self.text("Demonstrate a simple JavaScript.")

        with self.tag('p'):
            self.text("fadeToggle() operating with different speed parameters.")

        with self.tag('button'):
            self.text("Click to fade in/out boxes")

        with self.tag("div"):
            self._create_box('red-box', "red")
```

(continues on next page)

(continued from previous page)

```
            self._create_box('green-box', "#0f0")
            self._create_box('blue-box', "#0a1cf0")


def create_page(output_file_path):
    doc = MyPageTemplate(output_file_path, "Name of the page")
    doc.fill_with_content()
    doc.render()

# that's it
```

Will create a page as below. Note that the `placement=BODY_END` in `EmbedJs` call caused the script to be placed at the end. Withouth that argument it would get into `head` section.

That looks like this:

3.2.3 Sharing scripts and styles across multiple pages

Similar effect as above but with reusable java scripts and CSS can be obtained by defining them as class level attributes like this:

```
from yawrap import LinkCss, LinkJs


class MyPage(MyPageTemplate):
    resources = [
        ExternalJs("https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js
↪"),
        LinkCss("""\
        body {
            padding: 12px;
            font-family: helvetica, sans-serif;
            font-size: 14px;
        }
        .box {
            display: inline-block;
            height: 80px;
            margin: 8px;
            padding: 8px;
            width: 80px;
        }""", file_name="common_linked.css"),
        LinkJs("""\
        $("button").click(function(){
            $("#red-box").fadeToggle();
            $("#green-box").fadeToggle("slow");
            $("#blue-box").fadeToggle(3000);
        });
        """, placement=BODY_END, file_name="common_linked.js"),
    ]

    # uses methods inherited from MyPageTemplate defined above
```

(continues on next page)

(continued from previous page)

```
def create_page_with_linked_resources(output_file_path):
    doc = MyPage(output_file_path)
    doc.fill_with_content()
    doc.render()
```

Which produces the page:

..and the script `resources/common_linked.js`:

..and the style `resources/common_linked.css`:

3.3 Multiple-page structure with navigation

It's easy to navigate between several files in different paths, using `sub()` method of NavedYawrap instance. Each call of `sub()` returns instance of given NavedYawrap class, with same styles and scripts as the parent one. Each page will be rendered with navigation section, with relative paths. See the hrefs in the example below.

The first document (`home` in this example) will be the *root* of the document structure.

```
def test_naved_yawrap():

    from exampling_tools import get_output_file_path
    from yawrap import NavedYawrap, EmbedCss

    class MyPage(NavedYawrap):
        resources = [EmbedCss("""\
        body {
            margin: 16;
            font-family: Verdana, sans-serif;
        }""")
                     ]

    out_file_1 = get_output_file_path("nav01a.html")
    home = MyPage(out_file_1, title="Title Force One")

    # fill content of root document
    with home.tag('p'):
        home.text("I'm home")

    # create a sub-document
    out_file_2 = get_output_file_path(os.path.join("some", "deep", "nonexistent",
↪"path", "nav01b.html"))
    about = home.sub(out_file_2, 'Abouting')

    with about.tag('div'):
        about.text('Always do the abouting!')

    home.render_all_files()

    # at this point files are generated
```

Generates two html pages:

Notice, how the href is calculated.

Here's the page:

3.4 Examples

This is a tiny example. Such a code is sufficient to generate an html page:

```
>>> from yawrap import Yawrap

>>> jawrap = Yawrap('/tmp/example_1.html', 'the title')
>>> with jawrap.tag('div'):
...     with jawrap.tag('p'):
...         jawrap.text('Nothing much here.')
```

After executing that - calling *render()* function will store the page in a target file specified in *Yawrap* constructor:

```
>>> jawrap.render()
>>> print(open('/tmp/example_1.html', 'rt').read())
<!doctype html><html lang='en-US'>
  <head>
    <meta charset='UTF-8' />
    <title>the title</title>
  </head>
  <body><div><p>Nothing much here.</p></div></body>
</html>
```

3.5 yattag's heritage

Till `Yawrap` version 0.3.2 - its core was yattag. It used it as a library. But it's been decided to reimplement the XML-tag engine in own way. Yawrap requred more features and yattag was not able to provide it. Since version 0.4. Yawrap has no external dependencies at all.

Note: Yawrap is backward compatible with yattag. Vast functionality of `yattag.SimpleDoc` is supported. You can use yawrap in the same way as yattag or even just replace import statement in your existing code. Unfortunatelly I cannot guarantee 100% match, but I'm pretty sure it will work.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search