LyricsScraping

Release 0.1

Raul C.

Aug 03, 2020

CONTENTS

CHAPTER ONE

README [WORK-IN-PROGRESS]

LyricsScraping crawls and scraps lyrics from AZLyrics.

1.1 Dependencies

- **Platforms:** macOS, Linux, Windows
- **Python**: 3.5, 3.6, 3.7
- `BeautifulSoup` : used for crawling and parsing the lyrics webpages
- `requests` : used for requesting the HTML content of lyrics webpages
- `yaml` : used for reading configuration files (e.g. logging)
- `py-common-utils` : is a Python collection of utilities with useful functions and modules ready to be used in different projects. For instance, you will find code related to databases and logging.

1.2 Installation instructions

1. Download the LyicsScraping and py-common-utils libraries
2. . . .

1.3 Usage

These are the two ways to use the `lyrics-scraping` package:

1. Run the `scraper` script
2. Use the library as API in your own code

1.3.1 Run the main script

Run the script with:

```
$ python run_scraper.py -c
```

Note:

- The option `-c` is for adding color to log messages.

1.3.2 Use the library in your own code

CHAPTER TWO

API REFERENCE

2.1 `lyrics_scraping.scrapers`

`lyrics_scraping.scrapers` is a package that contains modules that define scrapers for specific lyrics websites, e.g. `azlyrics_scraper`.

2.1.1 `scrapers.lyrics_scraper`

Description

Module that defines the base class for scraping lyrics websites and saving the scraped content.

More specifically, the derived classes (e.g. `AZLyricsScraper`) are the ones that do the actual scraping of the lyrics webpages.

By default, the scraped data is saved in a dictionary (see the variable *`scraped_data`*).

The scraped data can also be saved in a database if a path to the SQLite database is given via the argument *db_filepath*.

See the structure of the music database as defined in the music.sql schema.

Class and methods

class scrapers.lyrics_scraper.**Album**(*album_title*, *artist_name*, *album_url*, *year*)

Bases: `object`

TODO

static check_album_year(*year_result*)

TODO

Parameters **year_result** (*list*) – TODO

class scrapers.lyrics_scraper.**Artist**(*song_title*, *artist_name*, *artist_url*)

Bases: `object`

TODO

class scrapers.lyrics_scraper.**ComputeCache**(*schema_filepath*, *ram_size*)

Bases: `object`

TODO

class scrapers.lyrics_scraper.**Lyrics**(*song_title*, *artist_name*, *album_title*, *lyrics_url*, *lyrics_text*, *year*)

Bases: `object`

TODO: remove, to be replaced by Song

class scrapers.lyrics_scraper.**LyricsScraper**(*db_filepath=''*, *overwrite_db=False*, *use_webcache=True*, *webcache_dirpath='~/.cache/lyric_scraping/'*, *expire_after=25920000*, *use_compute_cache=True*, *ram_size=100*, *http_get_timeout=5*, *delay_between_requests=8*, *headers=pyutils.webcache.WebCache.HEADERS*, *seed=123456*, *interactive=False*, *delay_interactive=30*, *best_match=False*, *simulate=False*, *ignore_errors=False*)

Bases: `object`

Base class for scraping and saving webpages locally.

This class is responsible for doing lots of configuration before the web scraping starts, such as setting up logging and the database.

The actual scraping of the lyrics websites is done by the derived classes (e.g. `AZLyricsScraper`) since each lyrics websites have their own way of being crawled (they are all designed differently). However, the base class is responsible for saving the scraped data in a dictionary (*scraped_data*) and in a database (if it was initially make configured).

Parameters

- **lyrics_urls** (*list [str]*) – List of URLs to lyrics webpages which will be scraped.
- **db_filepath** (*str, optional*) – File path to the SQLite music database (the default value is `None` which implies that no database will be used. The scraped data will be saved only in the *scraped_data* dictionary).
- **autocommit** (*bool, optional*) – Whether the changes to the database are committed right away (the default is False which implies that the changes won't take effect immediately).

- **overwrite_db** (*bool, optional*) – Whether the database will be overwritten. The user is given some time to stop the script before the database is overwritten (the default value is False).
- **update_tables** (*bool, optional*) – Whether the tables in the database can be updated (the default value is False).
- **cache_dirpath** (*str, optional*) – Path to the cache directory where webpages are saved (the default value is `None` which implies that the cache will not be used).
- **overwrite_webpages** (*bool, optional*) – Whether the webpages saved in cache can be overwritten (the default value is False).
- **http_get_timeout** (*int, optional*) – Timeout when a GET request doesn't receive any response from the server. After the timeout expires, the GET request is dropped (the default value is 5 seconds).
- **delay_between_requests** (*int, optional*) – A delay will be added between HTTP requests in order to reduce the workload on the server (the default value is 8 seconds which implies that there will be a delay of 8 seconds between successive HTTP requests).
- **headers** (*dict, optional*) – The information added to the HTTP GET request that a user's browser sends to a Web server containing the details of what the browser wants and will accept back from the server. (the default value is defined in `saveutils.SaveWebpages.headers`).
- **use_logging** (*bool, optional*) – Whether to log messages on console and file. The logging is setup according to the YAML logging file (the default value is False which implies that no logging will be used and thus no messages will be printed on the console).
- ****kwargs** (*dict*) – TODO

Variables

- **skipped_urls** (*dict [str, str]*) – Stores the URLs that were skipped because of an error such as `OSError` or `HTTP404Error`, along with the error message. The keys are the URLs and the values are the associated error messages.
- **good_urls** (*set*) – Stores the unique URLs that were successfully processed and saved.
- **checked_urls** (*set*) – Stores the unique URLs that were processed (whether successfully or unsuccessfully) during the current session. Thus, *checked_urls* should equal to *skipped_urls* + *good_urls*.
- **db_conn** (*sqlite3.Connection*) – SQLite database connection.
- **saver** (`saveutils.SaveWebpages`) – For retrieving webpages and saving them in cache. See `saveutils`.
- ***valid_domains*** (*list*) – Only URLs from these domains will be processed.
- **logging_filepath** (*str*) – Path to the YAML logging file which is used to setup logging for all custom modules.
- **schema_filepath** (*str*) – Path to music.sql schema for building the music database which will store the scraped data.
- ***scraped_data*** (*dict*) – The scraped data is saved as a dictionary. Its structure is based on the database's music.sql schema.

Notes

If the corresponding flags are activated, logging and database are setup in `__init__()`.

By default, the scraped data is saved in a dictionary whose structure is described below (see *scraped_data*). The scraped data will also be saved if a database is given via *db_filepath*.

See the structure of the music database as defined in the music.sql schema.

The scraped webpages can also be cached in order to reduce the number of HTTP requests to the server (See *db_filepath*).

get_lyrics_from_album(*album_title*, *artist_name=None*, *max_songs=None*)
TODO

Parameters

- **album_title** –
- **artist_name** –
- **max_songs** –

get_lyrics_from_artist(*artist_name*, *max_songs=None*, *year_after=None*, *year_before=None*)
TODO

Parameters

- **artist_name** –
- **max_songs** –
- **year_after** –
- **year_before** –

get_scraped_data()
Return the scraped data as a dictionary.

This method returns all the data that was scraped from the lyrics webpages. If a database was used, the scraped data is also saved in the SQLite database file found at *db_filepath*

See *scraped_data* for a detailed structure of the returned dictionary.

Returns **scraped_data** – The scraped data whose content is described in *scraped_data*.

Return type dict

get_song_lyrics(*song_title*, *artist_name=None*)
TODO

Parameters

- **song_title** –
- **artist_name** –

scraped_data = {'albums': {'data': [], 'headers': ('album_title', 'artist_name', 'y
The scraped data is saved as a dictionary.

Its keys and values are defined as follow:

```
scraped_data = {
    'albums': {
        'headers': ('album_title', 'artist_name', 'year',),
```

(continues on next page)

(continued from previous page)

```
        'data': []
    },
    'artists': {
        'headers': ('artist_name',),
        'data': []
    },
    'songs': {
        'headers': ('song_title', 'artist_name', 'album_title',
                    'lyrics_url', 'lyrics', 'year',),
        'data': []
    }
}
```

Note: The 'data' key points to a list of tuple that eventually will store the scraped data from different URLs, i.e. each scraped data from a given URL is added as a tuple to the list.

`search_album`(*album_title*, *artist_name=None*)
TODO

Parameters

- **`album_title`** –
- **`artist_name`** –

`search_artist`(*artist_name=None*)
TODO

Parameters **`artist_name`** –

`search_song_lyrics`(*song_title*, *artist_name=None*)
TODO

Parameters

- **`song_title`** –
- **`artist_name`** –

`start_scraping`()
Start the web scraping of lyrics websites.

This method iterates through each lyrics URL from the main config file and delegates the important tasks (URL processing and scraping) to separate methods (`_process_url()` and `_scrape_webpage()`).

Notes

This method catches all exceptions that prevent a given URL of being processed further, e.g. the webpage is not found (404 Error) or the URL is not from a valid domain.

Any exception that is not caught here is redirected to the main script calling this method. See for example the main script `scripts.scraper`.

`valid_domains = ['www.azlyrics.com']`

class `scrapers.lyrics_scraper.`**`Song`**(*song_title*, *artist_name*, *album_title*, *lyrics_url*, *lyrics_text*, *year*)
Bases: `object`

TODO

2.1.2 `scrapers.azlyrics_scraper`

2.1.3 `scrapers.exceptions`

2.2 `lyrics_scraping.scripts`

`lyrics_scraping.scrapers` is a package that contains modules that define scripts, e.g. `scripts.scraper`.

2.2.1 `scripts.scraper`

2.3 `lyrics_scraping.utils`

2.3.1 Description

Collection of utilities specifically for the lyrics scraping project.

2.3.2 Functions

`utils.`**`dump_cfg`**`(`*filepath*, *cfg_dict*`)`

TODO

Parameters

- **`filepath`** –
- **`cfg_dict`** –

`utils.`**`get_backup_cfg_filepath`**`(`*cfg_type*`)`

TODO

Parameters **`cfg_type`** –

`utils.`**`get_data_dirpath`**`()`

TODO

`utils.`**`get_data_filepath`**`(`*file_type*`)`

Return the path to a data file used by `lyrics_scraping`.

The data file can either be the:

- **default_log**: refers to the default logging configuration file used to setup the logging for all custom modules.
- **default_main**: refers to the default main configuration file used to setup a lyrics scraper.
- **log**: refers to the user-defined logging configuration file which is used to setup the logging for all custom modules.
- **main**: refers to the user-defined main configuration file used to setup a lyrics scraper.
- **schema**: refers to the SQL schema file music.sql used for creating the SQLite database which stores the scraped data.

Parameters **`file_type`** (*`str`*, *`{'default_log', 'default_main', 'log', 'main', 'schema'}`*) – The type of data file for which we want the path.

Returns **filepath** – The path to the data file.

Return type str

Raises **`AssertionError`** – Raised if the wrong type of data file is given to the function. Only {'default_log', 'default_main', 'log', 'main', 'schema'} are accepted for *file_type*.

`utils.`**`load_cfg`**(*cfg_type*)

TODO

Parameters **`cfg_type`** –

`utils.`**`plural`**(*obj*, *plural_end='s'*, *singular_end=''*)

Add plural ending if a number is greater than 1 or there are many values in a list.

If the number is greater than one or more than one item is found in the list, the function returns by default 's'. If not, then the empty string is returned.

Parameters

- **`obj`** (*`int`*, *`float or list`*) – The number or list that will be checked if a plural or singular ending will be returned.
- **`plural_end`** (*`str`*, *`optional`*) – The plural ending (the default value is "s" which implies that "s'" will be returned in the case that the number is greater than 1 or the list contains more than one item).
- **`singular_end`** (*`str`*, *`optional`*) – The singular ending (the default value is "" which implies that an empty string will be returned in the case that the number is 1 or less, or the list contains 1 item).

Returns **str** – "s" if number is greater than 1 or more than one item is found in the list, "" (empty string) otherwise.

Return type "s" or ""

Raises **`TypeError`** – TODO

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX