hyperpython Documentation

Fábio Macêdo Mendes

Sep 15, 2018

Contents

Warning: Beta software You are using a software that has not reached a stable version yet. Please beware that interfaces might change, APIs might disappear and general breakage can occur before *1.0*.

If you plan to use this software for something important, please read the roadmap, and the issue tracker in Github. If you are unsure about the future of this project, please talk to the developers, or (better yet) get involved with the development of Hyperpython!

Hyperpython is a library for creating web components by embedding a HTML in a Python sub-language.

CHAPTER 1

Overview

1.1 Hyperpython

Hyperpython is a Python interpretation of Hyperscript. If you are not familiar with Hyperscript, think of it as a pure Javascript alternative to JSX. Hyperpython allow us to generate, manipulate and query HTML documents using a small DSL embedded in Python. Just like Hyperscript, the default entry point is the *hyperpython.h()* function:

```
>>> from hyperpython import h
>>> elem = h('p', {'class': 'hello'}, ['Hello World!'])
```

This can be converted to HTML by calling `str()` on the element:

```
>>> print(elem)
<p class="hello">Hello World!</p>
```

It accepts Hyperscript's `h(tag, attributes, list_of_children)` signature, but we encourage to use more idiomatic Python version that uses keyword arguments to represent attributes instead of a dictionary. If the list of children contains a single element, we can also omit the brackets:

```
>>> h('p', 'Hello World!', class_='hello') == elem
True
```

Notice in the snippet above that we had to escape the "class" attribute because it is a reserved word in Python. Hyperpython maps Python keyword arguments by replacing underscores with dashes and by escaping Python reserved words such as "class", "for" "from", etc with a trailing underscore.

Elements can be be more conveniently created with standalone functions representing specific tags:

```
>>> print(p('Hello World!', class_='hello'))
<p class="hello">Hello World!</p>
```

In Python, keyword arguments cannot appear after positional arguments. This means that attributes are placed *after* the list of children, which isn't natural to represent HTML. For simple elements like the ones above, it does not hinders legibility, but for larger structures it can be a real issue. Hyperpython provides two alternatives. The first uses the index notation:

```
>>> from hyperpython import div, p, h1
>>> fragment = \
...     div(class_="alert-box")[
...         h1('Hello Python'),
...         p('Now you can write HTML in Python!'),
...     ]
```

The second alternative is to use the "children" pseudo-attribute. This is the approach taken by some Javascript libraries such as React:

```
>>> fragment = \
...     div(class_="alert-box",
...         children = [
...             h1('Hello Python'),
...             p('Now you can write HTML in Python!'),
...         ])
```

Hyperpython returns a DOM-like structure which we can introspect, query and modify later. The main usage, of course, is to render strings of HTML source code. We expect that the main use of Hyperpython will be to complement (or even replace) the templating language in a Python web application. That said, Hyperpython generates a very compact HTML that is efficient to generate and transmit over the wire. To get a more human-friendly output (and keep your sanity while debugging), use the .pretty() method:

```
>>> print(fragment.pretty())
<div class="alert-box">
  <h1>Hello Python</h1>
  <p>Now you can write HTML in Python!</p>
</div>
```

1.2 Replacing templates

The goal of `hyperpython` is to replace a lot of work that would be traditionally done with a template engine such as Jinja2 by Python code that generates HTML fragments. Templating languages are obviously more expressive than pure Python for string interpolation, and are a perfect match for ad hoc documents. For large systems, they offer little in terms of architecture, organization and code reuse.

A recent trend in Javascript is to promote direct creation of DOM or virtual DOM nodes sidestepping the whole business of rendering intermediate HTML strings. React was probably the library that better popularized this idea. As they nicely put, "Templates separate technologies, not concerns". There is no point on choosing a deliberately underpowered programming language that integrates poorly with your data sources just to output structured documents in a flat string representation.

The same lesson can be applied to Python on the server side. With Hyperpython, we can generate HTML directly in Python. Hyperpython plays nicely with existing templating systems, but it makes easy to migrate parts of your rendering sub-system to Python.

For those afraid of putting too much logic on templates, we recognize that Hyperpython doesn't prevent anyone from shooting themselves on the foot, but neither do any minimally powerful templating engine. It always requires some discipline to keep business logic separated from view logic. Our advice is to break code in small pieces and compose those pieces in simple and predictable ways. Incidentally, this is a good advice for any piece of code ;).

1.3 Can it be used on Django? Flask? Etc?

Of course! Hyperpython is completely framework agnostic. We have a few optional integrations with Django, but it does not prevent Hyperpython of being used in other frameworks or without any framework at all. It implements the __html__ interface which is recognized by most templating engines in Python. That way, it is possible to pass Hyperpython fragments to existing templates in Django, Jinja2 and others.

CHAPTER 2

Installation instructions

Hyperpython can be installed using pip:

```
$ python -m pip install hyperpython
```

This command will fetch the archive and its dependencies from the internet and install them.

If you've downloaded the tarball, unpack it, and execute:

```
$ python setup.py install --user
```

You might prefer to install it system-wide. In this case, skip the `--user` option and execute as superuser by prepending the command with `sudo`.

2.1 Troubleshoot

Hyperpython requires Python 3.6+. Make sure you use an updated distribution.

Windows users may find that these command will only works if typed from Python's installation directory.

Some Linux distributions (e.g. Ubuntu) install Python without installing pip. Please install it before. If you don't have root privileges, download the get-pip.py script at https://bootstrap.pypa.io/get-pip.py and execute it as `python get-pip.py --user`.

CHAPTER 3

Tutorial

3.1 A simple example

Suppose we want to implement a little Bootstrap element that shows a menu with actions (this is a random example taken from Bootstrap website).

```
<div class="btn-group">
  <button type="button"
          class="btn btn-default dropdown-toggle"
          data-toggle="dropdown"
          aria-haspopup="true"
          aria-expanded="false">
    Action <span class="caret"></span>
  </button>
  <ul class="dropdown-menu">
    <li><a href="#">Action</a></li>
    <li><a href="#">Another action</a></li>
    <li><a href="#">Something else here</a></li>
    <li role="separator" class="divider"></li>
    <li><a href="#">Separated link</a></li>
  </ul>
</div>
```

Of course we could translate this directly into Hyperpython by calling the corresponding `div()`, `button()`, etc functions. But first, let us break up this mess into smaller pieces.

```
from hyperpython import button, div, p, ul, li, span, a, classes

def menu_button(name, caret=True, class_=None, **kwargs):
    if caret:
        children = [name, ' ', span('caret')]
    else:
        children = [name]
```

(continues on next page)

(continued from previous page)

```
    return \
        button(
            type='button',
            class_=['btn', 'btn-default', 'dropdown-toggle', *classes(class_)],
            data_toggle="dropdown",
            aria_haspopup="true",
            aria_expanded="false",
            children=children,
            **kwargs
        )
```

It might look like it's a lot of trouble for a simple component. But now we can reuse this piece easily instead of writing a similar code from scratch every time a new button is necessary: `menu_button('File')`, `menu_button('Edit')`, `....` The next step is to create a function that takes a list of strings and return the corresponding menu (in the real world we might also want to control the href attribute). We are also going to be clever and use Ellipsis (`...`) as a menu separator.

```
def menu_data(values):
    def do_item(x):
        if x is ...:
            return li(role='separator', class_='divider')
        else:
            # This could parse the href from string, or take a tuple
            # input, or whatever you like. The hyperpython.components.hyperlink
            # function can be handy here.
            return li(a(x, href='#'))
    return ul(map(do_item, values), class_='dropdown-menu')
```

Now we glue both together...

```
def menu(name, values, caret=True):
    return \
        div(class_='btn-group')[
            menu_button(name, caret=True),
            menu_data(values),
        ]
```

... and create as many new menu buttons as we like:

```
menubar = \
    div(id='menubar')[
        menu('File', ['New', 'Open', ..., 'Exit']),
        menu('Edit', ['Copy', 'Paste', ..., 'Preferences']),
        menu('Help', ['Manual', 'Topics', ..., 'About']),
    ]
```

Look how nice it is now :)

3.2 How does it work?

Hyperpython syntax is just regular Python wrapped in a HTML-wannabe DSL. How does it work?

Take the example:

```
element = \
    div(class_="contact-card")[
        span("john", class_="contact-name"),
        span("555-1234", class_="contact-phone"),
    ]
```

In Hyperpython, we can declare attributes as keyword arguments and children as a index access. This clever abuse of Python syntax is good for creating expressive representations of HTML documents. Under the hood, Python calls div() and generates an *Element* instance. Indexing is used to insert the given elements as children and then return the tag itself as a result. We encourage using this syntax only during element creation in order to avoid confusion.

Tag functions also accept a few alternative signatures:

h1('title'): First positional argument can be a single child, string or list of children. This generates `<h1>title</h1>`.

h1({'class': 'foo'}, 'title'): If the first argument is a dictionary, it is interpreted as attributes. Notice that when passed this way, attribute names are not modified. This generates `<h1 class="foo">title</h1>`.

h1('title', class_='foo', data_foo=True): Keyword arguments receive a special treatment: trailing underscores are removed from names that conflict with Python keywords and underscores in the middle of the word are converted to dashes. This generates `<h1 class="foo" data-foo>title</h1>`.

h1(class_='foo', children=['title']): Children can also be passed as a keyword argument. This generates `<h1 class="foo">title</h1>`.

In HTML, all tag attributes are all stringly typed. This is far from ideal and can be easily fixed since we are representing HTML from a typed language. Hyperpython does the following coercions when interpreting attributes:

"class" attribute: Hyperpython expects a list of strings. If a single string is given, it is split into several classes and saved as a list. It has a similar semantics as the classList attribute in the DOM. The list of classes can also be passed as a dictionary. In that case, it includes all keys associated to a truthy value.

boolean attributes: A value of False or None for an attribute means that it should be omitted from generated HTML. A value of True renders the attribute without the associated value.

3.2.1 Imperative interface

We encourage users to adopt the declarative API and generally treat tags as immutable structures. Hyperpython does not enforce immutability and actually offers some APIs to change data structures inplace. Once a tag is created, it is possible to change it's attributes dictionary and list of children. There are also a few methods designed to manipulate Hyperpython data structures.

```
>>> elem = div('foo', class_='elem')
>>> elem.add_child('bar')
h('div', {'class': ['elem']}, ['foo', 'bar'])
```

Attributes are also exposed in the .attrs dictionary:

```
>>> elem.attrs['data-answer'] = 42
>>> elem.attrs.keys()
dict_keys(['class', 'data-answer'])
```

The "class" and "id" attributes are also exposed directly from the tag object since they are used so often:

```
>>> elem = div('foo', class_='class', id='id')
>>> elem.id, elem.classes
('id', ['class'])
```

Classes can be manipulated directly, but it is safer to use the `elem.add_class()` and `elem.set_class()` methods, since they understand all the different ways Hyperpython uses to specify a list of classes.

```
>>> elem.add_class('bar baz')
h(...)
>>> print(elem)
<div class="class bar baz" id="id">foo</div>
```

Notice that add_class() returns the changed element and hence can be used in a fluid API style.

CHAPTER 4

API Reference

API documentation for the Hyperpython module.

4.1 Basic types

class hyperpython.**Element**(*tag: str*, *attrs: dict*, *children: list*, *is_void=False*, *requires=()*)
Represents an HTML element.

add_child(*value*)
Add child element to data structure.

Caveat: Hyperpython *do not* enforce immutability, but it is a good practice to keep HTML data structures immutable.

add_class(*cls*, *first=False*)
Add class or group of classes to the class list.

copy()
Return a copy of object.

dump(*file*)
Dumps HTML data into file.

json()
JSON-compatible representation of object.

pretty(***kwargs*)
Render a pretty printed HTML.

This method is less efficient than .render(), but is useful for debugging

render()
Renders object as string.

set_class(*cls=()*)
Replace all current classes by the new ones.

walk()
Walk over all elements in the object tree, including Elements and Text fragments.

walk_tags()
Walk over all elements in the object tree, excluding Text fragments.

class hyperpython.**Text**(*data*, *escape=None*)
It extends the Markup object with a Element-compatible API.

dump(*file*)
Dump contents of element in the given file.

render()
Renders object as string.

unescaped
Convert all named and numeric character references (e.g. >, >, &x3e;) in the string s to the corresponding unicode characters. This function uses the rules defined by the HTML 5 standard for both valid and invalid character references, and the list of HTML 5 named character references defined in html.entities.html5.

class hyperpython.**Block**(*children*, *requires=()*)
Represents a list of elements *not* wrapped in a tag.

dump(*file*)
Dump contents of element in the given file.

4.1.1 Functions

The generic entry point is the *h()* function. It also has functions with same names of all HTML tags.

hyperpython.**h**(*tag*, **args*, *children=None*, ***attrs*)
Creates a tag.

It has many different signatures:

h('h1', 'content') Content can be a string, a child node or a list of children.

h('h1', {'class': 'title'}, 'content') If the second argument is a dictionary, it is interpreted as tag attributes.

h('h1', 'title', class_='title') Keyword arguments are also interpreted a attributes. The h function makes a few changes: underscores are converted to dashes and trailing underscores after Python keywords such as `class_`, `for_`, etc are ignored.

h('h1', class_='title')['content'] Children can also be specified using squared brackets. It understands strings, other tags, and lists of tags.

h('h1', class_='title', children=['content']) Optionally, the list of children nodes can be specified as a keyword argument.

4.2 Creation of HTML elements for Python objects

hyperpython.**html**(*obj*, *role=None*, ***kwargs*)
Convert object into a hyperpython structure.

Parameters

- **obj** – Input object.
- **role** – Optional description

- **context variables for the rendering process can be passed as** (*Additional*) –
- **arguments.** (*keyword*) –

Returns A hyperpython object.

hyperpython.**render**(*obj*, *role=None*, ***kwargs*)
Like *render()*, but return a string of HTML code instead of a Hyperpython object.

hyperpython.**fragment**(*path*, ***kwargs*)
Compute the Hyperpython element for the given fragment path.

Parameters **path** (*str*) – Argument path.

Examples

```
>>> fragment('page.header', user='me!')
h('header', ['Hello me!'])
```

4.3 Hyperpython components

All functions bellow belongs to the *hyperpython.components* module.

4.3.1 Hyperlinks

hyperpython.components.**hyperlink**(*obj*, *href=None*, ***attrs*) → hyperpython.core.Element
Converts object to an anchor (<a>) tags.

It implements some common use cases:

str: Renders string as content inside the <a>... tags. Additional options including href can be passed as keyword arguments. If no href is given, it tries to parse a string of "Value <link>" and uses href='#' if no link is found.

dict or mapping: Most keys are interpreted as attributes. The visible content of the link must be stored in the 'content' key:

```
>>> hyperlink({'href': 'www.python.com', 'content': 'Python'})
<a href="www.python.com">Python</a>
```

django model: It must define a get_absolute_url() method. This function uses this result as the href field and str(model) as its content.

```
>>> from django.contrib.auth import get_user_model
>>> user_model = get_user_model()
>>> hyperlink(user_model(first_name='Joe', username='joe123'))
<a href="/users/joe123">Joe</a>
```

In order to support other types, use the lazy_singledispatch mechanism:

```
@hyperlink.registerPlugin(MyFancyType)
def _(x, **kwargs):
    return safe(render_object_as_safe_html(x))
```

See also:

hyperpython.helpers.attrs(): See this function for an exact explanation of how keyword arguments are translated into HTML attributes.

hyperpython.components.**url**(*obj*)
Returns a url for the given object.

hyperpython.components.**a_or_p**(**args*, *href=None*, ***kwargs*)
Return a or p tag depending if href is defined or not.

hyperpython.components.**a_or_span**(**args*, *href=None*, ***kwargs*)
Return a or span tag depending if href is defined or not.

hyperpython.components.**breadcrumbs**(*links*, *class_='breadcrumbs'*)
Component that Receives a list of links and return a breadcrumbs element.

4.3.2 HTML data structures

Those functions convert Python data structures to their natural HTML representations.

hyperpython.components.**html_list**(*data*, *role=None*, *ordered=False*, ***kwargs*)
Convert a Python iterable into an HTML list element.

Parameters

- **data** – Sequence data.
- **ordered** – If True, returns an ordered list (<ol>) element.
- **role** – Role passed to render each item in the sequence.
- **keyword arguments are passed to the root element.** (*Additional*) –

Examples

```
>>> doc = html_list([1, 2, 3])
>>> print(doc.pretty())
<ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>'
```

hyperpython.components.**html_map**(*data*, *role=None*, *key_role=None*, ***kwargs*)
Renders mapping as a description list using dt as keys and dt as values.

Parameters

- **data** – Sequence data.
- **role** – Role passed to the dd (values) elements of the description list.
- **key_role** – Role passed to the dt (keys) elements of the description list.
- **keyword arguments are passed to the root element.** (*Additional*) –

Examples

```
>>> doc = html_map({'answer': 42, 'universe': True})
>>> print(doc.pretty())
<dl>
    <dt>answer</dt>
    <dd>42</dd>
    <dt>universe</dt>
    <dd>True</dd>
</dl>
```

hyperpython.components.**html_table**(*data*, *, *role=None*, *columns=None*, ***kwargs*)

Convert 2D matrix-like data to an HTML table.

Parameters

- **data** – Sequence data.
- **role** – Role used to render elements of the table.
- **columns** – A list of column names to be added as <thead>.
- **keyword arguments are passed to the root element.** (*Additional*) –

Examples

```
>>> doc = html_table([[1, 2], [3, 4]], columns=['a', 'b'])
>>> print(doc.pretty())
<table>
    <thead>
        <tr><th>a</th><th>b</th></tr>
    </thead>
    <tbody>
        <tr><td>1</td><td>2</td></tr>
        <tr><td>3</td><td>4</td></tr>
    </tbody>
</table>
```

4.3.3 Icons

Generic icon support using the <i> tag and helper functions for Font Awesome icons.

hyperpython.components.**icon**(*name*, *href=None*, *icon_class=<function <lambda>>*, *icon_data=<function <lambda>>*, ***kwargs*)

Returns a icon tag.

Parameters

- **name** (*str*) – Name of the icon.
- **href** (*str*) – If given, it wraps the results into a anchor link.
- **icon_class** (*callable*) – A function that maps an icon name into the corresponding class or list of classes that should be added to the icon.

hyperpython.components.**fa_icon**(*name*, *href=None*, *collection=None*, ***kwargs*)

Font awesome icon.

Parameters

- **name** (*str*) – Name of the icon.
- **href** (*str*) – If given, it wraps the results into a anchor link.
- **collection** (*'fa', 'fab', 'far', 'fal', 'fas'*) –

  The font-awesome collection:

  – fa/far/regular: regular icons

  – fab/brand: brand icons

  – fal/light: light icons

  – fas/solid: solid icons

Examples

```
>>> fa_icon('face')
<i class="fa fa-face"></i>
```

4.3.4 Text

hyperpython.components.**markdown**(*text*, *, *output_format='html5'*, ***kwargs*)

Renders Markdown content as HTML and return as a safe string.

CHAPTER 5

Django configuration

Folks in Django like to add lines on the list of installed apps:

```
# settings.py

INSTALLED_APPS = [
    'hyperpython.django',
    ...
]
```

It works, but for now we do not do anything ;). In the future this line may register new template tags and other integrations, but for now it is only possible to avoid breaking expectations people have about "Django integration".

5.1 Template integration

Hyperpython elements can be directly used in both Django and Jinja2 templates without any additional configuration: just wrap them with *{{* and *}}*‘ and we are good to go.

5.2 Going further

Hyperpython can have also have a more important architectural role in a traditional Django project with server side rendering. In Django, we are used to split our project into pages with view functions mapping each "request" into the appropriate "response". For complex layouts, this is often too much work in the hands of a single function for it has to organize the rendering of lots of smaller pieces that probably requires coordination of too many different responsibilities.

In practice, it is easy to fall into the anti-pattern of fat views + fat templates. A common alternative to solve some of those problems is to move most of the logic to the model, which creates a similarly bad problems with fat models and does not touch the problem with the templates. (Some people advocate for fat models since they provide better organization, testability and code reuse. While this is probably true, it can also promote bad usage patterns of Django's ORM since often some logic that now unnecessary lives on the model might require inefficient instantiation of objects

instead of direct manipulation of querysets. In its worst incarnation, it might create a case of the very hard to spot n + 1 problem in your codebase).

Hyperpython can help cutting cruft from the view functions by providing natural ways of splitting them into smaller reusable pieces.

5.2.1 Hyperpython roles

A Django view essentially maps an HTTP request (url + headers + some optional data) to the corresponding response (data + headers), which is generally processed by a templating engine. The view function is responsible for rendering a page.

Similarly, a Hyperpython role function receives an object and a role and return an Hyperpython rendering of it. The point is to isolate functionality that can be easily reused across different views. Even in cases where functionality is used in a single page, it can be useful to reduce complexity by splitting rendering in smaller and more focused parts.

Hyperpython stores a global mapping from objects and roles to their corresponding renderers. This is declared using the `register` decorator method of *`hyperpython.html()`* passing a type and a role:

```
from hyperpython import html, Text


@html.register(int)
def integer_fallback(x):
    """
    Fallback renderer for integers.
    """
    # This is not really necessary since the fallback renderer already
    # uses str() to create an HTML representation of an object.
    return Text(str(x))


@html.register(int, 'currency')
def dollars(x):
    """
    Represents a number as currency
    """
    return Text(f'U$ {x:.2f}')


@html.register(int, 'roman')
def transcribe(x):
    """
    Writes down the number as roman numeral.

    Works for numbers between 1 and 10.
    """
    numbers = ['I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII', 'IX', 'X']
    return Text(numbers[x - 1])
```

Now we can render integers using one of the specified roles:

```
>>> html(7, 'currency')
Text('U$ 7.00')
```

```
>>> html(7, 'roman')
Text('VII')
```

Role renderers can accept optional keyword arguments that may influence how the final result is generated. A better implementation of currency, for instance, could accept:

```
>>> html(7, 'currency', country='Brazil')
R$ 7,00
```

This is done by simply accepting additional keyword arguments in the function definition.

Keep in mind that each renderer is associated to both a type and a role. The functions above does not handle floats, for instance:

```
>>> html(7.0, 'currency')
Traceback (most recent call last):
...
TypeError: no "currency" role registered for float objects
```

Keeping that in mind, always consider using abstract types such as `types.Number` and `collections.abc.Sequence`.

5.2.2 Roles in templates

Role renderers are globally available in Python code and can also be made available inside templates. The exact configuration depends on your template engine.

Jinja2

You must register the `hyperpython.jinja2.filters.role()` filter in your Jinja2 environment. Now just use it to filter any variable:

```
{{ user|role('contact-info-card', favorite=True) }}
```

This will be translated into `html(user, 'contact-info-card', favorite=True)`.

Django

Not available yet, but PRs are welcome :)

5.2.3 Registered roles

Hyperpython has some builtin roles registered to common Python objects.

For now, the guideline is "read the code". (You can also contribute with documentation).

5.2.4 Sequences and Querysets

`html()` uses a type/role based dispatch. This means that objects that share the same type are not handled properly, which is precisely the case of lists Django and querysets.

Generally speaking, queryset instances are all of the same type `django.db.QuerySet`, even for queries resulting from different models. Hence, queryset renderers are not associated with models and cannot express useful constraints such as a renderer for a "queryset of users".

5.3 Fragments

`html()` solves the problem of "how render an object in some specific context". Sometimes, we do not have an object that can be naturally associated with an HTML fragment. For this, Hyperpython uses the `fragment()` function that instead associates a string path to some HTML structure. This is very useful to declare generic page elements such as headers, footers, etc:

```
from hyperpython import fragment, header, p

@fragment.register('page.header')
def render_header():
    return header('Minimalistic site header')
```

Now we render it using the `fragment()` function:

```
>>> fragment('page.header')
h('header', 'Minimalistic site header')
```

Those string paths can be parametrized and work very similarly to URLs in frameworks like Django or Flask.

```
@fragment.register('count-<int:n>')
def counter(n):
    # n is computed from the path given to the fragment function.
    return p(f'counting to {n}')
```

Fragments can be rendered using

```
>>> fragment('count-42')
h('p', 'counting to 42')
```

Beware to avoid pointless usage of path arguments (just like in the example above).:func:*fragment* accepts optional keyword arguments that are passed unchanged to the implementation and most of the time any extra parameter should be treated as keyword arguments instead of a location on the path.

```
@fragment.register('count')
def better_counter(n):
    return p(f'counting to {n}')
```

```
>>> fragment('count', n=42)
h('p', 'counting to 42')
```

CHAPTER 6

Frequently asked questions

6.1 Dependencies

6.1.1 Will it ever support Python 2.7?

No. Python 2.7 will meet its end-of-life in 2020, and all projects that use it should be planning on moving to Python 3.

6.1.2 Will it ever support Python 3.5 or lower?

Probably not. Hyperpython uses a functional programming library called Sidekick which requires Python 3.6+. It is not impossible to port this library to 3.5, but it is a very low priority for the developers. Of course, you can make this happen by sending pull requests ;)

CHAPTER 7

HyperJSON (WIP)

With the popularization of Virtual-DOM based Javascript frameworks, it has become increasingly problematic to retrieve unstructured blobs of HTML that overrides the innerHTML attribute of some element in the DOM. Hyperpython uses a specific JSON serialization format to represent HTML data structures that can be used as an alternative to transmit HTML from the back-end to the front-end. This plays more nicely with technologies such as React, ELM or Vue.js. Moreover, it provides the basis of a system to represent HTML templates in the client that not based on string interpolation.

7.1 Tags as data structures

DOM means "Document Object Model". In informal parlance we use it to refer to the "structure of our HTML document. In reality it describes much more: it describes the API and the inheritance chain of how all HTML elements are described in an object oriented fashion. We explicitly want to avoid that point of view and focus solely on structure. Hyperpython is not concerned with the DOM, but rather with the structure. The DOM is only relevant to the browser.

HyperJSON describes this nested HTML data structure using S-expressions. If you are not familiar with S-expressions, it is a very nice idea popularized by LISP to encode tree-like structures as nested lists. Our use-case is very simple, since all tags become 3 element S-expressions:

> tag ==> [<tag-name>, <attributes>, <children>]

The first element is a string with the tag name, the second is an object mapping attributes to their values and the third is a list of children nodes. Both the second and the third arguments can be optionally omitted. The list of children is formed by strings and other tag S-expressions.

An example is handy:

```
<div class="foo">
    <h1>Title</h1>
    <p>Hello world!</p>
</div>
```

Becomes:

```
["div", {"class": "foo"}, [
    ["h1", "Title"],
    ["p", "Hello World!"]
]
```

This is about the same size as the original HTML and represents the same data structure.

7.2 Formalism

This informal description is not good for language lawyers. Consider the elements in http://json.org to specify HyperJSON with the BNF notation:

```
element    : '[' name ',' attributes ','  children ']'
           | '[' name ',' attributes ']'
           | '[' name ',' children ']'

name       : string

attributes : object

children   : '[' items ']'
           | string

items      : item
           | item ',' items

item       : string
           | tag
```

7.2.1 Semantics

The attributes object does not support arbitrary values. Unless noted, all attribute values must be strings. There are a few exceptions, though:

class: Class is a list of classes.

7.3 Converting from/to JSON

Hyperpython structures can be easily serialized as JSON or restored from HyperJSON data. Use the `Element.to_json()` method or the `from_json()` function to convert from one format to the other.

7.4 Frontend integration

There are only two options for now: a vanilla Javascript library that spits DOM elements and an ELM library that produces nodes for ELM's virtual DOM.

7.5 Extensions and templates

We described the basic HyperJSON language. The client can optionally support HyperJSON templates that makes easy to interpolate values for easy generation of dynamic content on the client.

CHAPTER 8

Roadmap

8.1 Far future and crazy ideas

- **:feature:‘1‘** Integrate with PScript to transpile Hyperpython functions to Javascript components. Study solutions for Hyperscript, Mithrill, and React.

8.2 Near future backlog

- **:feature:‘2‘** Better django integration #2.
- Extend HyperJSON with basic templating capabilities.

8.3 Version 1.1

- Achieve 95% test code coverage.
- Zero issues in PyCharm.
- Stabilize HyperJSON format and integrate with Hyperpython.

8.4 Version 1.0

- Achieve 90% test code coverage.
- Basic Hyperpython API.
- Registering renderers and fragments.
- Functions for all HTML tags.

- Safe interpolation of strings and other objects.
- Register project in free CI services (Travis, Codecov, Code climate).
- Documentation in Read the Docs.

CHAPTER 9

License

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index