doreah

Jul 16, 2023

Contents:

CHAPTER 1

Overview

Doreah is a useful little toolkit that offers shortcuts and abstractions for common operations.

1.1 Installing

Install doreah with the simple command `pip install doreah`.

1.2 Configuration

Each module can be configured with a call to the function config(). However, it is recommended to use a `.doreah` configuration file in your project's directory. This way, the correct configuration will be used from the first import.

The .doreah file follows a simple key-value-format where the key is comprised of the module name, a dot and the configuration parameter, e.g.:

```
logging.verbosity = 2
```

CHAPTER 2

Modules

2.1 Authentication

2.1.1 API Reference

2.2 Caching

2.2.1 API Reference

2.3 Database

2.3.1 API Reference

2.4 Input/Output

2.4.1 API Reference

2.5 Logging

2.5.1 API Reference

2.6 Persistence

2.6.1 API Reference

2.7 PyHP / Python Hypertext Processor

2.7.1 How To

PyHP Syntax

Any strict (XML-compliant) html document is a valid pyhp document. Server-side processing can be achieved with the help of `<pyhp>` nodes. Their meaning is determined by their arguments. Nodes without any arguments are code blocks: they may contain arbitrary python code. Indentation can be chosen to match the position in the xml tree, but must then be consistent within the block. The first and last line (those containing the `<pyhp>` and `</pyhp>` tags) must not contain code.

The following pyhp nodes are available

echo The supplied expression will be evaluated and returned as string

```
<pyhp echo="len(stuff)" />
```

if Everything within this node is only sent to the client if the condition evaluates to true

```
<pyhp if="me['rank'] == 1 or !me.isintouchwithreality()">I'm the best!</pyhp>
```

for loop Everything within this node will be evaluated for each element in the iterable / mapping

```
<pyhp for="city" in="patriarchs" separator=" | ">The current patriarch of <pyhp echo=
↪"city" /> is <pyhp echo="patriarchs['city']" />. </pyhp>
```

assignment Assigns to a variable

```
<pyhp save="complicated_db_call(somestuff)[len(somelist)]['info']['important']" as=
↪"importantinfo" />
```

include Includes another pyhp file at this location.

```
<pyhp include="sidebar.pyhp" />
```

You can also access variables inside arguments of regular html nodes with curly braces:

```
<a href="{site.url}"><pyhp echo="site.name" /></a>
```

2.7.2 API Reference

2.8 Regular

2.8.1 API Reference

2.9 Scraping

2.9.1 How To

Step Instructions

This module provides a simplified interface to parse XML trees with a set of predefined steps. These need to be supplied in a list as dicts with the keys *steptype* and 'instruction', although the second may be omitted for steps that do not have any further instructions.

The following steps are possible:

- **Steps that work for both single elements and lists:**

  xpath follows the xpath down the tree, returns first element (node -> node or string)

  prefix adds a prefix to the string (string -> string)

  suffix appends a suffix to the string (string -> string)

  rmprefix removes a prefix if present (string -> string)

  regex Replaces the string matched by the supplied regex with its first capture group (string -> string)

  last splits the string and returns last element (string -> string)

- **Steps that work for single elements and return a single element:**

  follow follows the specified link and returns the root node of the resulting document (string -> node)

- **Steps that work for single elements and split them into a list:**

  split splits the string (string -> stringlist)

  makelist turns an element into a list consisting of that element (string -> stringlist, node -> nodelist)

  xpathls follows the xpath down the tree, returns all elements (node -> nodelist or stringlist)

- **Steps that work for lists and merge them back into a single element:**

  pick picks the n-th element from the list (nodelist -> node, stringlist -> string)

  combine combines all strings of the list (stringlist -> string)

Scraping feeds

`parse_all()` is a function to scrape any well-structured feed of regular elements. Since its arguments may be confusing, let's look at a simple example. Say we want to scrape all locations of a website that shows 3 entries per page and its URLs look like this:

https://coolplaces.tld/top?start=0
https://coolplaces.tld/top?start=3
https://coolplaces.tld/top?start=6
etc...

We would then supply `base_url="https://bestgallery.tld/newest?start={page}"`, `start_page=0` and `page_multiplier=3` (since Page 0 needs a 0, page 1 needs a 3 and so on).

If our page has a weird URL logic, we can simply supply a function instead that takes the logical page number (0, 1, 2, ...) as input and returns the string that should be inserted into the URL.

Now let's have a look at the relevant part of our webpage:

```
<body>
	<div id="cards_area">
		<div class="place_box" id="place_box_rivendell">
			<div style="background-image('/rivendell.png');"></div>
			<h3 class="place_name">Rivendell</h3>
			<span class="place_leader">Leader: Elrond</span>
		</div>
		<div class="place_box" id="place_box_gondolin">
			<div style="background-image('/tumladen_vale.jpg');"></div>
			<h3 class="place_name">Gondolin</h3>
			<span class="place_leader">Leader: Turgon</span>
		</div>
```

(continues on next page)

(continued from previous page)

```
                <div class="place_box" id="place_box_holymountain">
                        <div style="background-image('/oiolosse.png');"></div>
                        <h3 class="place_name">Taniquetil</h3>
                        <span class="place_leader">Leader: Manwë</span>
                </div>
        </div>
</body>
```

As `steps_elements` we need to supply the steps to acquire a list of elements - simple enough:

```
[
        {"type":"xpath","instruction":"//div[@id='cards_area']//div[@class='place_box
↪']"}
]
```

Now, we want to return several pieces of information from each element. As `steps_content`, we pass:

```
{
        "identifier":[
                {"type":"xpath","instruction":"./@id"},
                {"type":"rmprefix","instruction":"place_box_"}
        ],
        "image_url":[
                {"type":"xpath","instruction":"./div/@style"},
                {"type":"regex","instruction":"background-image('(.*)');"}
        ],
        "name":[
                {"type":"xpath","instruction":"./h3/text()"}
        ],
        "leader":[
                {"type":"xpath","instruction":"./span/text()"},
                {"type":"regex","instruction":"Leader: (.*)"}
        ]
}
```

This will iterate through all places and save the according values in a dictionary:

```
[
        {
                "identifier": "rivendell",
                "image_url": "rivendell.png",
                "name": "Rivendell",
                "leader": "Elrond"
        },
        {
                "identifier": "gondolin",
                "image_url": "tumladen_vale.jpg",
                "name": "Gondolin",
                "leader": "Turgon"
        },
        {
                "identifier": "holymountain",
                "image_url": "oiolosse.png",
                "name": "Taniquetil",
                "leader": "Manwë"
        },
]
```

If we pass the argument `stop=42`, the parsing will stop after we have found 42 arguments. Alternatively (or additionally), we can pass as `stopif` the following:

```
{
        "leader":lambda x: x=="Morgoth" or x=="Sauron",
        "image_url":lambda x: x.endswith(".gif")
}
```

This means that if we parse a place with the leader “Morgoth” or “Sauron”, or if we parse a place that has a .gif-image, we immediately stop parsing.

2.9.2 API Reference

2.10 Settings

2.10.1 API Reference

2.11 Timing

2.11.1 API Reference

2.12 TSV

2.12.1 API Reference