Hykup Documentation

Quelklef

Mar 02, 2019

Contents:

CHAPTER 1

What is Hykup?

Hykup stands for Hy marKup. Hykup is a library for writing clean, DRY HTML without wanting to die. Hykup is designed to be small, clean, and composable; Hykup's power comes from the fact that it rests within the Python/Hy environment, allowing all of that to be used for HTML generation.

CHAPTER 2

Hykup in 2 Minutes

Import and require Hykup:

```
(require [hykup [*]])
(import [hykup [*]])
```

and write some (fancy) HTML:

```
(setv some-hykup #kup

  ; Components are separated from children with a '/'
  [div /

    ; Empty components should still have a '/'
    [span /]

    ; Classes are prefixed with a dot
    [p .large .red /]

    ; id and all other attributes are denoted as keyword arguments
    [p :id my-paragraph :style "background-color: red" /]

    ; Content can be written plainly, without quotes
    [p / I am a sentence!]

    ; Or with quotes, if needed (or wanted)
    [p / "I will (( mess up (( the Hy (( parsing"]])
```

and render it:

```
(setv rendered-html (.render some-hykup))
(print rendered-hyml)
; gives (actual output is not prettified)
; <div>
;   <span />
```

(continues on next page)

(continued from previous page)

```
;   <p class="large red">
;   <p id="my-paragraph" style="background-color: red" />
;   <p>I am a sentence!</p>
;   <p>I will (( mess up (( the Hy (( parsing</p>
; </div>
```

or write it to a file:

```
(render-file "some_file.html" some-hykup)
```

Hy values can be interpolated either as non-vector expressions or as symbols starting with ~:

```
(setv
  class "yummy"
  utensil "spoon"
  attribute-name "tastiness"
  attribute-val "100%")

(defn generate-class [] "filling")
(defn generate-attribute-name "poison-level")
(defn generate-attribute-val "0%")
(defn generate-food [] "alphabet soup")

#kup [p .meal
         .~class
         .(generate-class)

         :id dinnertime
         :~attribute-name ~attribute-val
         :(generate-attribute-name) (generate-attribute-val) /

         I eat (generate-food) with a ~utensil]

; <p class="meal yummy filling"
;    id="dinnertime"
;    tastiness="100%"
;    poison-level="0%">
;    I eat alphabet soup with a spoon
; </p>
```

Text spacing is designed to work intuitively and desirably, but fine control is possible:

```
#kup [p / "Use   strings   to   add   spaces" - "..." - "use a dash to surpress spaces
↪"}
; <p>Use   strings   to   add   spaces...use a dash to surpress spaces</p>
```

Boolean attributes need not be written out:

```
#kup [p :contenteditable / I am editable]  ; Allowed
#kup [p :contenteditable contenteditable / I am ediable]  ; Behaves exactly the same
```

Custom components can be defined:

```
(defn component-box [width chidlren &kwargs attributes]
  #kup [span :style (+ "width: " width ";"  ; use a positional argument
                       "height: " (.pop attributes "height")) ";"  ; or a keyword␣
↪argument
```

(continues on next page)

(continued from previous page)

```
              #** ~attributes /
              #* ~@children])

#kup [box 10px :height 20px / This text is in a box]
; <span style="width: 10px; height: 20px;">This text is in a box</span>
```

CHAPTER 3

Digging Deeper

More info on the Hykup API.

3.1 Creating Tags with Functions

Macros are great for end-users, but not fantastic for writing composable code. Hykup includes a `tag` function which allows for tags to be created from a function rather than a macro.

Its signature is

```
(defn tag [tag-name &rest children &kwargs properties] ...)
```

The first argument is the tag name, which will be converted to a string with `str`. Positonal arguments will become the children to tag; they will be passed along unchanged. Keyword arguments will become the properties. They keywords themselves will automatically be unmangled. Both the keywords and the values will be converted to a string with `str` when the tag is rendered.

The `tag` function has no spacing rules. Spacing is preserved exactly as it was given.

A sample tag creation looks like:

```
(tag 'p :id 'my-fancy-p-tag
        :class "very-large very-small"
        (tag 'span "Text nodes")
        " are passed as strings.")
; <p id="my-fancy-p-tag" class="very-large very-small">
;   <span>Text nodes</span> are passed as strings</p>
```

3.2 Creating Custom Tags (Components)

Hykup would be close to useless if you weren't able to create your own components. Luckily, you can!

Components are actually just functions. They accept a number of positional arguments, the last of which is the children to the component, as well as keyword arguments (the tag attributes). Classes are passed in as the `class` keyword, but this is likely to change in the future.

The functions must be named `component-` and then the name of the component. For instance, the component for the `<p>` tag is named `component-p`.

For examples:

```
; We'll define a "quote" component which adds fancy quotes before and after
; the inner text.
; We'll also accept attributes which will be placed on the tag.
(defn component-quote [children &kwargs attributes]
  ; (Dashes surpress whitespace)
  #kup [span #** ~properties / “ - ~@children - ”])

(.render #kup [quote :style "color: red" / I am quoted!])
; <span style="color: red">“I am quoted!”</span>

; But perhaps fancy quotes don't always work, and we want to
; be able to choose between normal and fancy quotes?
(defn component-quote [fancy? children &kwargs attributes]
  (if fancy?
    #kup [span #** ~attributes / “ - ~@children - ”]
    #kup [span #** ~attributes / " - ~@children - "]))

(.render #kup [quote ~false  ; Must unqote otherwise the function would be passed the
                     ; string "flase" rather than the value `false`.
             I have normal quotes])
; <span>"I have normal quotes"</span>

; Or maybe we want the caller to be able to supply their own
; start and end quotes, with defaults?
(defn component-quote [children
                       &kwonly [start-quote "“"] [end-quote "”"]
                       &kwargs attributes]
  #kup [span / ~start-quote - ~@children - ~end-quote]

(.render #kup [quote :start-quote [[ :end-quote ]] / I have custom quotes])
; <span>[[I have custom quotes]]</span>
```

3.3 Spacing Rules

If a Hykup form is supplied such as

```
#kup [p / I am some text]
```

There need to be some rules to decide how the symbols `I am some text` should be rendered. In this particular example, it's simple: a space is placed between each symbol, rendering

```
<p>I am some text</p>
```

If children are nested elements or interpolated expressions, they are still surrounded with spaces:

```
(setv pronoun "I")
#kup [p / ~pronoun am [em / emphasized]]
; <p>I am <em>emphasized</em></em>
```

Strings allow for exact control of spaces:

```
#kup [p / "one two  three   four    "]
; <p>one two  three   four    </p>
```

but are still surrounded if placed next to other children:

```
#kup [p / "left" "middle" right]
; <p>left middle right</p>
```

Spaces may be explicitly surpressed with `-`:

```
#kup [p / "no" - "spaces" - "please"]
; </p>nospacesplease</p>
```

If text (symbol or string) starts with any of `, . ; : ) ] } ? !`, a space will not implicitly be placed before it:

```
; Note that the comma after the em is parsed as its own symbol
#kup [p / well, well, [em / well], what do we have here?]
; <p>well, well, <em>well</em>, what do we have here?</p>
```

Similarly, if text (symbol or string) ends with any of `( [ { ~ #`, a space will not implicitly be placed after it:

```
#kup [p / I own "~" (how-many-hamsters?) hamsters]
; <p>I own ~14 hamsters</p>
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search