scrapekit Documentation

Release 0.1

Friedrich Lindenberg

July 06, 2015

Contents

Many web sites expose a great amount of data, and scraping it can help you build useful tools, services and analysis on top of that data. This can often be done with a simple Python script, using few external libraries.

As your script grows, however, you will want to add more advanced features, such as **caching** of the downloaded pages, **multi-threading** to fetch many pieces of content at once, and **logging** to get a clear sense of which data failed to parse.

Scrapekit provides a set of useful tools for these that help with these tasks, while also offering you simple ways to structure your scraper. This helps you to produce **fast, reliable and structured scraper scripts**.

CHAPTER 1

Example

Below is a simple scraper for postings on Craigslist. This will use multiple threads and request caching by default.

```
import scrapekit
from urlparse import urljoin

scraper = scrapekit.Scraper('craigslist-sf-boats')

@scraper.task
def scrape_listing(url):
    doc = scraper.get(url).html()
    print(doc.find('.//h2[@class="postingtitle"]').text_content())


@scraper.task
def scrape_index(url):
    doc = scraper.get(url).html()

    for listing in doc.findall('.//a[@class="hdrlnk"]'):
        listing_url = urljoin(url, listing.get('href'))
        scrape_listing.queue(listing_url)

scrape_index.run('https://sfbay.craigslist.org/boo/')
```

By default, this save cache data to a the working directory, in a folder called `data`.

CHAPTER 2

Reporting

Upon completion, the scraper will also generate an HTML report that presents information about each task run within the scraper.

This behaviour can be disabled by passing `report=False` to the constructor of the scraper.

CHAPTER 3

Contents

3.1 Installation Guide

The easiest way is to install `scrapekit` via the Python Package Index using `pip` or `easy_install`:

```
$ pip install scrapekit
```

To install it manually simply download the repository from Github:

```
$ git clone git://github.com/pudo/scrapekit.git
$ cd scrapekit/
$ python setup.py install
```

3.2 Quickstart

Welcome to the scrapekit quickstart tutorial. In the following section, I'll show you how to write a simple scraper using the functions in scrapekit.

Like many people, I've had a life-long, hidden desire to become a sail boat captain. To help me live the dream, we'll start by scraping Craigslist boat sales in San Francisco.

3.2.1 Getting started

First, let's make a simple Python module, e.g. in a file called `scrape_boats.py`.

```
import scrapekit

scraper = scrapekit.Scraper('craigslist-sf-boats')
```

The first thing we've done is to instantiate a scraper and to give it a name. The name will later be used to configure the scraper and to read it's log ouput. Next, let's scrape our first page:

```
from urlparse import urljoin

@scraper.task
def scrape_index(url):
    doc = scraper.get(url).html()

    next_link = doc.find('.//a[@class="button next"]')
    if next_link is not None:
```

```
        # make an absolute url.
        next_url = urljoin(url, next_link.get('href'))
        scrape_index.queue(next_url)

scrape_index.run('https://sfbay.craigslist.org/boo/')
```

This code will cycle through all the pages of listings, as long as a *Next* link is present.

The key aspect of this snippet is the notion of a *task*. Each scrapekit scraper is broken up into many small tasks, ideally one for fetching each web page.

Tasks are executed in parallel to speed up the scraper. To do that, task functions aren't called directly, but by placing them on a queue (see *scrape_index.queue* above). Like normal functions, they can still receive arguments - in this case, the URL to be scraped.

At the end of the snippet, we're calling *scrape_index.run*. Unlike a simple queueing operation, this will tell the scraper to queue a task and then wait for all tasks to be executed.

3.2.2 Scraping details

Now that we have a basic task to scrape the index of listings, we might want to download each listing's page and get some data from it. To do this, we can extend our previous script:

```
import scrapekit
from urlparse import urljoin

scraper = scrapekit.Scraper('craigslist-sf-boats')

@scraper.task
def scrape_listing(url):
    doc = scraper.get(url).html()
    print(doc.find('.//h2[@class="postingtitle"]').text_content())


@scraper.task
def scrape_index(url):
    doc = scraper.get(url).html()

    for listing in doc.findall('.//a[@class="hdrlnk"]'):
        listing_url = urljoin(url, listing.get('href'))
        scrape_listing.queue(listing_url)

    next_link = doc.find('.//a[@class="button next"]')
    if next_link is not None:
        # make an absolute url.
        next_url = urljoin(url, next_link.get('href'))
        scrape_index.queue(next_url)

scrape_index.run('https://sfbay.craigslist.org/boo/')
```

This basic scraper could be extended to extract more information from each listing page, and to save that information to a set of files or to a database.

3.2.3 Configuring the scraper

As you may have noticed, Craigslist is sometimes a bit slow. You might want to configure your scraper to use caching, or a different number of simultaneous threads to retrieve data. The simplest way to set up caching is to set some

environment variables:

```
$ export SCRAPEKIT_CACHE_POLICY="http"
$ export SCRAPEKIT_DATA_PATH="data"
$ export SCRAPEKIT_THREADS=10
```

This will instruct scrapekit to cache requests according to the rules of HTTP (using headers like `Cache-Control` to determine what to cache and for how long), and to save downloaded data in a directory called `data` in the current working path. We've also instructed the tool to use 10 threads when scraping data.

If you wanto to make these decisions at run-time, you could also pass them into the constructor of your *`Scraper`*:

```
import scrapekit

config = {
  'threads': 10,
  'cache_policy': 'http',
  'data_path': 'data'
}
scraper = scrapekit.Scraper('demo', config=config)
```

For details on all available settings and their meaning, check out the configuration documentation.

3.3 Using tasks

Tasks are used by scrapekit to break up a complex script into small units of work which can be executed asynchronously. When needed, they can also be composed in a variety of ways to generate complex data processing pipelines.

3.3.1 Explicit queueing

The most simple way of using tasks is by explicitly queueing them. Here's an example of a task queueing another task a few times:

```
import scrapekit

scraper = scrapekit.Scraper('test')

@scraper.task
def each_item(item):
    print(item)

@scraper.task
def generate_work():
    for i in xrange(100):
        each_item.queue(i)

if __name__ == '__main__':
    generate_work.queue().wait()
```

As you can see, `generate_work` will call `each_item` for each item in the range. Since the items are processed asynchronously, the printed output will not be in order, but slightly mixed up.

You can also see that on the last line, we're queueing the `generate_work` task itself, and then instructing scrapekit to wait for the completion of all tasks. Since the double call is a bit awkward, there's a helper function to make both calls at once:

```
if __name__ == '__main__':
    generate_work.run()
```

3.3.2 Task chaining and piping

As an alternative to these explicit instructions to queue, you can also use a more pythonic model to declare processing pipelines. A processing pipeline connects tasks by feeding the output of one task to another task.

To connect tasks, there are two methods: chaining and piping. Chaining will just take the return value of one task, and queue another task to process it. Piping, on the other hand, will expect the return value of the first task to be an iterable, or for the task itself to be a generator. It will then initiate the next task for each item in the sequence.

Let's assume we have these functions defined:

```
import scrapekit

scraper = scrapekit.Scraper('test')

@scraper.task
def consume_item(item):
    print(item)

@scraper.task
def process_item(item):
    return item ** 3

@scraper.task
def generate_items():
    for i in xrange(100):
        yield i
```

The simplest link we could do would be this simple chaining:

```
pipline = process_item > consume_item
pipeline.run(5)
```

This linked `process_item` to `consume_item`. Similarly, we could use a very simple pipe:

```
pipline = generate_items | consume_item
pipeline.run()
```

Finally, we can link all of the functions together:

```
pipline = generate_items | process_item > consume_item
pipeline.run()
```

3.4 Caching

Caching of response data is implemented via CacheControl, a library that extends requests. To enable a flexible usage of the caching mechanism, the use of cached data is steered through a cache policy, which can be specified either for the whole scraper or for a specific request.

The following policies are supported:

- `http` will perform response caching and validation according to HTTP semantic, i.e. in the way that a browser would do it. This requires the server to set accurate cache control headers - which many applications are too stupid to do.
- `none` will disable caching entirely and always revert to the server for up-to-date information.
- `force` will always use the cached data and not check with the server for updated pages. This is useful in debug mode, but dangerous when used in production.

3.4.1 Per-request cache settings

While caching will usually be configured on a scraper-wide basis, it can also be set for individual (`GET`) requests by passing a `cache` argument set to one of the policy names:

```
import scrapekit
scraper = scrapekit.Scraper('example')

# No caching:
scraper.get('http://google.com', cache='none')

# Cache according to HTTP semantics:
scraper.get('http://google.com', cache='http')

# Force re-use of data, even if it is stale:
scraper.get('http://google.com', cache='force')
```

3.5 Utility functions

These helper functions are intended to support everyday tasks of data scraping, such as string sanitization, and validation.

`scrapekit.util.`**`collapse_whitespace`**(*text*)
Collapse all consecutive whitespace, newlines and tabs in a string into single whitespaces, and strip the outer whitespace. This will also accept an `lxml` element and extract all text.

3.6 Configuration

Scrapekit supports a broad range of configuration options, which can be set either via a configuration file, environment variables or programmatically at run-time.

3.6.1 Configuration methods

As a first source of settings, scrapekit will attempt to read a per-user configuration file, `~/.scrapekit.ini`. Inside the ini file, two sections will be read: `[scrapekit]` is expected to hold general settings, while a section, named after the current scraper, can be used to adapt these settings:

```
[scrapekit]
reports_path = /var/www/scrapers/reports

[craigslist-sf-boats]
threads = 5
```

After evaluating these settings, environment variables will be read (see below for their names). Finally, all of these settings will be overridden by any configuration provided to the constructor of *Scraper*.

3.6.2 Available settings

Name	Environment variable	Description
threads	SCRAPEKIT_THREADS	Number of threads to be started.
cache_policy	SCRAPEKIT_CACHE_POLICY	Policy for caching requests. Valid values are `disable` (no caching), `http` (cache according to HTTP header semantics) and `force`, to force local storage and re-use of any requests.
data_path	SCRAPEKIT_DATA_PATH	A storage directory for cached data from HTTP requests. This is set to be a temporary directory by default, which means caching will not work.
reports_path	SCRAPEKIT_REPORTS_PATH	A directory to hold log files and - if generated - the reports for this scraper.

3.6.3 Custom settings

The scraper configuration is not limited to loading the settings indicated above. Hence, custom configuration settings (e.g. for site credentials) can be added to the ini file and then retrieved from the `config` attribute of a *Scraper* instance.

3.7 API documentation

The following documentation aims to present the internal API of the library. While it is possible to use all of these classes directly, following the usage patterns detailed in the rest of the documentation is advised.

3.7.1 Basic Scraper

class `scrapekit.core.`**`Scraper`**(*name*, *config=None*, *report=False*)

Scraper application object which handles resource management for a variety of related functions.

`Session`()

Create a pre-configured `requests` session instance that can be used to run HTTP requests. This instance will potentially be cached, or a stub, depending on the configuration of the scraper.

`get`(*url*, ***kwargs*)

HTTP GET via `requests`.

See: http://docs.python-requests.org/en/latest/api/#requests.get

`head`(*url*, ***kwargs*)

HTTP HEAD via `requests`.

See: http://docs.python-requests.org/en/latest/api/#requests.head

`post`(*url*, ***kwargs*)

HTTP POST via `requests`.

See: http://docs.python-requests.org/en/latest/api/#requests.post

put(*url*, ***kwargs*)
HTTP PUT via `requests`.

See: http://docs.python-requests.org/en/latest/api/#requests.put

report()
Generate a static HTML report for the last runs of the scraper from its log file.

task(*fn*)
Decorate a function as a task in the scraper framework. This will enable the function to be queued and executed in a separate thread, allowing for the execution of the scraper to be asynchronous.

3.7.2 Tasks and threaded execution

This module holds a simple system for the multi-threaded execution of scraper code. This can be used, for example, to split a scraper into several stages and to have multiple elements processed at the same time.

The goal of this module is to handle simple multi-threaded scrapers, while making it easy to upgrade to a queue-based setup using celery later.

class `scrapekit.tasks.`**Task**(*scraper*, *fn*, *task_id=None*)
A task is a decorator on a function which helps managing the execution of that function in a multi-threaded, queued context.

After a task has been applied to a function, it can either be used in the normal way (by calling it directly), through a simple queue (using the *queue* method), or in pipeline mode (using *chain*, *pipe* and *run*).

chain(*other_task*)
Add a chain listener to the execution of this task. Whenever an item has been processed by the task, the registered listener task will be queued to be executed with the output of this task.

Can also be written as:

```
pipeline = task1 > task2
```

pipe(*other_task*)
Add a pipe listener to the execution of this task. The output of this task is required to be an iterable. Each item in the iterable will be queued as the sole argument to an execution of the listener task.

Can also be written as:

```
pipeline = task1 | task2
```

queue(**args*, ***kwargs*)
Schedule a task for execution. The task call (and its arguments) will be placed on the queue and processed asynchronously.

run(**args*, ***kwargs*)
Queue a first item to execute, then wait for the queue to be empty before returning. This should be the default way of starting any scraper.

wait()
Wait for task execution in the current queue to be complete (ie. the queue to be empty). If only *queue* is called without *wait*, no processing will occur.

class `scrapekit.tasks.`**TaskManager**(*threads=10*)
The *TaskManager* is a singleton that manages the threads used to parallelize processing and the queue that manages the current set of prepared tasks.

put(*task*, *args*, *kwargs*)
Add a new item to the queue. An item is a task and the arguments needed to call it.

Do not call this directly, use Task.queue/Task.run instead.

wait()
Wait for each item in the queue to be processed. If this is not called, the main thread will end immediately and none of the tasks assigned to the threads would be executed.

3.7.3 HTTP caching and parsing

class scrapekit.http.**PolicyCacheController**(*cache=None*, *cache_etags=True*, *serializer=None*)
Switch the caching mode based on the caching policy provided by request, which in turn can be given at request time or through the scraper configuration.

class scrapekit.http.**ScraperResponse**
A modified scraper response that can parse the content into HTML, XML, JSON or a BeautifulSoup instance.

html()
Create an lxml-based HTML DOM from the response. The tree will not have a root, so all queries need to be relative (i.e. start with a dot).

json(***kwargs*)
Create JSON object out of the response.

xml()
Create an lxml-based XML DOM from the response. The tree will not have a root, so all queries need to be relative (i.e. start with a dot).

class scrapekit.http.**ScraperSession**
Sub-class requests session to be able to introduce additional state to sessions and responses.

scrapekit.http.**make_session**(*scraper*)
Instantiate a session with the desired configuration parameters, including the cache policy.

3.7.4 Exceptions and Errors

exception scrapekit.exc.**DependencyException**
Triggered when an operation would require the installation of further dependencies.

exception scrapekit.exc.**ParseException**
Triggered when parsing an HTTP response into the desired format (e.g. an HTML DOM, or JSON) is not possible.

exception scrapekit.exc.**ScraperException**
Generic scraper exception, the base for all other exceptions.

class scrapekit.exc.**WrappedMixIn**(*wrapped*)
Mix-in for wrapped exceptions.

CHAPTER 4

Contributors

`scrapekit` is written and maintained by Friedrich Lindenberg. It was developed as an outcome of scraping projects for the African Network of Centers for Investigative Reporting (ANCIR), supported by a Knight International Journalism Fellowship from the International Center for Journalists (ICFJ).

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

S