Memorious Documentation

Release 1

OCCRP

May 02, 2019

Contents

CHAPTER 1

Memorious

> The solitary and lucid spectator of a multiform, instantaneous and almost intolerably precise world.
>
> —Funes the Memorious, Jorge Luis Borges

`memorious` is a distributed web scraping toolkit. It is a light-weight tool that schedules, monitors and supports scrapers that collect structured or un-structured data. This includes the following use cases:

- Maintain an overview of a fleet of crawlers
- Schedule crawler execution in regular intervals
- Store execution information and error messages
- Distribute scraping tasks across multiple machines
- Make crawlers modular and simple tasks re-usable
- Get out of your way as much as possible

1.1 Design

When writing a scraper, you often need to paginate through through an index page, then download an HTML page for each result and finally parse that page and insert or update a record in a database.

`memorious` handles this by managing a set of `crawlers`, each of which can be composed of multiple `stages`. Each `stage` is implemented using a Python function, which can be re-used across different `crawlers`.

The basic steps of writing a Memorious crawler:

1. Make YAML crawler configuration file
2. Add different stages
3. Write code for stage operations (optional)
4. Test, rinse, repeat

1.2 Documentation

The documentation for Memorious is available at memorious.readthedocs.io. Feel free to edit the source files in the `docs` folder and send pull requests for improvements.

To build the documentation, inside the `docs` folder run `make html`

You'll find the resulting HTML files in /docs/_build/html.

1.2.1 Table of contents

Installation (running your own crawlers)

We recommend using Docker Compose to run your crawlers in production, and we have an example project to help you get started.

- Make a copy of the `memorious/example` directory.
- Add your own crawler YAML configurations into the `config` directory.
- Add your Python extensions into the `src` directory (if applicable).
- Update `setup.py` with the name of your project and any additional dependencies.
- If you need to (eg. if your database connection or directory structure is different), update any environment variables in the `Dockerfile` or `docker-compose.yml`, although the defaults should work fine.
- Run `docker-compose up -d`. This might take a while when it's building for the first time.

You can access the Memorious CLI through the `worker` container:

```
docker-compose run --rm worker /bin/bash
```

To see the crawlers available to you:

```
memorious list
```

And to run a crawler:

```
memorious run my_crawler
```

See Usage (or run `memorious --help`) for the complete list of Memorious commands.

Note: you can use any directory structure you like, `src` and `config` are not required, and nor is separation of YAML and Python files. So long as the `MEMORIOUS_CONFIG_PATH` environment variable points to a directory containing, within any level of directory nesting, your YAML files, Memorious will find them.

Environment variables

Your Memorious instance is configured by a set of environment variables that control database connectivity and general principles of how the sytem operates. You can set all of these in the `Dockerfile`.

- `MEMORIOUS_CONFIG_PATH`: a path to crawler pipeline YAML configurations.
- `MEMORIOUS_DEBUG`: whether to go into a simple mode with task threading disabled. Defaults to `False`.
- `MEMORIOUS_INCREMENTAL`: executing part of a crawler only once per an interval. Defaults to `True`.
- `MEMORIOUS_HTTP_CACHE`: HTTP request configuration.
- `MEMORIOUS_DATASTORE_URI`: connection path for an operational database (which crawlers can send data to using the `db` method). Defaults to a local `datastore.sqllite3`.
- `MEMORIOUS_THREADS`: how many threads to use for execution.
- `MEMORIOUS_MAX_SCHEDULED`: maximum number of scheduled tasks at the same time. Defaults to the same as the number of threads.
- `REDIS_URL`: address of Redis instance to use for crawler logs (uses a temporary FakeRedis if missing).
- `ARCHIVE_TYPE`: either `file` or `s3`.
- `ARCHIVE_PATH`
- `ARCHIVE_BUCKET`
- `AWS_KEY_ID`: AWS Access Key ID.
- `AWS_SECRET`: AWS Secret Access Key.
- `AWS_REGION`: a regional AWS endpoint.
- `ALEPH_HOST`, default is `https://data.occrp.org/`, but any instance of Aleph 2.0 or greater should work.
- `ALEPH_API_KEY`, a valid API key for use by the upload operation.

Shut it down

To gracefully exit, run `docker-compose down`.

Files which were downloaded by crawlers you ran, Memorious progress data from the Redis database, and the Redis task queue, are all persisted in the `build` directory, and will be reused next time you start it up. (If you need a completely fresh start, you can delete this directory).

Building a crawler

To understand what goes into your `config` and `src` directories, check out the examples and reference documentation.

Development mode

When you're working on your crawlers, it's not convenient to rebuild your Docker containers all the time. To run without Docker:

- Copy the environment variables from the `env.sh.tmpl` to `env.sh`.
- Run `source env.sh`.

Make sure `MEMORIOUS_CONFIG_PATH` points to your crawler YAML files, wherever they may be.

Then either:

- Run `pip install memorious`. If your crawlers use Python extensions, you'll need to run `pip install` in your crawlers directory as well;
- **or** clone the Memorious repository and run `make install` (this will also install your crawlers for you).

Usage

`memorious` is controlled via a command-line tool, which can be used to monitor or invoke a crawler interactively. Most of the actual work, however, is handled by a daemon service running in the background. Communication between different components is handled via a central message queue.

See the status of all crawlers managed by memorious:

```
memorious list
```

Force an immediate run of a specific crawler:

```
memorious run my_crawler
```

Check which crawlers are due for scheduled execution and execute the ones that need to be updated:

```
memorious scheduled
```

Clear all the run status and cached information associated with a crawler:

```
memorious flush my_crawler
```

Building a crawler

Memorious contains all of the functionality for basic Web crawlers, which can be configured and customised entirely through YAML files. For more complex crawlers, Memorious can be extended with custom Python functions, which you can point a crawler at through its YAML config.

We'll start by describing the included functionality.

The first few lines of your config are to set up your crawler:

- `name`: A unique slug, eg. "my_crawler", which you can pass to `memorious run` to start your crawler.
- `description`: An optional description, will be shown when you run `list`.
- `schedule`: one of `hourly`, `daily`, `weekly` or `monthly`.

The Pipeline

Memorious crawlers are made up of stages, each of which take care of a particular part of a crawler's pipeline. Each stage takes an input from the previous stage, and yields an output for the next stage. For example, a crawling stage might find every URL on a webpage and pass it to a parsing stage which fetches and downloads the contents of each URL.

The first stage can be configured to automatically generate the starting input, or you can pass an input directly. *See initializers*.

The final stage is likely to be for storage, either via an API like aleph or writing to disc. *See storing*.

Every stage has access to the crawler's persistent *context* object and the data that was passed from the previous stage. The `data` dict depends on the output of the previous stage. See the specific stages for what this looks like in each case.

You probably only need to think about the `context` if you're writing *extensions*.

Stages

Each stage of a crawler is delimited by a child of the `pipeline` key in its YAML config. You can name the stages anything you like, and use these keys to refer to one stage from another.

A stage must contain:

- `method`: what do you want Memorious to do when it gets to this stage.
- `handle`: which stage is triggered next and under what conditions.
  - The default condition is `pass`. (ie. `pass: crawl` means in the case of a 'pass' condition invoke the stage called 'crawl').
  - Some in-built methods may return different conditions depending on the input - see method-specific sections.
  - You will care more about this if you're *extending* Memorious.

```
name: my_crawler
pipeline:
  init:
    method: xxx
    ...
    handle:
      pass: crawl
  crawl:
    method: yyy
    ...
    handle:
      pass: save
  save:
    method: zzz
    ...
```

A stage may also contain a `params` key which lets you pass values *in* from the config. The data that comes *out* of each stage are available to the next stage via the `data` dict. Read on for the standard methods Memorious makes available to you, the parameters they take, and their output variables.

Skip to *extending* to see how to use custom methods if you need something that Memorious doesn't do.

Initializers

The initializer methods are:

- `sequence`: generate a sequence of numbers.
- `dates`: generate a sequence of dates.
- `enumerate`: loop through a list of items.
- `seed`: loop through a list of URLs.

Sequence

Parameters (all optional):

- `start`: the start of the sequence. Defaults to 1.
- `stop`: the end of the sequence.
- `step`: how much to increment by. Defaults to 1; can be negative.
- `delay`: numbers can be generated one by one with a delay to avoid large sequences clogging up the queue.
- `prefix`: a string which ensures each number will be emitted only once across multiple runs of the crawler.

If this stage is preceded by a stage which outputs a `number` (for example, another `sequence` stage), it will use this value as the start of the sequence instead of `start`.

Output data:

- `number`: the number in the sequence.

Dates

This generates a sequence of dates, counting backwards from `end`, either to `begin` or according to the number of `steps`, and the `days`/`weeks` value is the size of each step.

Parameters (all optional):

- `format`: date format to expect and/or output. Defaults to "%Y-%m-%d".
- `end`: latest date to generate (should match `format`). Defaults to 'now'.
- `begin`: earliest date to generate (should match `format`). Overrides `steps`.
- `days`: the time difference to increment by. Defaults to 0.
- `weeks`: the time difference to increment by. Defaults to 0.
- `steps`: The number of times to increment. Defaults to 100. Ignored if `begin` is set.

Output data:

- `date`: a date formatted by the input `format`.
- `date_iso`: a date in ISO format.

Enumerate

Emits each item in a list so they can be passed one at a time to the next stage.

Parameters:

- `items`: a list of items to loop through.

Output data:

- `item`: one of the items from the input list.

Seed

Starts a crawler with URLs, given as a list or single value .If this is called as a second stage in a crawler, the URL will be formatted against the supplied `data` values, ie: `https://crawl.site/entries/%(number)s.html`

Parameters:

- `url` or `urls`: one or more URLs to loop through.

Output data:

- `url`: each URL, with data from the previous stage substituted if applicable.

Fetching and parsing

Fetch

The `fetch` method does an HTTP `GET` on the value of `url` in data passed from the previous stage.

Parameters (optional):

- `rules`: only the URLs which match are retrieved. See *Rules*.

Output data:

- The serialized result of the HTTP `GET` response.

Ftp Fetch

The `ftp_fetch` method does an FTP `NLIST` on the value of `url` in data passed from the previous stage.

Parameters:

- `username`: for FTP username authentication, defaults to `Anonymous`.
- `password`: for FTP password authentication, defaults to `anonymous@ftp`.

Output data:

- The serialized result of the FTP `NLIST` response.

Clean

The `clean_html` takes an HTTP response from something like `fetch` and strips down the HTML according to the parameters you pass. You can also use it to set metadata from an XPath (so far, `title`).

Parameters:

- `remove_paths`: a list of XPaths to strip from the HTML.
- `title_path`: a single XPath to indicate where to find the title of the document.

Output data:

- What went in, plus added metadata, with the HTML content hash replaced with the cleaned version.

DAV index

The `dav_index` method lists the files in a WebDAV directory and does nad HTTP `get` on them; the directory is passed via the `url` of the previous stage data.

Output data:

- The serialized result of the HTTP `GET` response.

Session

The `session` method sets some HTTP parameters for all subsequent requests.

Parameters:

- `user`: for HTTP Basic authentication.
- `password`: for HTTP Basic authentication.
- `user_agent`: the User-Agent HTTP header.
- `proxy`: proxy server address for HTTP tunneling.

Output data:

- Emits the same `data` dict that was passed in, unmodified.

Parse

The `parse` method recursively finds URLs in webpages. It looks in the `href` attributes of `a` and `link` elements, and the `src` attributes of `img` and `iframe` elements.

As `data` input from the previous stage, it expects a `ContextHttpResponse` object.

Parameters (optional):

- `store`: only the results which match are stored. See *Rules*. If no rules are passed, everything is stored.
- `include_paths`: A list of XPaths. If included, parse will only check these routes for URLs.
- `meta`: A list of key-value pairs of additional metadata to parse from the DOM, where the key is the key for `data` and the value is an XPath of where to find it.
- `meta_date`: The same as `meta` but the value is parsed as a date.

Output:

- If the input data contains HTML, it passes each URL it finds therein to the current stage's `fetch` handler.
- The input data (unmodified) is also passed to the current stage's `store` handler, filtered by any *rules* passed via the `store` param if applicable.

An example `parse` configuration, which crawls links and stores only documents:

```
parse:
  method: parse
  params:
    store:
      mime_group: documents
    include_paths:
     - './/aside'
```

(continues on next page)

(continued from previous page)

```
      - './/article
    meta:
      creator: './/article/p[@class="author"]'
      title: './/h1'
    meta_date:
      published_at: './/article/time'
      updated_at: './/article//span[@id="updated"]'
  handle:
    fetch: fetch
    store: store
```

DocumentCloud

The `documentcloud_query` method harvests documents from a documentcloud.org instance.

Parameters:

- `host`: the URL of the DocumentCloud host. Defaults to 'https://documentcloud.org/'.
- `instance`: the name of the DocumentCloud instance. Defaults to 'documentcloud'.
- `query`: the query to send to the DocumentCloud search API.

Output data:

- `url`: the URL of the document.
- `source_url`: the canonical URL from documentcloud metadata.
- `foreign_id`: a unique ID from the instance and the document ID.
- `file_name`: where the document is stored locally (?).
- `mime_type`: hardcoded to `application/pdf`.
- `title`: from documentcloud metadata.
- `author`: from documentcloud metadata.
- `languages`: from documentcloud metadata.
- `countries`: from documentcloud metadata.

Storing

The final stage of a crawler is to store the data you want.

Directory

The `directory` method stores the collected files in the given directory.

The input data from the previous stage is expected to be a `ContextHttpResponse` object.

Parameters:

- `path`: the directory to store files in, relative to the `MEMORIOUS_BASE_PATH` environment variable (another directory will be created in here, named after the specific crawler, so it's safe to pass the same `path` to multiple crawlers).

Output:

- The file is stored in `path`.
- The `data` dict is dumped as a JSON file in `path` too.

Database

The `db` method stores `data` as a row in a specified database table with appropriate timestamps. `__last_seen` and `__first_seen` timestamps are added based on when a row was updated or inserted respectively.

Parameters:

- `table`: the name of the database table in which data will be stored
- `unique`: A list of keys in data. If `unique` is defined, we try to update existing columns based on the values of keys in `unique`. If no matching row is found, a new row is inserted.

Rules

You can configure rules per stage to tell certain methods which inputs to process or skip. You can nest them, and apply `not`, `and` and `or` for the combinations you desire.

- `mime_type`: Match the MIME type string.
- `mime_group`: See mime.py for handy MIME type groupings (`web`, `images`, `media`, `documents`, `archives` and `assets`).
- `domain`: URL contains this domain.
- `pattern`: URL matches this regex.

Extending

If none of the inbuilt methods do it for you, you can write your own. You'll need to package your methods up into a python module, and install it (see installation instructions in readme).

You can then call these methods from a YAML config instead of the Memorious ones. eg:

```
my_stage:
  method: custom.module:my_method
  params:
    my_param: my_value
  handle:
    pass: store
```

Your method needs to accept two arguments, `context` and `data`.

The `data` dict is what was output from the previous stage, and what it contains depends on the the method from that stage. The *context object* gives you access to various useful variables and helper functions...

Context

Access the YAML config:

- You can access `params` with `context.params.get('my_param')`.

- You can also access other properties of the crawler, eg. `context.get('name')` and `context.get('description')`.

The HTTP session:

- `context.http` is a wrapper for requests. Use `context.http.get` (or `.post`) just like you would use requests, and benefit from Memorious database caching; session persistence; lazy evaluation; and serialization of responses between crawler operations.
- Properties of the `ContextHTTPResponse` object:
  - `url`
  - `status_code`
  - `headers`
  - `encoding`
  - `file_path`
  - `content_hash`
  - `content_type`
  - `ok` (bool)
  - The content as `raw`, `text`, `html`, `xml`, or `json`
  - `retrieved_at`: the date the GET request was made.
  - `modified_at`: from the `Last-Modified` header, provided it wasn't in the last 16 seconds.

Data validation: As part of the context logic the following data validation helpers are available:

- `is_not_empty`: whether value is not empty.
- `is_numeric`: whether value is numeric.
- `is_integer`: whether value is an integer.
- `match_date`: whether value is a date.
- `match_regexp`: whether value matches a regexp.
- `has_length`: whether value has a given length.
- `must_contain`: whether value contains a string.

The datastore:

- Create and access tables in the Memorious database to store intermediary useful crawler data: `table = context.datastore['my_table']`.
- See dataset for the rest of how this works..

Output:

- Call `context.recurse(data=data)` to have a stage invoke itself with a modified set of arguments (this is useful for example for paging through search results and handing off each list of links to a `fetch` stage).
- To pass data from `my_method` to the next stage, use: `context.emit(data={'my_key': 'my_value'})`.
- `context.store_file(path, content_hash)`: Put a file into permanent storage so it can be visible to other stages.

Logs:

- `context.log.info()`, `.warning()`, `.error()` to explictly log things.

Helpers

Memorious contains useful helper functions you might like to use:

```
from memorious.helpers import ...
```

- `ViewForm`: Helper for VIEWSTATE in ASP-driven web sites.
- `convert_snakecase`: Convert a given string to 'snake_case'.
- `soviet_checksum`: Ensure a company code from [TODO: countries] is valid.
- `search_results_total`: Extracts the total search results count from a search index page. Pass it the page as an html object, an xpath route to the element containing the results text, a string to check that you're looking in the right element, and a string delimiter which occurs immediately before the actual number.
- `search_results_last_url`: Get the URL for the 'last' button in search results listing.
- `parse_date`: Parse a string and return a string representing the date and time. Optional: use format codes.
- `iso_date`: Return a date string in ISO 8601 format.
- `make_id`: Make a string key out of many criteria.

OCR

```
from memorious.helpers.ocr import ...
```

Memorious contains some helpers that use Tesseract to OCR images. This depends on tesserocr, which depends on Tesseract version 0.3.4+. If you wish to use these helpers you need to install an up to date version of Tesseract (and *its* dependencies), *then* `pip install tesserocr`.

- `read_word`: OCR a single word from an image.
- `read_char`: OCR a single character from an image.

See the Tesseract wiki for more installation details.

`tesserocr` is not listed as a Memorious dependency, because Tesseract is not a sane dependency unless you're actually going to use it.

Postprocessing

It's possible to run predefined postprocessing tasks after a Memorious crawler has finished running. The postprocessing task is defined under `aggregator` section in a crawler's YAML config.

`aggregator` should contain:

- `method`: which function to execute for postprocessing
- `params` (optional): params to pass to the postprocessing method

eg:

Here's an example from `example/config/extended_web_scraper.yml`

```
name: ...
description: ...
schedule: ...
pipeline:
  ...
aggregator:
  method: example.quotes:export
  params:
    filename: all_quotes.json
```

Development

Get the code here: https://github.com/alephdata/memorious

Licensing

see LICENSE