HTM(x)L

Release 0.8.5

unknown

Jul 27, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Status <https://coveralls.io/github/schireson/htmxl?branch=master>`_

1.1 Introduction

```
from htmxl.compose import Workbook

workbook = Workbook()
workbook.add_sheet_from_template(
    template="""
    <head>{{ title }}</head>
    <body>
      <div>
        Hello down there, {{ name }}!
      </div>
      <div>
        <table>
          <thead>
            <tr>
              {% for column_name in column_names %}
                <th>{{ column_name }}</th>
              {% endfor %}
            </tr>
          </thead>
          <tbody>
            {% for row in rows %}
              <tr>
                <td>{{ row.a }}</td>
                <td>{{ row.b }}</td>
              </tr>
            {% endfor %}
          </tbody>
        </table
      </div>
    </body>
    """,
    data=dict(
        title="Hello World",
        name='Bob',
```

(continues on next page)

(continued from previous page)

```
        column_names=['A', 'B'],
        rows=[{'a': 'a', 'b': 2}, {'a': 'b', 'b': 2}, {'a': 'c', 'b': 3}],
    )
)

workbook.compose('hello_world.xlsx')
```

1.2 The Pitch

Essentially, HTM(x)L is an attempt to allow the declaration of Excel files in a (mostly) declarative manner that separates the format that the document should take from the data being added to it.

The "normal" manner of producing Excel files essentially amounts to a series of imperitive statements about what cells to modify to get the excel file you want. For any file of moderate complexity, this leads to there being very little intuition about what the resulting Excel file will actually look like.

Particularly once you start adding cell styling, or finding yourself inserting dynamically-sized data (requiring a bunch of cell offset math), the relative ease of designing and visualizing the template using a familiar idiom, HTML, can be make it much much easier to author and maintain these kinds of templates.

1.3 Features

General features include:

- HTML elements as metaphors for structures in the resulting Excel document

  Due to the obviously grid-oriented structure of Excel, the metaphors **can** sometimes be approximate, but should hopefully make intuitive sense!

  For example:

  – `<span>`: An inline-block element which pushes elements after it to the right

  – `<div>`: An block element which push elements after it downward

  – `<table>`: Self-explanatory!

  See the documentation about elements for more details

- Styling

  Some commonly/obviously useful and style options like width/height (`style="width:  50px"`) or rowspan/colspan `colspan="2"` have been implemented, but there's no reason that various different options that make intuitive sense (like colors) could be implemented also

  See the documentation about styling for more details

- Classes

  While inline color styles are not (yet) implemented, one can supply classes normally, `class="class1 class2"` and supply the definitions for those classes as inputs to the Workbook

```
styles = [
    {"name": "odd", "pattern_fill": {"patternType": "solid", "fgColor": "FBEAFB"}}
↪,
]
Workbook(styles=styles)
```

1.4 Installation

There is no default parser (for HTML) included with a default installation of the package. We do this for both backwards compatibility and future compatibility reasons.

In order to keep the base package lean when one opts to use one or the other parser, we include a set of bundled parser-adaptor implementations for known supported parser libraries

To opt into the installation of the dependent library for the parser you chose:

```
# Slower, but more permissive
pip install htmxl[beautifulsoup]

# Faster, but more strict
pip install htmxl[lxml]
```

By default, the package will detect installed parser libraries and choose the first one we find, so a vanilla `Workbook()` instantiation should Just Work.

However, we encourage users to explicitly select their parser to avoid inadvertant selection of the "wrong" parser at runtime (given that they have template compatibility issues)

```
from htmxl.compose import Workbook

workbook = Workbook(parser='beautifulsoup')
workbook = Workbook(parser='lxml')
```

CHAPTER

TWO

LIBRARY DESIGN

Key Ideas:

- Custom composed excel documents require a markdown capability to design and update
- **HTML is an exceptional markdown language for this:**
    - Native support for styling of blocks of content
    - Commonly understood semantics for basic tags
    - syntax for adding style and attributes to blocks
    - Existing templating dialects
    - Extremely good support for table design
- Templating languages exist to dynamically generate HTML from data

2.1 Supported Elements

The root `<html>` tag is supported, but optional. One can do any of:

- Define html content nested within an `<html>` block
- Define top-level `<head>` and `<body>` sibling blocks
- Omit `<body>` entirely (if you have no `<head>`) and define all markup at the top level.

2.1.1 <head>

This block is optional.

Today, any raw content in the `head` section is essentially a shortcut for specifying a page `<title>`.

<title>

Any raw content is interpreted as the Excel sheet name.

2.1.2 <body>

This block is optional, should you not define a `<head>` block for setting the sheet name.

The child elements defined within a body follow a number of different "block" semantics which essentially attempt to mimic the semantics of HTML itself.

At a high level, they boil down to a few different options that we'll call:

- top-level
- top-right
- bottom-left
- bottom-right

The name indicates where the "cursor" will be left after exiting the context of the enclosing block being defined.

For example a `<div>` (described in the next section) is a bottom-left-type, which means that after performing the layout for the content of that div, the bounding box defined by the interior content of the `<div>` will be calculated and layout for sibling elements **after** that div will start at the bottom-left of that bounding box.

```
+- div -------------------------------------------+
|+- table ---------------------------------------+|
||+- thead -------------------------------------+||
|||+- th ----+-- th -----+-- th -----+ -- th --+|||
|||| example | example 2 | example 3 | example ||||
|||+---------+-----------+-----------+---------+|||
||+---------------------------------------------+||
||                                               ||
||+- tbody -------------------------------------+||
|||+- tr --------------------------------------+|||
||||+- td ---+-- td ----+-- td ----+ -- td ---+||||
||||| qwertu | abcdefgh | 23456789 | 12345678 |||||
||||+--------+----------+----------+----------+||||
||||                                           ||||
||||+- td ---+-- td ----+-- td ----+ -- td ---+||||
||||| qwertu | abcdefgh | 23456789 | 12345678 |||||
||||+--------+----------+----------+----------+||||
|||+-------------------------------------------+|||
||+---------------------------------------------+||
|+-----------------------------------------------+|
+-------------------------------------------------+

+- div -------------------------------------------+
+ ... more content ...                            |
+-------------------------------------------------+
```

Note, ASCII/graphics will not be a perfect visualization, as it pushes the content to the right as nesting occurs, which will not occur in Excel. However, it should give a sense of where content will start after certain kinds of elements close, and hopefully should give some initial intuition what *kind* (top-left, bottom-left, top-right) each element might be.

<div>

A `div` is a bottom-left element. Content after a `div` will resume in the cell directly below the `div`'s interior content.

It has no behavior other than the laying out of child elements.

<span>

A `<span>` is a top-right element. Content after a `<span>` will resume in the cell directly to the right of the `<span>`'s interior content.

Generally a `<span>` will only contain raw text content.

<br>

A `<br>` is a bottom-left element. Content after a `<br>` will resume in the cell directly below the `<br>`. It has no interior content.

<table>

A `<table>` is a bottom-left element. Content after a `<table>` will resume in the cell directly below the `<table>`.

A `<table>` is essentially a `<div>`, but for the purpose of defining a visual table structure.

<thead/tbody>

These elements exist primarily to group the table content, but otherwise will not affect layout, relative to the table itself. They are both bottom-left element types.

It **can** be a convenient markup location to apply styling (which will naturally cascade to the relevant subsections of the table).

<tr>

A `<tr>` is a bottom-left element. Content after a `<tr>` will resume in the cell directly below the `<tr>`.

<th/td>

These elements are semantically equivalent to a `<span>`, but for the purpose of defining a table's cell content.

<datalist>

A `<datalist>` element does not get directly rendered into the excel file. As such a datalist element will not have any effect on the placement of the cursor or layout of actual DOM elements.

Referencing the documentation on what a datalist does, it will register the set of `<option>` children as a form of cell validation in the document.

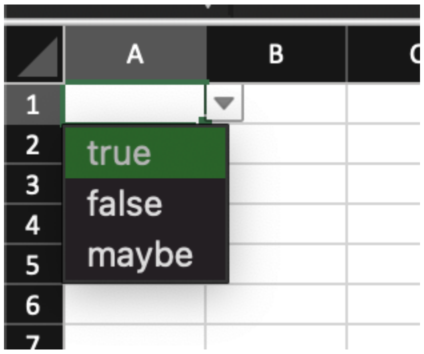
For example:

```
<datalist id="truthy-values">
  <option value="true" />
  <option value="false" />
  <option value="maybe" />
</datalist>

<input list="truthy-values">
```

In this case, an excel data validation will be registered and then applied to the cell pointed at by the `<input>`.

That will then be represented in the excel file like so



2.2 Styling Cells

The styling of cells is done using the html `class` attribute on an element. Using a style works similarly to the normal html. To apply a style "class", you simply set `class="style-name"` on an element.

You can compose a specific style from a number of more generic styles by simply including all the styles separated by a space i.e. `class="xl-centered xl-underlined"`. Simlar to html, in the event 2 styles set the same style attribute, the last one to set a particular style attribute "wins".

2.2.1 Make use of default styles

The library exposes some basic builtin styles for convenience. To see the full set, for the current version of your library you can run:

```
>>> from htmxl.compose.style import default_styles
>>> print([style['name'] for style in default_styles])
['xl-centered', 'xl-underlined']
```

And to use those styles, the classes would

```
<tr class="xl-centered">
  <td>c1</td>
  <td>c2</td>
</tr>
```

2.2.2 Combining styles together

```
<tr class="xl-centered xl-underlined">
  <td>c1</td>
  <td>c2</td>
</tr>
```

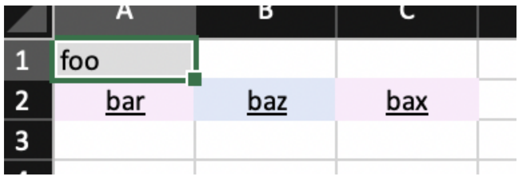
2.2.3 Define custom styles

```
from htmxl.compose import Workbook

template = """
    <body>
        <div>
            <span class="title">foo</span>
        </div>
        <div>
            {% for column_name in column_names %}
                <span class="xl-centered xl-underlined {{ loop.cycle('odd', 'even')}}
↪">{{ column_name }}</span>
            {% endfor %}
        </div>
    </body>
"""

styles = [
    {"name": "title", "pattern_fill": {"patternType": "solid", "fgColor": "DDDDDD"}},
    {"name": "odd", "pattern_fill": {"patternType": "solid", "fgColor": "FBEAFB"}},
    {"name": "even", "pattern_fill": {"patternType": "solid", "fgColor": "DFE7F8"}},
]


workbook = Workbook(styles=styles)
workbook.add_sheet_from_template(template=template, data=dict(column_names=["bar",
↪"baz", "bax"]))
workbook.compose("filename.xslx")
```



	A	B	C
1	foo		
2	bar	baz	bax
3			

2.3 Implementation Design

This information is not necessary to use this library, but might help contribution, and reduction of “magic”.

Libraries being used:

- **openpyxl**
  - Reads and writes excel files
  - Applies built in styles

    - Provides low and medium level APIs for doing excel manipulation

- **beautifulsoup4**
    - Parse raw HTML and provides iterables for writing out cells in excel
    - Provides access to tag level attribute and style notions
- **Jinja2**
    - Provides good python support for in-template flow control and variable access and templating
    - Could be replaced by any templating engine. It is only used to create raw HTML.

The writing implementation is built on few concepts:

- **A "cursor" or "write head" concept, implemenated in part by `htmxl.compose.write.Writer`.**
    - This cursor moves around a sheet, writing data and applying styles, etc.
    - **This cursor keeps "recordings" of where it has been within the context of writing a "tag".**
        - This permits nested tags each applying/overriding styles from their parents
        - This also allows for the implementation of cell merging
- **A htmxl.compose.write.write function**
    - Knowns how to interpret each HTML tag, or delegates to a specific tag implementation function
    - Is a recursive function, allowing deeply nested tags
- **Tag specific implementation functions**
    - Specify where the cursor should end up after the tag is finished being written.
    - Typically calls the generic htmxl.compose.write.write function on all its child elements.
- A mix of supported inline styles and single class based styling. While the underlying excel library supports "adding" styles together, the current implementation only permits one class per element.

CHAPTER

THREE

HOW TO'S

3.1 Writing Files Excel

3.1.1 Single Page

TODO

3.1.2 Multi Page

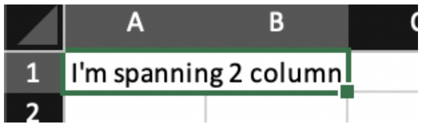
TODO

3.2 Merging Cells

Merging cells is performed in the template, and tries to match the html spec by making use of the existing `colspan` and `rowspan` attributes that you can set on elements like `<th>` to change how many visual cells they span across.

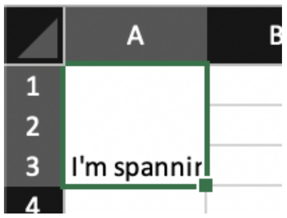
Note Currently merging cells are only supported in `<th>` elements.

3.2.1 Merging Cells Horizontally



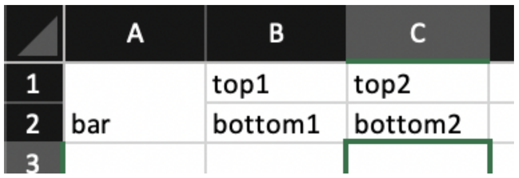
```
<table>
  <thead>
    <tr>
      <th colspan="2">I'm spanning 2 columns!</th>
    </tr>
  </thead>
</table>
```

3.2.2 Merging Cells Vertically



```
<table>
  <thead>
    <tr>
      <th rowspan="3">I'm spanning 3 rows!</th>
    </tr>
  </thead>
</table>
```

3.2.3 Creating a “container” row



```
<table>
  <thead>
    <tr>
      <th rowspan="2">bar</th>
      <tr>
        <th>top1</th>
        <th>top2</th>
      </tr>
      <tr>
        <td>bottom1</td>
        <td>bottom2</td>
      </tr>
    </tr>
  </thead>
</table>
```

CHAPTER FOUR

PARSER COMPATIBILITY

By default, the package will detect installed parser libraries and choose the first one we find, so a vanilla *Workbook()* instantiation should Just Work.

```
from htmxl.compose import Workbook

workbook = Workbook(parser='beautifulsoup')
workbook = Workbook(parser='lxml')
```

Importantly, the parsers will not exhibit identical behavior when handed the same templates. In general the `lxml` parser will be stricter and more prone to requiring "correct" HTML, while `beautifulsoup` is more permissive and will allow erroneous HTML. That tradeoff, however, generally leads to lxml being noticeably faster with large amounts of data.

While you can look to the specific libraries for a comprehensive set of behaviors, we can identify those which we've we've seen in the wild during the use of this library.

4.1 Closing Tags

Lxml requires closing tags to be explicitly closed.

This manifests itself in two obvious ways that might allow a template to succeed with the BeautifulSoup parser, while it might fail (raise an Exception) with lxml.

Consider

```
<div>
   <span>Foo
</div>
```

BeautifulSoup will happily allow the above template, while lxml will complain about the unclosed <*span*> block.

Also consider

```
<div>
   Foo
   <br>
   Bar
</div>
```

Self-closing tags, like `br` must **also** have the closing part, i..e `<br />` with lxml.

CHAPTER FIVE

API

5.1 htmxl

5.1.1 compose

workbook

write

style

Module dedicated to adding, finding, and applying styles.

constants

attributes

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX