crawlster Documentation

Release 0.1.1

Vlad Calin

Feb 25, 2018

Contents:

A simple, lightweight web crawling framework

Features:

- HTTP crawling
- Various data extraction methods (regex, css selectors, xpath)
- Very configurable and extensible

CHAPTER 1

What is crawlster?

Crawlster is a web crawling library designed to build lightweight and reusable web crawlers. It is very extensible and provides many shortcuts for the most common tasks in a web crawler, such as HTTP request sending and parsing and info extraction.

It was created out of need of a lighter framework for web crawling, as an alternative to Scrapy.

CHAPTER 2

Installation

From PyPi:

```
pip install crawlster
```

From source:

```
git clone https://github.com/vladcalin/crawlster.git
cd crawlster
python setup.py install
```

CHAPTER 3

Quick example

This is the hello world equivalent for this library:

```
import crawlster
from crawlster.handlers import JsonLinesHandler


class MyCrawler(crawlster.Crawlster):
    # items will be saved to items.jsonl
    item_handler = JsonLinesHandler('items.jsonl')

    @crawlster.start
    def step_start(self, url):
        resp = self.http.get(url)
        # we select elements with the expression and we are interested
        # only in the 'href' attribute. Also, we get only the first result
        # for this example
        events_uri = self.extract.css(resp.text, '#events > a', attr='href')[0]
        # we specify what method should be called next
        self.schedule(self.step_events_page, self.urls.join(url, events_uri))

    def step_events_page(self, url):
        resp = self.http.get(url)
        # We extract the content/text of all the selected titles
        events = self.extract.css(resp.text, 'h3.event-title a', content=True)
        for event_name in events:
            # submitting items to be processed by the item handler
            self.submit_item({'event': event_name})


if __name__ == '__main__':
    # defining the configuration
    config = crawlster.Configuration({
        # the start pages
        'core.start_urls': ['https://www.python.org/'],
        # the method that will process the start pages
```

```
        'core.start_step': 'step_start',
        # to see in-depth what happens
        'log.level': 'debug'
    })
    # starting the crawler
    crawler = MyCrawler(config)
    # this will block until everything finishes
    crawler.start()
    # printing some run stats, such as the number of requests, how many items
    # were submitted, etc.
    print(crawler.stats.dump())
```

Running the above code will fetch the event names from python.org and save them in a `items.jsonl` file in the current directory.

For more advanced usage, consult the documentation.

CHAPTER 4

Helpers

A helper is a utility class that provides certain functionality. The `Crawlster` class requires the `.log`, `.stats`, `.http` and `.queue` helpers to be provided (and are by default) for internal behaviour. These are called *core helpers*

Also, besides the core helpers, the `Crawlster` class also provides the `.urls`, `.extract` and `.regex` helpers for common tasks.

You can also create other helpers and attach them to the crawler to enhance it.

4.1 Tutorial

We will build a crawler from scratch in which we will use all the major features of this framework. We are going to write a crawler for the python.org documentation which will extract items containing the name of the module and the url for the documentation of that module for all standard libs.

You can find the example in `examples/python_org.py`.

Firstly, import the required modules

```
from crawlster import Crawlster, start, Configuration
from crawlster.handlers.jsonl import JsonLinesHandler
from crawlster.handlers.log_handler import LogItemHandler
```

- *crawlster.Crawlster* is the base class which the our crawler will inherit
- *crawlster.start()* is a decorator for specifying the first step that will be executed
- *crawlster.handlers.JsonLinesHandler* is the item handler that will write all found items (results) to a json files, one per line
- *crawlster.handlers.LogItemHandler* is the item handler that will write all found items to the console

Next, we need to start implementing the crawler class and configure it

```
class PythonOrgCrawler(Crawlster):
"""
This is an example crawler used to crawl info about all the Python modules
"""
item_handler = [LogItemHandler(),
                JsonLinesHandler('items.jsonl')]
```

Here, we subclass the `Crawlster` base class and provide the `item_handler` attribute as a list of item handlers. When we submit an item, it will pe passed to each item handler in the order they are defined.

Next, we will start implementing the start step

```
@start
def step_start(self, url):
    data = self.http.get(url)
    if not data:
        return
```

Here we decorate the `step_start` method to be used as entry point for the crawling process, then we fetch the start url using the core `http` helper (see *crawlster.helpers.RequestsHelper*). If data is `None` it means the request failed, so we return immediately.

Then we parse and extract data from the response

```
self.urls.mark_seen(url)
hrefs = self.extract.css(data.body, 'a', attr='href')
full_links = self.urls.multi_join(url, hrefs)
```

Here, with the `urls` core helper we mark the url as being seen so that it will not be processed multiple times (the python docs contain a lot of cross references between pages). Read more on it on `crawlster.helpers.UrlsHelper`.

Then we go on an extract from the body only the content of the `href` attributes of all `a` elements. The `extract` helper provides the `css` method which we use to select all relevant elements from the content of the page (the anchors) and then return only the part that we need, the `href` attribute.

After that, there is a high chance that all the extracted content are the paths of the final url (eg. `/path/to/some/page.html`) and we need to convert them to full url (`https://python.org/path/to/some/page.html`). The `urls` helper has a method which performs urljoins on elements from a list at once.

Next we need to schedule the next steps in the crawling process

```
for link in full_links:
    if '#' in link:
        continue
    self.schedule(self.process_page, link)
```

For each extracted link, we send it to `self.process_page` to process it. We do that using the *crawlster.Crawlster.schedule()* method because all steps are executed in parallel, inside workers that run in separate threads.

Next, we need a way to tell if the current page represents a module reference page.

```
def looks_like_module_page(self, page_content):
    return b'Source code:' in page_content
```

I know, kind of lame but hey... it does the job!

Next, we do all the request fetching thing again, in the next step's method

```
def process_page(self, url):
    if not self.urls.can_crawl(url):
        return
    resp = self.http.get(url)
    self.urls.mark_seen(url)
```

This time we check of the current page can be crawler (in other words, if it has been already crawled). We don't want to get stuck in an infinite loop because of the numerous cross references between pages.

Then, we check if the page is a module reference page using the method defined earlier

```
module_name = self.extract.css(resp.body,
                               'h1 a.reference.internal code span',
                               content=True)
if not module_name:
    return
self.submit_item({'name': module_name[0], 'url': url})
```

So what happens here is that we extract only the content from the elements. In some cases, that elements does not exist, so we skip the page as it is not a valid module page (eg. https://docs.python.org/3/library/idle.html).

When finding a module name, we submit it and send it through the item handlers with the *crawlster.Crawlster.submit_item()* method.

The crawler class is done!

All that is left to do is starting it:

```
if __name__ == '__main__':
    crawler = PythonOrgCrawler(Configuration({
        "core.start_urls": [
            "https://docs.python.org/3/library/index.html"
        ],
        "log.level": "debug",
        "pool.workers": 3,
    }))
    crawler.start()
    pprint.pprint(crawler.stats.dump())
```

There, we initialize the `PythonOrgCrawler` with a configuration.

- `core.start_urls` is a list of starting urls. The start step will be called once for each item in this list.
- `log.level` will set the logging level so that we'll see more in the console when we run the crawler.
- `pool.workers` will set the worker thread pool's size. For this example, a concurrency level of 3 is more than enough.

By calling the *crawlster.Crawlster.start()* method we start the crawling process and after that we we'll print some nice stats about what happened.

Now go to a terminal, and assuming you wrote the crawler in a `python_org.py` file, run:

```
python python_org.py
```

Now all should work and after approx. 15 seconds, it should finish.

There should be stats printed in console

```
{'http.download': 16373171,
 'http.requests': 300,
```

```
 'http.upload': 0,
 'items': 185,
 'time.duration': 14.879282,
 'time.finish': datetime.datetime(2018, 1, 1, 17, 39, 5, 917190),
 'time.start': datetime.datetime(2018, 1, 1, 17, 38, 51, 37908)}
```

and the results should be in the `items.jsonl` file in the current directory

```
(crawlster) vladcalin@mylaptop ~/crawlster $ tail items.jsonl
{"name": "calendar", "url": "https://docs.python.org/3/library/calendar.html"}
{"name": "struct", "url": "https://docs.python.org/3/library/struct.html"}
{"name": "stringprep", "url": "https://docs.python.org/3/library/stringprep.html"}
{"name": "textwrap", "url": "https://docs.python.org/3/library/textwrap.html"}
{"name": "difflib", "url": "https://docs.python.org/3/library/difflib.html"}
{"name": "collections", "url": "https://docs.python.org/3/library/collections.html"}
{"name": "codecs", "url": "https://docs.python.org/3/library/codecs.html"}
{"name": "string", "url": "https://docs.python.org/3/library/string.html"}
{"name": "re", "url": "https://docs.python.org/3/library/re.html"}
{"name": "datetime", "url": "https://docs.python.org/3/library/datetime.html"}
```

That's all! Have fun crawling (in a responsible manner)!

Here's the whole crawler code after putting everything together:

```
import pprint

from crawlster import Crawlster, start, JsonConfiguration, Configuration
from crawlster.handlers.jsonl import JsonLinesHandler
from crawlster.handlers.log_handler import LogItemHandler


class PythonOrgCrawler(Crawlster):
    """
    This is an example crawler used to crawl info about all the Python modules
    """
    item_handler = [LogItemHandler(),
                    JsonLinesHandler('items.jsonl')]

    @start
    def step_start(self, url):
        data = self.http.get(url)
        if not data:
            return
        self.urls.mark_seen(url)
        hrefs = self.extract.css(data.body, 'a', attr='href')
        full_links = self.urls.multi_join(url, hrefs)
        for link in full_links:
            if '#' in link:
                continue
            self.schedule(self.process_page, link)

    def process_page(self, url):
        if not self.urls.can_crawl(url):
            return
        resp = self.http.get(url)
        self.urls.mark_seen(url)
        if not self.looks_like_module_page(resp.body):
            return
```

```
        module_name = self.extract.css(resp.body,
                                       'h1 a.reference.internal code span',
                                       content=True)
        if not module_name:
            return
        self.submit_item({'name': module_name[0], 'url': url})

    def looks_like_module_page(self, page_content):
        return b'Source code:' in page_content


if __name__ == '__main__':
    crawler = PythonOrgCrawler(Configuration({
        "core.start_urls": [
            "https://docs.python.org/3/library/index.html"
        ],
        "log.level": "debug",
        "pool.workers": 3,
    }))
    crawler.start()
    pprint.pprint(crawler.stats.dump())
```

4.2 How to

Here you will find some more in-depth guides on various topics.

4.2.1 Configuration

Configuration is managed via the `config` attribute of the crawler class.

The following configuration methods are available:

- *`crawlster.config.Configuration`* - the most basic configuration where you specify the configuration directly into your crawler
- *`crawlster.config.JsonConfiguration`* - manage configuration from a JSON file.

Configuration keys

Configuration keys are populated from all helpers when the crawling starts.

The following configuration keys are the default:

- `core.start_urls` - a list of urls that will be firstly processed in the start step. Is required.
- `core.workers` - the number of worker threads to be used. Defaults to the number of CPU core.

Each helper defines some extra configuration options, usually under the name `helper_name.option_name`.

Examples

The basic configuration:

```
config = Configuration({
    'core.start_step': 'my_custom_step',
    'core.workers': 10,
    'core.start_urls': ['http://example.com', 'http://example2.com']
})
```

Json configuration:

```
config = JsonConfiguration('config.json')

# config.json

{
    "core.start_step": "my_custom_step",
    "core.workers": 10,
    "core.start_urls": ["http://example.com", "http://example2.com"]
}
```

We then pass the configuration object on the crawler class initialisation

```
configuration = Configuration(...)
crawler = MyCrawlerClass(configuration)
crawler.start()
```

This is very useful when we need to reuse the same crawler to crawl multiple sites and only some configuration differs.

4.2.2 Making HTTP requests

Http requests are made through the `.http` helper which is a *crawlster.helpers.RequestsHelper* instance.

4.2.3 Parsing requests and extracting data

We can parse the response data (or basically any string or bytes sequences) using the core `.extract` helper (*crawlster.helpers.ExtractHelper*)

4.2.4 Submitting results

Submitting results is done via the *crawlster.Crawlster.submit_item()* method. The single argument must be a `dict` that represents the item.

After being submitted, the item will be passed through all the defined item handlers.

See also:

The module reference for *crawlster.handlers* for more details and all the available item handler classes.

4.2.5 Extending the crawler with helpers

The `crawlster` library makes very easy to extend the functionality of the crawler through helpers. A helper is only a utility class that is attached to the crawler instance.

Core helpers:

- *crawlster.helpers.RequestsHelper* available as `http`.

- `crawlster.helpers.UrlsHelper` available as `urls`.
- *`crawlster.helpers.ExtractHelper`* available as `extract`.
- `crawlster.helpers.StatsHelper` available as `stats`.
- `crawlster.helpers.LoggingHelper` available as `log`.
- `crawlster.helpers.QueueHelper` available as `queue`.
- `crawlster.helpers.RegexHelper` available as `regex`.

Create your own helper

In order to create your own helper to enhance your crawler with super powers you need to subclass the `crawlster.helpers.BaseHelper` base class.

Then you can start implementing the functionality you need.

Methods

There is no required method that has to be overwritten, but there are some methods that can be overwritten to act as hooks. So far the only two available hooks are

- `crawlster.helpers.BaseHelper.initialize()` that performs actions on crawler start.
- `crawlster.helpers.BaseHelper.finalize()` that performs actions on crawler stop (when there are no more items to process).

Configuration

Helpers can take advantage of the configuration system the library provides by providing the `config_options` attribute, a mapping of option name and option value.

Attributes

The two attributes that are available inside the helper are `config` and `crawler`.

The `config` attribute will hold the `Configuration` instance used to initialize the crawler. You can get values from the configuration using the `self.config.get(option_name)` method.

The `crawler` attribute holds the current crawler instance through which the helper can access other helpers. Although it is recommended to make the helper as independent as possible, sometimes you would need to use the functionality already provided by some already existent helper (stats aggregation, logging, etc).

Attaching the helper to the crawler

In the crawler definition, provide the helper instance as a class attribute

```
class MyCrawler(Crawlster):

    my_helper = MyHelperClass()

# ...

def some_step(self, url):
```

```
    # ...
    self.my_helper.do_amazing_things()
    # ...
```

4.3 Module reference

4.3.1 The crawlster module

class crawlster.**Crawlster**(*config=None*)
Base class for web crawlers

Any crawler must subclass this and provide a valid Configuration object as the config class attribute.

finalize()
Performs the finalize action on all item handlers and helpers

get_pool()
Creates and returns the worker pool

init_context()
Initializes the crawler context (the queue and the worker pool)

inject_config_and_crawler(*to_be_injected*)
Injects the config instance and crawler instance into the object

The crawler instance will be accessible through the .crawler attribute, the config instance will be accessible through the .config attribute. After injection, the .initialize() is called to perform the init actions.

Parameters **to_be_injected** (*object which has initialize()*) – An object in which will be injected the config and crawler attributes. Must have an .initialize() method

inject_handlers()
Injects and initializes all the known item handlers

inject_helpers()
Injects and initializes all the known helpers

iter_helpers()
Iterates through all the item handlers

iter_item_handlers()
Iterates through all the known item handlers

populate_config()
Populates the config with the options from helpers and item handlers

Each helper and item handler defines a list of options that it uses. This method will visit each helper and item handler to populate the config instance with those options.

process_job(*job*)
Processes a single job and enqueues the results

report_error(*e*, *failed_job*)
Reports a failed job

Parameters

- **e** (*Exception*) – The exception instance that was thrown
- **failed_job** (*Job*) – The job instance that caused the exception

schedule(*func*, **args*, ***kwargs*)
Schedules the next tep to be executed by workers

start()
Starts crawling based on the config

submit_item(*item*)
Submit an item to be handled by the item handlers

Parameters **item** (*dict*) – The item that has to be processed

worker()
Worker body that executes the jobs

crawlster.**start**(*method*)
Decorator for specifying the start step.

Must decorate a single method from the crawler class

4.3.2 The crawlster.config module

Configuration classes

class crawlster.config.**Configuration**(*options=None*)
Configuration object that stores key-value pairs of options

class crawlster.config.**JsonConfiguration**(*file_path*)
Reads the configuration from a json file

Configuration options

class crawlster.config.**ConfigOption**(*validators*, *default=None*, *required=False*)
Class for configuration option definitions

class crawlster.config.**Required**(*validators*)
A required value that the user must provide.

class crawlster.config.**NumberOption**(*default=None*, *required=False*, *extra_validators=None*)
A numeric option

class crawlster.config.**StringOption**(*default=None*, *required=False*, *extra_validators=None*)
A string option

class crawlster.config.**ListOption**(*default=None*, *required=False*, *extra_validators=None*)
A list/tuple option

class crawlster.config.**ChoiceOption**(*choices*, *default=None*, *required=False*, *extra_validators=None*)
An option whose value must be one from the specified choices

class crawlster.config.**UrlOption**(*default=None*, *required=False*, *extra_validators=None*)
An option whose value must be a valid URL

4.3.3 The crawlster.exceptions module

exception crawlster.exceptions.**ConfigurationError**
Thrown when configuration is invalid

exception crawlster.exceptions.**CrawlsterError**
Base exception for application specific exceptions

exception crawlster.exceptions.**MissingValueError**
Thrown when a required config value is not provided

exception crawlster.exceptions.**OptionNotDefinedError**
Thrown when trying to access an option that is not defined

4.3.4 The crawlster.validators module

This module contains various validators.

They are used mainly in the config options definitions.

class crawlster.validators.**ValidateIsInstance**(*req_type*)
Validates that a value is of certain type

exception crawlster.validators.**ValidationError**
Thrown when validation fails

crawlster.validators.**is_url**(*value*)
Validates that the value represents a valid URL

crawlster.validators.**one_of**(*choices*)
Validates that an instance is one of the specified values

crawlster.validators.**validate_isinstance**
alias of *ValidateIsInstance*

4.3.5 Helpers

Http helpers

class crawlster.helpers.**RequestsHelper**
Helper for making HTTP requests using the requests library

delete(*url*, *data=None*, *query_params=None*, *headers=None*)
Makes a DELETE request

get(*url*, *query_params=None*, *headers=None*)
Makes a GET request

initialize()
Initializes the session used for making requests

open(*http_request: crawlster.helpers.http.request.HttpRequest*)
Opens a given HTTP request.

Parameters **http_request** (HttpRequest) – The crawlster.helpers.http.request.HttpRequest instance with the required info for making the request

Returns crawlster.helpers.http.response.HttpResponse

options(*url*, *query_params=None*, *headers=None*)
Makes an OPTIONS request

patch(*url*, *data=None*, *query_params=None*, *headers=None*)
Makes a PATCH request

post(*url*, *data=None*, *query_params=None*, *headers=None*)
Makes a POST request

Http requests

class crawlster.helpers.http.request.**HttpRequest**(*url*, *method='GET'*, *data=None*, *query_params=None*, *headers=None*)
Class representing a http request

class crawlster.helpers.http.request.**GetRequest**(*url*, *query_params=None*, *headers=None*)
A HTTP GET request

class crawlster.helpers.http.request.**PostRequest**(*url*, *data=None*, *query_params=None*, *headers=None*)
A HTTP POST request

class crawlster.helpers.http.request.**JsonRequest**(*url*, *method='GET'*, *data=None*, *query_params=None*, *headers=None*)
A generic JSON request.

The data must be an object that can be safely encoded as JSON.

Examples

JsonRequest('http://example.com', 'POST', data={'hello': 'world'})

class crawlster.helpers.http.request.**XhrRequest**(*url*, *method='GET'*, *data=None*, *query_params=None*, *headers=None*)
A XHR Post request

Http responses

class crawlster.helpers.http.response.**HttpResponse**(*request*, *status_code*, *headers*, *body*)
Class representing a http response

body_str
Returns the decoded content of the request, if possible.

May raise UnicodeDecodeError if the body does not represent a valid unicode encoded sequence.

content_type
Returns the response content type if available

server
Returns the server header if available

Extract helpers

class crawlster.helpers.**ExtractHelper**

css(*text*, *selector*, *attr=None*, *content=None*)
Extracts data using css selector.

See :py:meth:`Content.css` for more info.

Utility classes

class crawlster.helpers.extract.**Content**(*raw_data*)
Content wrapper that provides common data extraction methods

css(*pattern*, *get_attr=None*, *get_text=False*)
Extracts data using css selector

Returns a list of elements (as strings) with the extracted data

Parameters

- **pattern** (*str*) – the CSS selector
- **get_attr** (*str or None*) – if present, returns a list of the attributes of the extracted items
- **get_text** (*bool*) – If should return only the content/text of the element

Returns

If get_attr and get_text are not specified, returns a list of strings with the matches.

If get_attr is specified, returns a list with the values of the specified attribute, if present. Elements that match the query pattern and does not have that attribute are ignored.

If get_text is specified, returns a list with the text from the matched elements (direct children that are not nested tags).

parsed_data
Access the underlying bs4.BeautifulSoup4 instance

This property is provided for more advanced usage.

4.3.6 Item handlers

Base class

class crawlster.handlers.**BaseItemHandler**

Basic item handlers

class crawlster.handlers.**JsonLinesHandler**(*filename*)

class crawlster.handlers.**LogItemHandler**(*level='warning'*)
Item handler that logs the submitted items via the crawler logger

class crawlster.handlers.**StreamItemHandler**(*stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

Database item handlers

4.4 Contributing

In order to contribute, please do as follows:

- fork the repository
- switch on a branch whose name will reflect what you are working on
- when finished, run the tests (`make test`)
- if everything is alright, open a pull request.

Don't forget to:

- run the tests!
- check that the documentation builds (`make docs`) without errors
- write tests for the added features or bugs fixed.
- write in a few words in the changelog, under the `current version` subsection the change you just made.

CHAPTER 5

Changelog

5.1 Current version

- Work in progress

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

R

S

U

V

W

X