crawly Documentation

Release 0.1b

Mouad Benchchaoui

October 25, 2012

CONTENTS

Crawly is a Python library that allow to crawl website and extract data from this later using a simple API.

Crawly work by combining different tool, that ultimately created a small library (~350 lines of code) that fetch website HTML, crawl it (follow links) and extract data from each page.

Libraries used:

- requests It's a Python HTTP library, it's used by **crawly** to fetch website HTML, this library take care of maintaining the Connection Pool, it's also easily configurable and support a lot of feature including: SSL, Cookies, Persistent requests, HTML decoding ... .
- gevent This is the engine responsible of the speed in crawly, with gevent you can run concurrent code, using green thread.
- lxml a fast, easy to use Python library that used to parse the HTML fetched to help extracting data easily.
- logging Python standard library module that log information, also easily configurable.

CHAPTER

ONE

USER GUIDE:

1.1 Installation

This part of the documentation covers the installation of Crawly.

1.1.1 From Pypi (stable):

Installing crawly is simple with pip:

```
$ pip install crawly
```

or, with easy_install:

```
$ easy_install crawly
```

1.1.2 From repository (Unstable):

You can download crawly.py into your project directory.

or you can install crawly from source code:

```
$ hg clone https://bitbucket.org/mouad/crawly
$ cd crawly
$ python setup.py install
```

1.2 API

This part of the documentation covers all the interfaces of crawly.

1.2.1 Runner:

This class is not offered as a public interface, instead users should use `runner` module attribute that is an instance of `_Runner`.

class `crawly._Runner`
Class to manage running all requests concurrently and extracting data from the website and writing them back to pipelines.

add_pipeline(*pipeline*)
Add a pipeline which is a callable that accept a `WebPage` class or subclass instance, which will be passed after extracting all the data instructed.

Return: `self` to allow "Fluent Interface" creation pattern.

fetch(*request*)
Execute send `request` in a greenlet from the pool of requests.

filter(*predicate*)
Add a predicate to filter pages (URLs) to include only the ones with which the predicate return True.

The difference between this method and `_Runner.takewhile()` is that `_Runner.filter()` method allow only to filter individual URLs while `_Runner.takewhile()` will stop at a given URL when the predicate return False and all URLs which come after this last URL will not be crawled.

Return: `self` to allow "Fluent Interface" creation pattern.

log(*msg*, *level=20*)
Log a message under `level`, default to INFO.

on_exception(*func*)
Add a function to be executed when the crawler find an exception.

Argument: func: A function that should accept one arguments, that will be the greenlet that raised the exception.

Return: `self` to allow "Fluent Interface" creation pattern.

on_finish(*func*)
Add a function to be executed when the crawler finish crawling and all the greenlet has been joined.

Argument: func: A function that should accept no arguments.

Return: `self` to allow "Fluent Interface" creation pattern.

report
Get execution report.

The report contains the following fields:

- CRAWLED URLS: count of crawled URLs.
- EXTRACTED DATA: count of extracted data passed to pipelines.
- EXCEPTIONS COUNTER: count number of exceptions raised.
- START TIME: Date time when the crawler started.
- FINISH TIME: Date time when the crawler finished.
- TOTAL TIME: The total time spend crawling.
- SHUTDOWN REASON: Reason why the crawler finished, i.e. show the exception that made the crawler stop if there is one, else show 'FINISH CRAWLING' which mean the crawler finish normally.

set_website(*website*)
Set the website to crawl, the `website` argument can be an instance or a class that inherit from `WebSite` class.

Return: `self` to allow Fluent Interface creation pattern.

start(*argv=None*)
Start/Launch crawling.

Argument: argv: Command line arguments, default to `sys.argv[1:]`.

Command line argument:

-config=file.json

The file.json configuration file should be in JSON format which will replace default configuration that is taken from the *global configuration*.

takewhile(*predicate*)

Add a predicate that will stop adding URLs to fetch when the predicate will return False.

Argument: predicate: A function that accept a page as an argument and return a boolean; when the predicate return False all URLs after this one in the website will not be fetched.

Return: `self` to allow "Fluent Interface" creation pattern.

WARNING: The page when passed to the `predicate` is not fetched yet, so no data is extracted from this page yet.

Website structures:

class crawly.**WebSite**

An abstract super class that represent a website.

Class inheriting from this class should implement the `url` class variable, else this class will raise an Exception.

Examples

```
>>> class PythonQuestions(WebSite):
...     url = "http://stackoverflow.com/question/tagged/python"
...     Pagination = Pagination(
...         'http://stackoverflow.com/questions/tagged/python',
...         data={'page': '{page}'},
...         end=4
...     )
...
>>> [page.url for page in PythonQuestions().pages]
['<GET http://stackoverflow.com/questions/tagged/python?page=1>',
 '<GET http://stackoverflow.com/questions/tagged/python?page=2>',
 '<GET http://stackoverflow.com/questions/tagged/python?page=3>',
 '<GET http://stackoverflow.com/questions/tagged/python?page=4>']
```

WebPageCls

alias of `WebPage`

pages

Get pages from the website.

If `WebSite.Pagination` class variable was set, this return a list of pages yield by the pagination, else it return a list with a single element which is a `WebPage` instance of this url.

class crawly.**Pagination**(*url*, *data*, *method='GET'*, *start=1*, *end=None*)

Class that iterate over a website pages and return a request for each one of them.

Arguments:

- url: Pagination url.
- data: Dictionary of data to send with URL to get the next page, you can pass the string template `{page}` as a value of a dictionary key, which will be replaced by the exact page value before sending the request.
- method: HTTP method to use to request the url, default: GET.

- start: Page number to start requesting from included, default: 1.
- end: Last pagination's page number **included**, default to `None` in this case developers must override the `end_reached` method to be able to stop somewhere.

Example

```
>>> stackoverflow_pages = Pagination(
...     'http://stackoverflow.com/questions/tagged/python',
...     data={'page': '{page}'},
...     end=4
... )
>>> [r.pretty_url for r in stackoverflow_pages]
['<GET http://stackoverflow.com/questions/tagged/python?page=1>',
 '<GET http://stackoverflow.com/questions/tagged/python?page=2>',
 '<GET http://stackoverflow.com/questions/tagged/python?page=3>',
 '<GET http://stackoverflow.com/questions/tagged/python?page=4>']
```

end_reached()
Method meant to be overrided to stop iterating over pagination if `end` constructor argument wasn't set.

Return True to stop paginating else False.

next()
Get the next page request.

class `crawly.`**WebPage**(*url_or_request*, *parent=None*, *initial=None*)
Class that represent a WEB site page that can be used to extract data or extract links to follow.

Extract data from the page

```
>>> class PythonJobs(WebPage):
...     toextract = {
...         'title': '//div[5]/div/div/div[2]/h2/a/text()'
...     }
...
>>> page = PythonJobs('http://www.python.org/community/jobs/')
>>> page.extract()
{'title': ...}
```

Extract links to follow

```
>>> class PythonJobs(WebPage):
...     tofollow = '//div[5]/div/div/div[2]/h2/a/@href'
...
>>> page = PythonJobs('http://www.python.org/community/jobs/')
>>> list(page.follow_links())
[...]
```

Arguments:

- url_or_request: This argument can be a string representing the URL of this page or for better customizing it can be also a request.
- parent: A `WebPage` or a `WebSite` instance that represent the parent site/page of this one.
- initial: Initial data related to this page.

data
Get extracted data.

WARNING: This property will recalculate each time the data to return when it's accessed, so be careful about side effect, what i mean by that is if you override this method and for example the new method define a new value that change in each call e.g. `datetime.now()`, than you will have inconsistency in your data. In this case and if inconsistency is a problem, developers should use `WebPage._getdata()` method instead to define any extra data, which is computed only the first time this property is accessed.

extract (*toextract=None*, *update=True*)
Extract the data given by `toextract`.

Argument:

- toextract: same argument accepted by `HTML.extract()` method.
- update: Boolean that enable updating the internal data holder when it's set to True (default) else it will return extracted data w/o updating internal data holder.

Return: Extracted data.

Raise:

- `ExtractionError` if extraction failed.
- `ValueError` if the argument didn't follow the documentation guidline.

follow_links (*tofollow=None*)
Follow the links given and return a `WebPage.WebPageCls` instance for each link.

Argument: tofollow: same argument accepted by `HTML.extract()` method, if `tofollow` is a dictionary it must contain a key "links" which should point to the path to use to extract URLs to follow and any extra keys in the dictionary will be used to extract extra data that **must be of the same length as the URLs to follow** and this data will be passed to the generated `WebPageCls` instances.

Return: Generate a list of `WebPageCls` instances for each link to follow.

Raise:

- `ExtractionError` if extraction failed.
- `ValueError` if the argument didn't follow the documentation guideline.

html
Get the HTML of this page as a `HTML` class instance.

request
Get the request used by this page.

url
Get a pretty URL of this page in the form <(method: data) url>.

class `crawly.`**HTML** (*html*)
Class to represent HTML code.

This class is a wrapper around `lxml.html.HtmlElement` class, so developers can interact with instance of this class in the same way you do with `lxml.html.HtmlElement` instances, with the addition that this class define a new method `HTML.extract()` that allow extracting data from the html.

Example

```
>>> html = HTML('<html><body><div><h2>test</h2></div></body></html>')
>>> html.extract('//div/h2/text()')
'test'
```

extract (*extractor*)
Extract from this HTML the data pointed by `extractor`.

Argument: extractor: Can be a dictionary in the form `{'name': <callable> or <string>}`, or unique callable object that accept a `lxml.html.HtmlElement` e.g. `XPath` class instance or a string which in this case the string will be automatically transformed to an `XPath` instance.

Return: The extracted data in the form of a dictionary if the `extractor` argument given was a dictionary else it return a list or string depending on the extractor callbacks.

Raise: `ExtractionError` if extraction failed.

Extraction tools:

class `crawly.`**`XPath`**(*xpath*, **callbacks*)

Callable class that define XPATH query with callbacks.

Arguments:

- xpath: A string representing the XPath query.
- callbacks: A list of functions to call in order (first to last) over the result returned by `lxml.etree.XPath`, this class have also a `callbacks` class variable that can be set by subclasses which have priority over the callbaks passed in this argument, which mean that callbacks passed here will be called after the class variable callbacks.

  Illustration

  ```
  XPath("...", callback1, callback2, callback3)
      <=>
  callback3( callback2( callback1( XPath("...") ) ) )
  ```

Raise: `ExtractionError` if extraction failed.

Example

```
>>> import string

>>> x = XPath('//div/h2/text()', string.strip)
>>> x('<html><body><div><h2>\r\ntest\n</h2></div></body></html>')
'test'

>>> x = XPath('//ul/li/text()', lambda ls: map(int, ls))
>>> x('<html><body><ul><li>1</li><li>2</li></ul></body></html>')
[1, 2]
```

Exceptions:

exception `crawly.`**`ExtractionError`**

Error raised when extracting data from HTML fail.

Configuration:

Crawly can be configured by passing a JSON formatted file in the `--config` command line option that will override the default configuration, which is a combinaison of requests configuration and logging configuration.

```
{
    'timeout': 15,
    # Requests configuration: http://tinyurl.com/dyvdj57
    'requests': {
```

```
        'base_headers': {
            'Accept': '*/*',
            'Accept-Encoding': 'identity, deflate, compress, gzip',
            'User-Agent': 'crawly/' + __version__
        },
        'danger_mode': False,
        'encode_uri': True,
        'keep_alive': True,
        'max_redirects': 30,
        'max_retries': 3,
        'pool_connections': 10,
        'pool_maxsize': 10,
        'safe_mode': True,   # Default in False in requests.
        'strict_mode': False,
        'trust_env': True,
        'verbose': False
    },
    # Logging configuration: http://tinyurl.com/crt6rkw
    'logging': {
        'version': 1,
        'formatters': {
            'standard': {
                'format': '%(asctime)s [%(levelname)s] %(name)s: %(message)s'
            }
        },
        'handlers': {
            'console': {
                'formatter': 'standard',
                'class': 'logging.StreamHandler',
            }
        },
        'loggers': {
            '': {
                'handlers': ['console'],
                'level': 'DEBUG',
                'propagate': False,
            }
        }
    }
}
```

1.3 Examples

I invite to check the list of example in the repository that can give you real world example of spiders that are used to scrape data from website.

Here is an example of a spider for StackOverFlow website which should be used like this:

```
$ python stackoverflow.py --config=config.json
```

```
# -*- encoding: utf8 -*-
"""Crawler for Stackoverflow website.

http://stackoverflow.com/questions/tagged/python

The Stackoverflow represent a perfect example of what most website look like,
which is a:
```

```
    * A **list** page(s) that contain a summary of all items.
    * The list page(s) are divided in multiple page using a **pagination**.
    * Each item in the list page(s) contain a link for a **detail** page that
      contain extra data.

So using crawly you can structure your crawler in the given form.

    1) A ''WebSite'' subclass that define the front structure of the website,
    which is : Pagination, ListPage
    2) The ListPage is a subclass of a ''WebPage'' which define links
    ''tofollow'' to get DetailPage and data ''toextract'' from the list page.
    3) The DetailPage is the final page which contain the data ''toextract''.

"""
import sys; sys.path.append('..')  # Include crawly in system path.

import json

import requests
from gevent.coros import Semaphore

from crawly import XPath, HTML, WebSite, WebPage, Pagination, runner

# If this was a real crawler for StackOverFlow it will be better if we
# changed the 'pagesize' parameter in the url to 50 (maximum) to make
# the least possible queries.
URL = "http://stackoverflow.com/questions/tagged/python?&sort=newest&pagesize=15"
FILENAME = 'questions.json'  # File used to dump crawled pages.
NEW_DATA = {}                # Hold new extracted data.
LOCK = Semaphore()           # Synchronize write to NEW_DATA.
try:
    data = json.load(open(FILENAME))
except IOError:              # First time run, file doesn't exist yet.
    CRAWLED = {}
else:
    CRAWLED = set(data)      # Get a list of crawled URLs.
    del data


def _get_end():
    "Get last page to crawl."
    response = requests.get(URL)
    last = int(
        HTML(response.content).extract(
            '//span[@class="page-numbers"]/text()'
        )[-1]  # -1 for last page.
    )
    runner.log('Number of pages detected is: %d' % last)
    return 1  # XXX: You can ''return last'' to crawl all the website.


class QuestionPage(WebPage):

    # All data extracted here can be extracted from the list page but
    # i preferred to do it here to show how you can use crawly to follow
    # links and extract data from this later.
    toextract = {
        'title': '//div[@id="question-header"]/h1/a/text()',
```

```
        'user_name': '//div[@id="question"]//div[@class="user-details"]/a/text()',
        'datetime': '//div[@id="question"]//div[@class="user-action-time"]/span/@title',
        'tags': '//div[@id="question"]//a[@class="post-tag"]/text()',
        'accepted': XPath(
            '//span[starts-with(@class, "vote-accepted-on")]',
            bool
        )
    }


class ListPage(WebPage):

    tofollow = {
        'links': '//div[@id="questions"]//a[@class="question-hyperlink"]/@href',
        'vote': '//span[@class="vote-count-post"]/strong/text()',
        'answers_count': '//div[@class="question-summary"]//div[starts-with(@class, "status")]/strong
    }
    WebPageCls = QuestionPage


class StackOverFlow(WebSite):

    url = URL
    Pagination = Pagination(
        URL,
        data={'page': '{page}'},
        end=_get_end()
    )
    WebPageCls = ListPage


def isnew(page):
    "Check that the url wasn't already crawled."
    # I am assuming that already crawled question don't change, and because
    # this spider crawl question in newest to oldest, so when ever crawly see
    # an URL that was already crawled, this mean that all URLs that will follow
    # was crawled too, so better to stop here.
    return page.url not in CRAWLED


def save(page):
    "Save extracted page in a list."
    # This function is run in a greenlet (b/c it's used as crawly pipeline) so
    # that explain why we are using a Lock here.
    LOCK.acquire()
    try:
        NEW_DATA[page.url] = page.data
    finally:
        LOCK.release()


def tojson():
    "Write extracted data in the JSON format to a file."
    old = {}
    try:
        old = json.load(open(FILENAME))
    except IOError:
        pass
```

```
    old.update(NEW_DATA)
    json.dump(old, open(FILENAME, 'w'), indent=4)
    runner.log('Dump all questions')


if __name__ == '__main__':
    runner.set_website(StackOverFlow).takewhile(isnew) \
          .add_pipeline(save).on_finish(tojson).start()
```

The example of configuration file (`config.json`) that instruct logging to log to console and a file.

```
{
    "logging": {
        "version": 1,
        "formatters": {
            "standard": {
                "format": "%(asctime)s [%(levelname)s] %(name)s: %(message)s"
            }
        },
        "handlers": {
            "console": {
                "class": "logging.StreamHandler",
                "formatter": "standard",
                "stream": "ext://sys.stdout"
            },
            "file": {
                "class": "logging.handlers.RotatingFileHandler",
                "formatter": "standard",
                "filename": "/tmp/crawly.log",
                "maxBytes": 1000000,
                "backupCount": 3
            }
        },
        "root": {
            "handlers": ["console", "file"],
            "level": "INFO"
        }
    }
}
```

1.4 FAQ

1.4.1 Existing study: Why you shouldn't use Crawly ?

First of all, have you checked scrapy (http://scrapy.org) ? if not, you should, it's a very powerful framework, but in my case and unfortunately i have found some drawbacks with Scrapy which lead me to create **Crawly**, which are:

- Scrapy was too big and too hard to hack in, as i had some problems with it, especially concerning consistency of scraped data which is a huge problem when it came to scapers, but a lot of spaghetti code make it also very hard to dig in :).
- Most website just look the same (at least the one that i crawled) but scrapy didn't help make my code clean because of a lot of boilerplate.
- Scrapy is huge in term of architecture (scrapyd, web interface, ...), and all of this was consuming a lot of memory and my little server wasn't able to support, so it was crushing other process each time scrapy start crawling.

1.4.2 Define the need: Why I have created Crawly ?

Because i love micro-frameworks (Flask VS Django) and because i believe that

```
Inside every large, complex program is a small, elegant program that does the same thing, correctly
```

And because i wanted to fix all the problems listed above without having to dig in Scrapy, and when i estimated the cost of digging into scrapy and the cost of me creating a new crawler library and what i will gain, well guess what ?!

1.4.3 Goals: What should a crawler library do ?

IMHO, a crawler library should (**not** in order of importance):

- Simple Usage: Make it easy to instruct the library to crawl a given website, by handling most common pattern existing for website design, for example: single page, list->detail, paginate->list->detail and such, and make it easy to extend for special website.
- Feedback: Log everything to user.
- Configurable: Something that all library should offer.
- Encoding: Handle all HTML encoding (utf8, latin1 ...).
- Scraping: Give developer easy way to extract data from a website, using XPath for example.
- Speed: crawling a website should be fast.
- Play nice: by handling rate limits, so we don't DOS the servers.

1.4.4 Status: How is Crawly compared to Scrapy ?

- Speed: In term of speed i can tell you with confidence that Crawly is very fast and that all thanks to Gevent. Most of the times in my tests i remarked that Crawly was a **little bit faster** than Scrapy, but nothing very noticeable (few seconds of difference) because Scrapy is already very fast.
- Memory: Well Crawly is small so in term of memory and it's very light.
- Usage/Simplicity: Well i may be a little biased on this one, but that was on of the main reason for me to create Crawly.
- Features: For the mean time Scrapy has a lot of feature that don't have a match in Crawly.
- Maturity: Crawly is still new at this stage while Scrapy is very mature Open Source project so nothing to compare here :)

PYTHON MODULE INDEX