generate-html

Release 1.0.0

Jasmijn Wellner

Aug 04, 2021

CONTENTS:

CHAPTER

ONE

TUTORIAL

generate-html is a library to help you generate dynamic HTML using Python. Let's start with a simple example:

```
from generate_html import fragment, tag, render_html

@fragment
def say_hello(who):
    yield 'Hello '
    yield tag.b(who)
    yield '!'

print(render_html(say_hello('World')))
```

This should print the following:

```
Hello <b>World</b>!
```

Hopefully that makes some sense to you.

It might seem strange to use `yield` in this way. The reason for that is that it allows us to use regular flow control to create dynamic HTML fragments and documents:

```
@fragment
def show_friends(friend_list):
    if friend_list:
        yield 'Your friends are:'
        for friend in friend_list:
            yield tag.button(friend)
    else:
        yield tag.i('You have no friends :(')
```

Then you can give this a list of functions:

```
print(render_html(show_friends(['Joey', 'Monica', 'Phoebe'])))
```

Which prints:

```
Your friends are:<button>Joey</button><button>Monica</button><button>Phoebe</button>
```

On the other hand, if you pass an empty list:

```
print(render_html(show_friends([])))
```

... you should get:

```
<i>You have no friends :(</i>
```

What if you want to have nested tags, though? `for` loops are statements, so you can't pass it as an argument.

For this reason, elements are also usable as context managers:

```
@fragment
def show_friends_2(friend_list):
    if friend_list:
        with tag.ul():
            for friend in friend_list:
                yield tag.li(friend)
    else:
        yield tag.i('You have no friends :(')
```

Now this:

```
print(render_html(show_friends_2(['Joey', 'Monica', 'Phoebe'])))
```

… will produce:

```
<ul><li>Joey</li><li>Monica</li><li>Phoebe</li></ul>
```

You can nest elements as much as you need:

```
@fragment
def okay():
    with tag.main(), tag.div(class_='wrapper'):
        yield tag.p('One')
        # You can even combine both types
        with tag.p('Abso-'):
            yield tag.em('fucking')
            yield '-lutely'

print(render_html(okay()))
```

```
<main><div class="wrapper"><p>One</p><p>Abso-<em>fucking</em>-lutely</p></div></main>
```

You don't need to use context managers in this case. The following produces equivalent output:

```
@fragment
def okay():
    yield tag.main(
        tag.div(
            tag.p('One'),
            tag.p('Abso-', tag.em('fucking'), '-lutely'),
            class_='wrapper'))
```

The advantage of using context managers mainly appears when you're doing more complicated things.

Notice how the tag constructor accepts keyword arguments. Those will be automatically converted into HTML attributes, using the rule that trailing underscores are removed, and all other underscores are converted into dashes. This rule also applies for tag names. For example:

```
@fragment
def weird():
    yield tag.import_(tag.fake_html(), class_='test', data_accept='y')

print(render_html(weird()))
```

Produces:

```
<import class="test" data-accept="y"><fake-html></fake-html></import>
```

If you wish to do more dynamic things with tags, you can also use `generate_html.create_element()`. `create_element('foo', children, attrs)` is equivalent to `tag.foo(*children, **attrs)`.

CHAPTER

TWO

API OVERVIEW

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search