dhtmlparser3

Release 3.0.17

unknown

Jan 28, 2023

CONTENTS

dhtmlparser3 is a lightweight HTML/XML parser created for one purpose - quick and easy picking selected tags from DOM.

It can be very useful when you are in need to write own "guerilla" API for some webpage, or a scrapper.

If you want, you can also create HTML/XML documents more easily than by joining strings.

The usage is super simple, and allows you to do things like pattern matching in the HTML tree.

CHAPTER

ONE

SOURCES

Source codes can be found here; https://github.com/Bystroushaak/dhtmlparser3

CHAPTER

TWO

INSTALLATION

dhtmlparser3 is hosted at pypi, so you can install it using pip:

```
pip3 install dhtmlparser3
```

CHAPTER

THREE

QUICK INTRODUCTION

3.1 Parse DOM

The first thing you'll need to do in order to work with the HTML code is parse it into the DOM.

To do that, import the module and then call a *parse()* on the text you want to parse.

```
import dhtmlparser3

example_html = """<html>
<head><title>Title</title></head>
<body>
<h1>HTML</h1>
<p>Some content. <a href="https://blog.rfox.eu">Link</a>.</p>
</body>
</html>"""

dom = dhtmlparser3.parse(example_html)
```

Now you have everything parsed in the tree structure of the *Tag* objects.

3.2 Find links

Let's say that you want all links in the HTML. There is a handy method *Tag.find()*, which does exactly that:

```
>>> dom.find("a")
[Tag('a', parameters=SpecialDict([('href', 'https://blog.rfox.eu')]), nonpair=False)]
```

The parameters are as follows:

1. String name of the tag you want to find.
2. Dict with the parameters, to specify for example, class name.
3. (Lambda) function, which takes the tag as a parameter and should return `True` if it matches.

You can get the link from the tag using *parameters* property:

```
>>> a = dom.find("a")[0]
>>> a.parameters["href"]
'https://blog.rfox.eu'
```

The second important property is *content*, where the content of the tag is stored.

```
>>> a.content
['Link']
```

content property consists of either *Tag*, *Comment*, or `str` objects. This is because the parser is trying to keep all of the information to be able to restore whitespace-perfect original representation of the HTML.

But in general, if you want a string from the content, call *content_str()*:

```
>>> a.content_str()
'Link'
```

Let's see the same example with the <p> tag:

```
>>> dom.find("p")[0].content_str()
'Some content. <a href="https://blog.rfox.eu">Link</a>.'
```

If you want the content without tags, you can call *content_without_tags()*:

```
>>> dom.find("p")[0].content_without_tags()
'Some content. Link.'
```

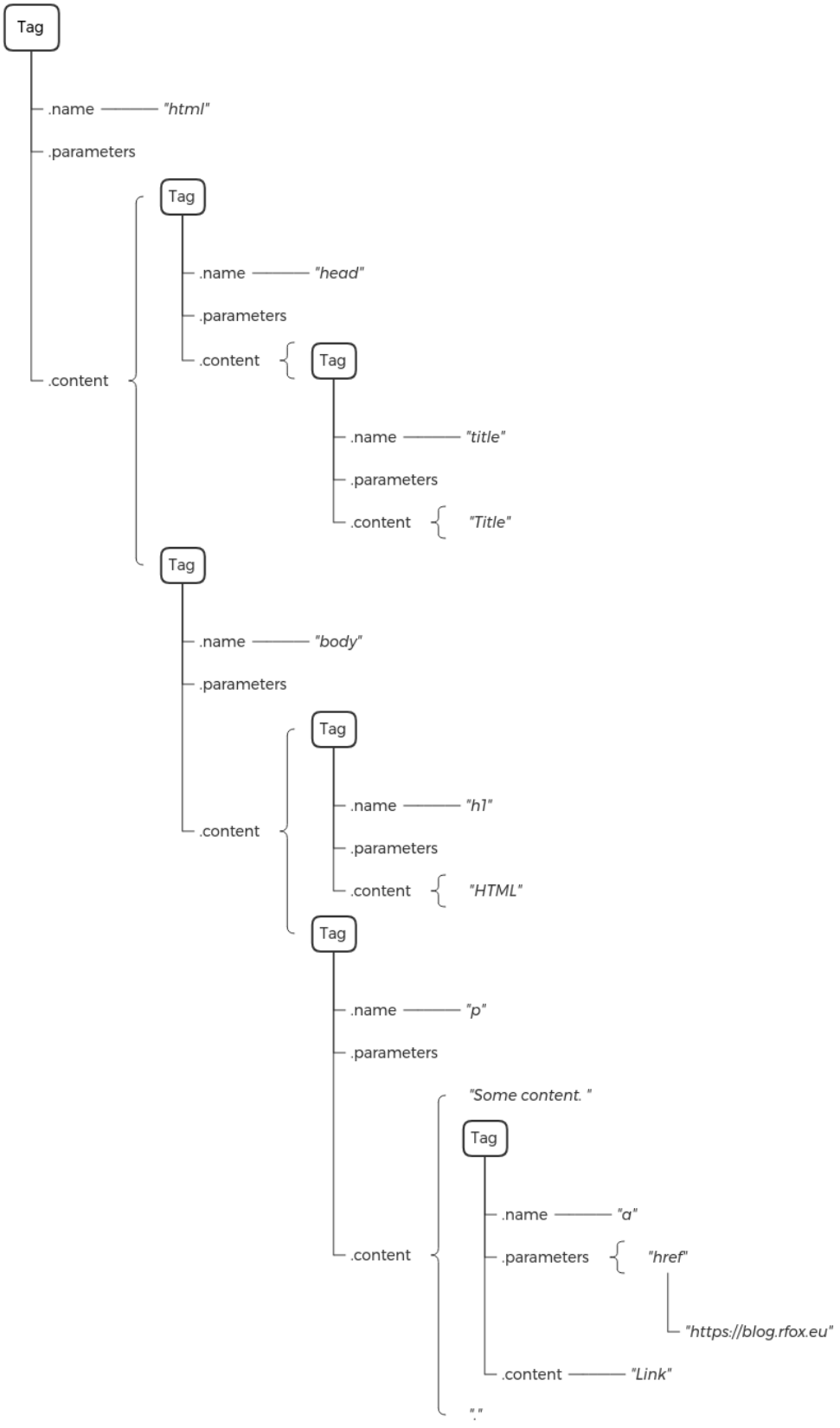
3.3 Structure of the in-memory objects

In order to be effective when working with the DOM, it is useful to understand how the tree actually looks in memory.

For example:

```
<html>
<head><title>Title</title></head>
<body>
<h1>HTML</h1>
<p>Some content. <a href="https://blog.rfox.eu">Link</a>.</p>
</body>
</html>
```

is parsed to following structure:


Tag
.name "html"
.parameters
.content
Tag
.name "head"
.parameters
.content
Tag
.name "title"
.parameters
.content "Title"
Tag
.name "body"
.parameters
.content
Tag
.name "h1"
.parameters
.content "HTML"
Tag
.name "p"
.parameters
.content
"Some content. "
Tag
.name "a"
.parameters "href"
"https://blog.rfox.eu"
.content "Link"
"."

Note, that the whitespaces are part of the *content* properties.

3.4 Properties of the Tag object

As mentioned before, the *Tag* object basically consist of two properties *parameters* and *content*. First one contains dictionary for the tag parameters, second one contains everything that was in the HTML tag as a list.

Since writing `.content` and `.prarameters` is cumbersome, you can also use shortcuts *p* and *c* to access the same data:

```
>>> dom = dhtmlparser3.parse('<tag param1="1" param2="2">  <content />  </tag>')
>>> dom.p
SpecialDict([('param1', '1'), ('param2', '2')])
>>> dom.c
['  ', Tag('content', parameters=SpecialDict(), is_non_pair=True), '  ']
```

But the *Tag* objects also support dictionary-like interface, so you can access both *parameters* and *content* using square brackets:

```
>>> dom["param1"]
'1'
>>> dom[0]
Tag('content', parameters=SpecialDict(), is_non_pair=True)
```

Notice, that `dom[0]` returns expected tag, even though the *content* property contains as a first element a whitespace:

```
>>> dom.c
['  ', Tag('content', parameters=SpecialDict(), is_non_pair=True), '  ']
```

That is because it is not using the *content* for access, but *tags* property, which returns only tags:

```
>>> dom.tags
[Tag('content', parameters=SpecialDict(), is_non_pair=True)]
```

Square brackets can be also used for setting and deleting the sub-elements:

```
>>> dom[0] = dhtmlparser3.Tag("new_content", {"param": "new_param"},  is_non_pair=True)
>>> str(dom)
'<tag param1="1" param2="2">  <new_content param="new_param" />  </tag>'
```

```
>>> del dom[0]
>>> str(dom)
'<tag param1="1" param2="2">    </tag>'
```

3.4.1 Inserting tags using square brackets

To make your life a bit easier, you can also insert tags using square brackets with slices. This is similar to calling `.insert()` method on a list, but it does support `-1` as an index for appending:

```
>>> dom[0:] = dhtmlparser3.Tag("test", is_non_pair=True)
>>> dom[-1:] = dhtmlparser3.Tag("another", is_non_pair=True)
>>> str(dom)
'<tag param1="1" param2="2"><test />    <another /></tag>'
```

Note that the whitespaces were not affected, as the tags were *inserted*, not replaced.

3.5 Looking for specific things

Most common usage of the dhtmlparser3 is to look for specific things. For this, there are two handy methods *find()* and *wfind()*. First one traverses the tree depth first, second breadth first.

Usually, you'll probably use *find()*.

Let's see an example:

```
dom = dhtmlparser3.parse("""
    <div class="page-body">
      <figure id="d4cb77d7-f9fe-4796-b38c-f84ac9efb178" class="link-to-page">
        <h1><a class="unroll_category" href="en/Weekly_updates.html" title="Weekly␣
↪updates"> <u>Category: <em>Weekly updates</em></u></a></h1>
        <div style="margin-left: 1em">
          <h4><a class="" href="en/Weekly_updates/Newsletter_2021-10-29_Monster_post.html
↪" title="Newsletter 2021-10-29; Monster post"> Newsletter 2021-10-29; Monster post</a>
↪<time>(@2021-10-29)</time></h4>
          <p style="margin-top: -1em;"><em>Some of the more important points in my year,␣
↪and also general update regarding the tinySelf, objWiki and other projects of mine.␣
↪Expect a lot of text.</em></p>
        </div>
        <div style="margin-left: 1em">
          <h4><a class="" href="en/Weekly_updates/Newsletter_2021-01-08_Defragmentation_
↪in_progress.html" title="Newsletter 2021-01-08; Defragmentation in progress">␣
↪Newsletter 2021-01-08; Defragmentation in progress</a> <time>(@2021-01-08)</time></h4>
          <p style="margin-top: -1em;"><em>Opensource contributions, book report,␣
↪improvements, published blogposts and generally progress in my life and work.</em></p>
        </div>
        <div style="margin-left: 1em">
          <h4><a class="" href="en/Weekly_updates/Newsletter_2020-09-12_Waves_of_
↪productivity.html" title="Newsletter 2020-09-12; Waves of productivity"> Newsletter␣
↪2020-09-12; Waves of productivity</a> <time>(@2020-09-12)</time></h4>
          <p style="margin-top: -1em;"><em>Some of the stuff I did in last month and a␣
↪half.</em></p>
        </div>
        <h4 style="text-align: right;"><a href="en/Weekly_updates.html" title="Weekly␣
↪updates">& 23 more blogposts</a></h4>
      </figure>
      <figure id="aaede967-1ab7-4910-ba42-5ea5f8ee480f" class="link-to-page">
        <h1><a class="unroll_category" href="en/Technological_marvels.html" title=
```

(continues on next page)

(continued from previous page)

```
↪"Technological marvels"> <u>Category: <em>Technological marvels</em></u></a></h1>
        <div style="margin-left: 1em">
          <h4><a class="" href="en/Technological_marvels/Microtron_under_the_Vitkov_Hill.
↪html" title="Microtron under the Vítkov Hill"> Microtron under the Vítkov Hill</a>
↪<time>(last modified @2021-06-26)</time></h4>
          <p style="margin-top: -1em;"><em>Introduction to rare particle accelerator in␣
↪Prague.</em></p>
        </div>
        <div style="margin-left: 1em">
          <h4><a class="" href="en/Technological_marvels/LVR-15_research_reactor_near_
↪Prague.html" title="LVR-15 research reactor near Prague"> LVR-15 research reactor near␣
↪Prague</a> <time>(@2019-11-24)</time></h4>
          <p style="margin-top: -1em;"><em>Notes from my visit of Czechoslovakian LVR-15␣
↪research nuclear reactor in Řež near Prague.</em></p>
        </div>
        <div style="margin-left: 1em">
          <h4><a class="" href="en/Technological_marvels/The_beauty_of_a_fusion_reactor.
↪html" title="The beauty of a fusion reactor"> The beauty of a fusion reactor</a> <time>
↪(@2019-06-12)</time></h4>
          <p style="margin-top: -1em;"><em>Introduction of interesting research fusion␣
↪reactor built in Germany. Contains a lot of pictures.</em></p>
        </div>
      </figure>
    </div>
""")
```

Let's say, that I want to obtain all links that point to the weekly updates category:

```
>>> dom.find("a", fn=lambda x: x.p.get("href", "").startswith("en/Weekly"))
[Tag('a', parameters=SpecialDict([('class', 'unroll_category'), ('href', 'en/Weekly_
↪updates.html'), ('title', 'Weekly updates')]), is_non_pair=False),
 Tag('a', parameters=SpecialDict([('class', ''), ('href', 'en/Weekly_updates/Newsletter_
↪2021-10-29_Monster_post.html'), ('title', 'Newsletter 2021-10-29; Monster post')]), is_
↪non_pair=False),
 Tag('a', parameters=SpecialDict([('class', ''), ('href', 'en/Weekly_updates/Newsletter_
↪2021-01-08_Defragmentation_in_progress.html'), ('title', 'Newsletter 2021-01-08;␣
↪Defragmentation in progress')]), is_non_pair=False),
 Tag('a', parameters=SpecialDict([('class', ''), ('href', 'en/Weekly_updates/Newsletter_
↪2020-09-12_Waves_of_productivity.html'), ('title', 'Newsletter 2020-09-12; Waves of␣
↪productivity')]), is_non_pair=False),
 Tag('a', parameters=SpecialDict([('href', 'en/Weekly_updates.html'), ('title', 'Weekly␣
↪updates')]), is_non_pair=False)]
```

As you can see, I am using lambda parameter to match specific part of the URL.

Note that I am specifically using `x.p.get("href", "")` to access the `href` parameter, because not every <a> element has to have it set, in which case it will cause and exception.

To obtain the URL's, all I need to do is use simple list comprehension:

```
>>> links = dom.find("a", fn=lambda x: x.p.get("href", "").startswith("en/Weekly"))
>>> [f"https://blog.rfox.eu/{x['href']}" for x in links]
['https://blog.rfox.eu/en/Weekly_updates.html',
```

(continues on next page)

(continued from previous page)

```
 'https://blog.rfox.eu/en/Weekly_updates/Newsletter_2021-10-29_Monster_post.html',
 'https://blog.rfox.eu/en/Weekly_updates/Newsletter_2021-01-08_Defragmentation_in_
↪progress.html',
 'https://blog.rfox.eu/en/Weekly_updates/Newsletter_2020-09-12_Waves_of_productivity.html
↪',
 'https://blog.rfox.eu/en/Weekly_updates.html']
```

If I wanted content of the tags, it would be as simple as:

```
>>> [x.content_without_tags() for x in links]
[' Category: Weekly updates',
 ' Newsletter 2021-10-29; Monster post',
 ' Newsletter 2021-01-08; Defragmentation in progress',
 ' Newsletter 2020-09-12; Waves of productivity',
 '& 23 more blogposts']
```

More complex example may include matching the elements in the tree, for example select all links, which don't have as a parent, but have `class` parameter set to `active`:

```
>>> dom = dhtmlparser3.parse("""
<div>
  <a href="not this" class="active">link</a>
  <a href="not this">link</a>
  <span>
    <a href="not this">link</a>
    <a href="this one" class="active">this one</a>
  </span>
</div>
""")
>>> dom.find("a", {"class": "active"}, fn=lambda x: x.parent.name != "div")
[Tag('a', parameters=SpecialDict([('href', 'this one'), ('class', 'active')]), is_non_
↪pair=False)]
```

3.5.1 match()

In addition to *find()*, there is also *match()*. What it does is that it matches the paths specified by the arguments:

```
>>> dom.match("span", "a")
[Tag('a', parameters=SpecialDict([('href', 'not this')]), is_non_pair=True),
 Tag('a', parameters=SpecialDict([('href', 'this one'), ('class', 'active')]), is_non_
↪pair=False)]
```

As you can see, the elements which are <a> tags in were matched. You can use all arguments that *find()* takes as dictionaries (`**kwargs`):

```
>>> dom.match("span", {"name": "a", "p": {"class": "active"}})
[Tag('a', parameters=SpecialDict([('href', 'this one'), ('class', 'active')]), is_non_
↪pair=False)]
```

or lists (`*args`):

```
>>> dom.match("span", ["a", {"class": "active"}])
[Tag('a', parameters=SpecialDict([('href', 'this one'), ('class', 'active')]), is_non_
↪pair=False)]
```

This way, you can look for patterns and sub-patterns and so on.

3.5.2 wfind()

wfind() implements similar pattern matching to *match()*, but always wraps the result in the empty container object. This way, it is possible to chain the calls (which is not possible for *find()* and *match()*, because they return `list`):

```
>>> dom.wfind("span").wfind("a")
Tag(name="", content=[Tag('a', parameters=SpecialDict([('href', 'not this')]), is_non_
↪pair=False), Tag('a', parameters=SpecialDict([('href', 'this one'), ('class', 'active
↪')]), is_non_pair=False)])
```

BTW, you can check if the returned container matches anything, by using `if` condition on the whole container, because it implements bool magic method:

```
>>> bool(dom.wfind("span").wfind("a"))
True
>>> bool(dom.wfind("blah"))
False
```

So for example:

```
if dom.wfind("span").wfind("a"):
  # .. do something
```

3.6 Other useful things to know

3.6.1 remove()

remove() will remove given element from the sub-tree. The element has to be an actual element from the tree, that is the result of *find()* call or some other method of traversal of the tree.

For example, to remove all links:

```
>>> for link in dom.find("a", {"href": "not this"}):
...  dom.remove(link)
...
True
True
True
>>> print(str(dom))

<div>


  <span>
```

(continues on next page)

(continued from previous page)

```
    <a href="this one" class="active">this one</a>
  </span>
</div>
```

3.6.2 Tag.prettify()

As you can see, a lot of whitespaces were left. To get rid of them, you can call *Tag.prettify()*:

```
>>> print(dom.prettify())
<div>
  <span>
    <a href="this one" class="active">this one</a>
  </span>
</div>
```

3.6.3 replace_with()

You can replace tags by calling *replace_with()*, and depending on the `keep_content` parameter (default `True`), it will keep the content same:

```
>>> dom.find("span")[0].replace_with(dhtmlparser3.Tag("p"))
>>> print(dom.prettify())
<div>
  <p>
    <a href="this one" class="active">this one</a>
  </p>
</div>
```

3.6.4 Create DOM

And you can of course create the DOM from scratch:

```
>>> from dhtmlparser3 import Tag
>>> xml = Tag("xml")
>>> xml[0:] = Tag("container")
>>> xml[0][-1:] = Tag("item", {"parameter": "value"}, ["content"])
>>> xml[0][-1:] = Tag("item", {"parameter": "another value"}, ["another content"])
>>> print(xml.prettify())
<xml>
  <container>
    <item parameter="value">content</item>
    <item parameter="another value">another content</item>
  </container>
</xml>
```

3.7 Things that may be useful to know

Parsing is case sensitive.

Matching using *find()* is case insensitive. You can make it case sensitive by setting `case_sensitive` parameter to `True`.

Instead of *find()*, you can call *find_depth_first_iter()* to get lazy evaluated iterator.

You can compare elements using ==, and it will compare only equality of *name*, *parameters* and `is_non_pair`, not the subtree.

It is possible to iterate over tags in given element by simply using it in `for` loop. This will skip whitespaces.

If you want `bytes` representation of the DOM string, call `bytes(dom)` and it will work.

All elements have *parent* set by default. If you insert new elements using square brackets operator, it will be correctly set. If you however set new part of the sub-tree manually by inserting it to *content*, you have to set it manually, or call *double_link()* on the element where you've inserted it.

parse() returns either root element, or virtual container element, which is *Tag* with empty name, if there are multiple root elements.

Non-pair elements are autodetected even if they are not valid HTML, and parser should in general handle gracefully malformed HTML.

Non-key-value parameters like for example `<tag rectangle>` are parsed to empty value in *parameters*:

```
>>> dhtmlparser3.parse("<tag rectangle>")
Tag('tag', parameters=SpecialDict([('rectangle', '')]), is_non_pair=True)
>>> str(dhtmlparser3.parse("<tag rectangle>"))
'<tag rectangle />'
```

Notice how the tag was correctly recognized as non-pair.

CHAPTER

FOUR

PACKAGE CONTENT

4.1 dhtmlparser3

Most important function here is `parse()`, which is used to process string and create Document Object Model.

class dhtmlparser3.**FileParser**(*path: str*, *case_insensitive_parameters=True*)

write(*path: Optional[str] = None*)

dhtmlparser3.**parse**(*string: str*, *case_insensitive_parameters=True*)

dhtmlparser3.**parse_file**(*path: str*, *case_insensitive_parameters=True*)

4.1.1 Submodules

dhtmlparser3.tags.tag

class dhtmlparser3.tags.tag.**Tag**(*name*, *parameters=None*, *content=None*, *is_non_pair=False*)

Bases: object

name

Name of the parsed tag.

Type
str

parameters

Dictionary for the parameters.

Type
SpecialDict

content

List of sub-elements.

Type
list

parent

Reference to parent element.

Type
Tag

property p: Dict[str, str]

Shortcut for .parameters, used extensively in tests.

property c

Shortcut for .content, used extensively in tests.

property tags: List[*Tag*]

Same as .c, but returns only tag instances. Useful for ignoring whitespace and comment clutter and iterating over the real dom structure.

double_link()

Make the DOM hierarchy double-linked. Each content element now points to the parent element.

content_without_tags() → str

Return content but remove all tags.

This is sometimes useful for processing messy websites.

remove(*offending_item: Union[str,* Tag*,* Comment*]*) → bool

Remove *offending_item* anywhere from the dom.

Item is matched using *is* operator, so it better be something you've found using .find() or other relevant methods.

Returns
True if the item was found and removed.

Return type
bool

remove_item(*item: Union[str,* Tag*,* Comment*]*)

Remove the item from the .content property.

to_string() → str

Get HTML representation of the tag and the content.

tag_to_str() → str

Convert just the tag with parameters to string, without content.

content_str(*escape=False*) → str

Return everything in between the tags as string.

Parameters
escape (*bool*) – Escape the content. Default False.

replace_with(*item:* Tag*, keep_content: bool = False*)

Replace this Tag with another *item*.

Parameters

- **item** (Tag, *str*) – Item to replace this with.
- **keep_content** (*bool*) – Keep the original content. Default *False*.

wfind(*name*, *p=None*, *fn=None*, *case_sensitive=False*)

match(**args*)

Recursively call *find* for each element in **args*. That means fuzzy matching, like "find all * `s, which have this `<p>* element, which has this *<a>* in it.

Example

dom.match("div", ["p", {"class": "great"}], "a")

Parameters
***args** (*list*) – List of paths to match.

Returns
List of matched elements.

Return type
list

match_paths(**args*)

Exactly match the path given by the arguments.

Example

dom.match("body", ["div", {"class": "page-body"}], "p")

This will match the path only if it really goes like this. If the <*p*> is for example wrapped in , it won't be matched.

Parameters
***args** (*list*) – List of paths to match.

Returns
List of matched elements.

Return type
list

find(*name*, *p=None*, *fn=None*, *case_sensitive=False*) → List[*Tag*]

Find (depth first) all tags with given parameters.

Parameters

- **name** (*str*) – Name of the tag you are looking for. Use *""* for all.
- **p** (*dict*) – Parameters to match.
- **fn** (*lambda fn*) – Lambda expecting one argument. It will be tested for each element in the tree.
- **case_sensitive** (*bool*) – Use case sensitive search. Default *False*.

findb(*name*, *p=None*, *fn=None*, *case_sensitive=False*) → List[*Tag*]

Find (breadth first) all tags with given parameters.

Parameters

- **name** (*str*) – Name of the tag you are looking for. Use *""* for all.
- **p** (*dict*) – Parameters to match.
- **fn** (*lambda fn*) – Lambda expecting one argument. It will be tested for each element in the tree.
- **case_sensitive** (*bool*) – Use case sensitive search. Default *False*.

find_depth_first_iter(*name*, *p=None*, *fn=None*, *case_sensitive=False*) → Iterator[*Tag*]

find_breadth_first_iter(*name*, *p=None*, *fn=None*, *case_sensitive=False*) → Iterator[*Tag*]

depth_first_iterator(*tags_only=False*) → Iterator[Union[*Tag*, str, *Comment*]]

breadth_first_iterator(*tags_only=False*, *_first_call=True*) → Iterator[Union[*Tag*, str, *Comment*]]

prettify(*depth=0*, *dont_format=False*) → str

dhtmlparser3.tags.comment

class dhtmlparser3.tags.comment.**Comment**(*content=None*)

Bases: object

to_string()

prettify(*depth*, *dont_format=False*)

dhtmlparser3.parser

class dhtmlparser3.parser.**Parser**(*string: str*, *case_insensitive_parameters=True*)

Bases: object

NONPAIR_TAGS = {'base', 'br', 'frame', 'hr', 'img', 'input', 'meta', 'spacer'}

parse_dom() → *Tag*

dhtmlparser3.tokenizer

class dhtmlparser3.tokenizer.**Tokenizer**(*string: str*)

Bases: object

tokens: List[*Token*]

MAX_ENTITY_LENGTH = 20

tokenize() → List[*Token*]

tokenize_iter() → Iterator[*Token*]

return_reset_buffer()

advance()

is_at_end()

peek_is(*char*)

peek()

peek_two_is(*char*)

peek_two()

dhtmlparser3.tokens

class dhtmlparser3.tokens.**Token**

Bases: object

class dhtmlparser3.tokens.**TextToken**(*content=''*)

Bases: *Token*

to_text()

class dhtmlparser3.tokens.**TagToken**(*name=''*, *parameters=None*, *is_non_pair=False*, *is_end_tag=False*)

Bases: *Token*

to_tag()

class dhtmlparser3.tokens.**ParameterToken**(*key=''*, *value=''*)

Bases: *Token*

class dhtmlparser3.tokens.**CommentToken**(*content=''*)

Bases: *Token*

class dhtmlparser3.tokens.**EntityToken**(*content=''*)

Bases: *Token*

NAMED_ENTITIES = {'&': '&', ''': "'", '¢': '¢', '©': '©', '€': '€', '>': '>', '<': '<', ' ': '\xa0', ' ': '\xa0', '£': '£', '"': '"', '®': '®', '¥': '¥'}

to_text()

dhtmlparser3.quoter

This module provides ability to quote and unquote strings using backslash notation.

dhtmlparser3.quoter.**escape**(*inp*)

Escape *quote* in string *inp*.

Example usage:

```
>>> escape('hello "')
'hello &quot;'
```

Parameters
inp (*str*) – String in which *quote* will be escaped.

Returns
Escaped string.

Return type
str

dhtmlparser3.specialdict

class dhtmlparser3.specialdict.**SpecialDict**(**args*, ***kwargs*)

Bases: OrderedDict

This dictionary stores items case sensitive, but compare them case INsensitive.

clear() → None. Remove all items from od.

get(*k*, *d=None*)

Return the value for key if key is in the dictionary, else default.

has_key(*key*)

iteritems(**args*, ***kwargs*)

iterkeys(**args*, ***kwargs*)

itervalues(**args*, ***kwargs*)

keys() → a set-like object providing a view on D's keys

items() → a set-like object providing a view on D's items

values() → an object providing a view on D's values

copy() → a shallow copy of od

4.2 Unittests

Almost everything should be tested. You can run the tests like this:

```
$ export PYTHONPATH="src/:$PYTHONPATH"
$ py.test
============================= test session starts ==============================
platform linux -- Python 3.8.10, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/bystrousak/Desktop/Syncthing/c0d3z/python/libs/pyDHTMLParser3
collected 101 items

tests/test_comment.py .                                                  [  0%]
tests/test_escapers.py ..                                                [  2%]
tests/test_parser.py ..........                                          [ 12%]
tests/test_specialdict.py ............                                   [ 24%]
tests/test_tag.py ..................................                     [ 62%]
tests/test_tokenizer.py .....................................            [100%]

============================== 101 passed in 0.16s ==============================
```

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

N

P

R

S

T

V

W