scrapy-mosquitera Documentation

Release 0.1.0

Scrapinghub

May 19, 2016

Contents

> How can I scrape items off a site from the last five days?
>
> —Scrapy User

That question started the development of **scrapy-mosquitera**, a tool to help you restrict crawling and scraping scope using *matchers*.

Matchers are simple Python functions that return the validity of an element under certain restrictions.

The first goal in the project was date matching, but you can create your own matcher for your own crawling and scraping needs.

CHAPTER 1

How it works

In the case where the dates are available in the URLs, you will just use the matcher function directly in your code:

```
from scrapy_mosquitera.matchers import date_matches

 date = scrape_date_from_url(url)

 if date_matches(data=date, after='5 days ago'):
    yield Request(url=url, callback=self.parse_item)
```

To handle the case when the date is only available at the time when you scrape the items, **scrapy-mosquitera** provides a `PaginationMixin` to control the crawl according to the dates scraped.

Head on to the remaining of the documentation for more details.

CHAPTER 2

Installation

The quick way:

```
pip install scrapy-mosquitera
```

CHAPTER 3

Documentation contents

3.1 Matchers

3.1.1 Creating your own matcher

A matcher is a simple function taking the data to be evaluated as argument(s) and returning a boolean value according to its validity.

Current matchers

3.1.2 Date Matchers

The date matchers use a lot of words to delimit their date range. They are separated to set the maximum and minimum date. In order of precedence they are for minimum date:

- min_date
- on
- after
- since

And for maximum date:

- max_date
- on
- before

Their values could be dates parseables by dateparser, `date` or `datetime` objects. They also support `None` value, so that limit isn't verified.

`scrapy_mosquitera.matchers.`**`date_matches`**(*data*, ***kwargs*)

Return `True` if `data` is a date in the valid date range. Otherwise `False`.

Parameters

- **`data`** (*`string, date or datetime`*) – the date to validate
- **`kwargs`** (*`dict`*) – special delimitation parameters

Return type bool

scrapy_mosquitera.matchers.**date_in_period_matches**(*data*, *period='day'*, *check_maximum=True*, ***kwargs*)

Return `True` if `data` is a date in the valid date range defined by `period`. Otherwise `False`.

This matcher is ideal for cases like the following one.

A forum post is created at *04-10-2016*. Then on *04-28-2016*, I try to scrape the forum covering the last few days. However, the forum doesn't display the post date but some sentences like *X weeks ago*. So, in the forum nomenclature, the posts fall in the next table:

Start date	End date	Name
04-15-2016	04-21-2016	One week ago
04-08-2016	04-14-2016	Two weeks ago
04-01-2016	04-07-2016	Three weeks ago

On *04-28-2016*, if I calculate *two weeks ago* it will return *04-14-2016*. Comparing it to the forum meaning, we're working with fixed dates and the forum with date ranges. Then, if I scrape until *04-10-2016*, the crawl will miss the posts from *04-10-2016* to *04-13-2016* since the last valid date would be *two weeks ago* (*three weeks ago* is out of scope (*04-07-2016* < *04-10-2016*)).

This matcher comes to solve this, so you can provide the period (in this case **week**) and you won't miss items by coverage issues. However, it's inclusive because to satisfy the date *04-10-2016* it will include the full week [04-08-2016, 04-14-2016], so a post-filtering should be made to only allow valid items.

Parameters

- **data** (*string, date or datetime*) – the date to validate
- **period** (*string*) – the period to evaluate ('day', 'month', 'year')
- **check_maximum** (*bool*) – check maximum date
- **kwargs** (*dict*) – special delimitation parameters

Return type bool

3.2 PaginationMixin

PaginationMixin is a mixin with a group of decorators to control the logic of requesting the next page. It has an interesting flow, which could be summarized as:

1. At the listing parsing method, every item page request is yielded. Each request is marked to be associated with the current response and any pagination requests is enqueued.
2. At the item parsing method, the matching logic is applied and each valid item and its related request is registered.
3. After comparing the yielded requests at step 1 and the requests which yielded valid items at step 2, the mixin decides to dequeue the next page request only if every request yielded a valid item.

To understand better its working, please review the *examples*.

class scrapy_mosquitera.mixin.**PaginationMixin**(**args*, ***kwargs*)

static deregister_response(*fn*)
Deregister response from the registry.

It's a decorator.

static enqueue_next_page_requests(*fn*)
Enqueue next page requests to be only requested if they meet the conditions.

It's a decorator.

static **register_requests** (*fn*)
Register requests yielded from `fn` in the registry using as key its parent response id.

It's a decorator.

3.3 Examples

scrapy-mosquitera aims scenarios where there are listings involved. However, **scrapy-mosquitera** takes a different approach whether the data to match is present in the listing or not. As it started for date validation, let's review what to do when dates are present or absent.

3.3.1 Dates present

In example, we'll consider a blog archive page.

```
<div>
  <h3>Title for Post 1</h3>
  <a href="/post1">Link</a>
  <span>Posted on 2016-04-01</span>
</div>
<div>
  <h3>Title for Post 2</h3>
  <a href="/post2">Link</a>
  <span>Posted on 2016-04-02</span>
</div>
<div>
  <h3>Title for Post 3</h3>
  <a href="/post3">Link</a>
  <span>Posted on 2016-04-03</span>
</div>
```

It's the simpler case since we can do the matching at the method parsing the listing. We will use *date_matches* to do the match and it let us control the pagination in an easy way.

```
from scrapy_mosquitera.matchers import date_matches

def parse(self, response):
  continue_to_next_page = True

  for news in response.xpath("//div"):
    date = news.xpath("./span/text()").re_first('Posted on (.*)')
    path_url = news.xpath("./a/@href").extract_first()
    url = response.urljoin(path_url)

    if date_matches(data=date, after='5 days ago'):
      yield Request(url=url, callback=self.parse_item)
    else:
      continue_to_next_page = False

  if continue_to_next_page:
    yield self.call_next_page(response)
```

3.3.2 Dates absent

For this case, we'll consider the following blog archive page layout.

```
<div>
  <h3>Title for Post 1</h3>
  <a href="/post1">Link</a>
</div>
<div>
  <h3>Title for Post 2</h3>
  <a href="/post2">Link</a>
</div>
<div>
  <h3>Title for Post 3</h3>
  <a href="/post3">Link</a>
</div>
```

Dates aren't present on the listing, but they are in each post page.

```
<h1>Title for Post</h1>
<div>Posted on 2016-04-02</div>
[...]
```

Here comes *PaginationMixin* which is a mixin specialize for these cases. To see it in action in a comparable way with the first example, let's start using their decorators. *@PaginationMixin.register_requests* has to be applied to the listing parsing method.

```
from scrapy_mosquitera.matchers import PaginationMixin

@PaginationMixin.register_requests
def parse(self, response):
  for news in response.xpath("//div"):
    path_url = news.xpath("./a/@href").extract_first()
    url = response.urljoin(path_url)

    yield Request(url=url, callback=self.parse_item)

  yield self.call_next_page(response)
```

Unfortunately, each time that the listing parsing method is called every item request will be made since we don't know yet if its content is valid or not. The method in charge of returning the next page request, in this case `call_next_page`, has to be decorated with *@PaginationMixin.enqueue_next_page_requests*.

```
@PaginationMixin.enqueue_next_page_requests
def call_next_page(self, response):
  return Request([...])
```

This decorator saves the request to be called only if it's necessary. Then, the last decorator has to be applied on the method parsing the item since it has to register if a valid item was returned. This decorator is *@PaginationMixin.deregister_response*.

```
@PaginationMixin.deregister_response
def parse_item(self, response):
  date = response.xpath("//div/text").re_first('Posted on (.*)')
  item = {'created_at': date}

  if date_matches(data=item['created_at'], after='5 days ago'):
    return item
```

After that, we're ready to run our spider. First, it will make three requests, one for each post page and the pagination request will be saved. Then, if the three post are valid, they will be scraped and the next page request will be made. Otherwise, it only scrape the valid posts and the spider run will finish.

S

D

E

P

R

S