darc

Release 0.1.6

Jarry Shaw

Mar 15, 2020

CONTENTS

CHAPTER

ONE

DARKWEB CRAWLER PROJECT

`darc` is designed as a swiss army knife for darkweb crawling. It integrates `requests` to collect HTTP request and response information, such as cookies, header fields, etc. It also bundles `selenium` to provide a fully rendered web page and screenshot of such view.

1.1 Main Processing

The *darc.process* module contains the main processing logic of the `darc` module.

`darc.process.`**`_dump_last_word`**(*errors=True*)

Dump data in queue.

Parameters **`errors`** (*bool*) – If the function is called upon error raised.

The function will remove the backup of the `requests` database `_queue_requests.txt.tmp` (if exists) and the backup of the `selenium` database `_queue_selenium.txt.tmp` (if exists).

If `errors` is `True`, the function will copy the backup of the `requests` database `_queue_requests.txt.tmp` (if exists) and the backup of the `selenium` database `_queue_selenium.txt.tmp` (if exists) to the corresponding database.

The function will also remove the PID file `darc.pid`

See also:

- *darc.const.getpid()*
- *darc.db.save_requests()*
- *darc.db.save_selenium()*

`darc.process.`**`_get_requests_links`**()

Fetch links from queue.

Returns List of links from the `requests` database.

Return type List[str]

Deprecated since version 0.1.0: Use *darc.db.load_requests()* instead.

`darc.process.`**`_get_selenium_links`**()

Fetch links from queue.

Returns List of links from the `selenium` database.

Return type List[str]

Deprecated since version 0.1.0: Use *darc.db.load_selenium()* instead.

darc.process.**_load_last_word**()
Load data to queue.

The function will copy the backup of the `requests` database `_queue_requests.txt.tmp` (if exists) and the backup of the `selenium` database `_queue_selenium.txt.tmp` (if exists) to the corresponding database.

The function will also save the process ID to the `darc.pid` PID file.

See also:

- *darc.const.getpid()*
- *darc.db.load_requests()*
- *darc.db.load_selenium()*

darc.process.**_signal_handler**(*signum=None*, *frame=None*)
Signal handler.

The function will call *_dump_last_word()* to keep a decent death.

If the current process is not the main process, the function shall do nothing.

Parameters

- **signum** (*Union[int, signal.Signals, None]*) – The signal to handle.
- **frame** (*types.FrameType*) – The traceback frame from the signal.

See also:

- *darc.const.getpid()*

darc.process.**process**()
Main process.

The function will register *_signal_handler()* for `SIGTERM`, and start the main process of the `darc` darkweb crawlers.

The general process can be described as following:

0. *process()*: obtain URLs from the `requests` link database (c.f. *load_requests()*), and feed such URLs to *crawler()* with *multiprocessing* support.
1. *crawler()*: parse the URL using *parse_link()*, and check if need to crawl the URL (c.f. *PROXY_WHITE_LIST*, *PROXY_BLACK_LIST* , *LINK_WHITE_LIST* and *LINK_BLACK_LIST*); if true, then crawl the URL with `requests`.

   If the URL is from a brand new host, `darc` will first try to fetch and save `robots.txt` and sitemaps of the host (c.f. *save_robots()* and *save_sitemap()*), and extract then save the links from sitemaps (c.f. *read_sitemap()*) into link database for future crawling (c.f. *save_requests()*). Also, if the submission API is provided, *submit_new_host()* will be called and submit the documents just fetched.

   If `robots.txt` presented, and *FORCE* is `False`, `darc` will check if allowed to crawl the URL.

   Note: The root path (e.g. `/` in https://www.example.com/) will always be crawled ignoring `robots.txt`.

At this point, darc will call the customised hook function from darc.sites to crawl and get the final response object. darc will save the session cookies and header information, using *save_headers()*.

Note: If requests.exceptions.InvalidSchema is raised, the link will be saved by save_invalid(). Further processing is dropped.

If the content type of response document is not ignored (c.f. *MIME_WHITE_LIST* and *MIME_BLACK_LIST*), darc will save the document using *save_html()* or *save_file()* accordingly. And if the submission API is provided, *submit_requests()* will be called and submit the document just fetched.

If the response document is HTML (text/html and application/xhtml+xml), *extract_links()* will be called then to extract all possible links from the HTML document and save such links into the database (c.f. *save_requests()*).

And if the response status code is between 400 and 600, the URL will be saved back to the link database (c.f. *save_requests()*). If **NOT**, the URL will be saved into selenium link database to proceed next steps (c.f. *save_selenium()*).

2. *process()*: after the obtained URLs have all been crawled, darc will obtain URLs from the selenium link database (c.f. *load_selenium()*), and feed such URLs to *loader()*.

Note: If *FLAG_MP* is True, the function will be called with *multiprocessing* support; if *FLAG_TH* if True, the function will be called with *multithreading* support; if none, the function will be called in single-threading.

---

3. *loader()*: parse the URL using *parse_link()* and start loading the URL using selenium with Google Chrome.

   At this point, darc will call the customised hook function from darc.sites to load and return the original selenium.webdriver.Chrome object.

   If successful, the rendered source HTML document will be saved using *save_html()*, and a full-page screenshot will be taken and saved.

   If the submission API is provided, *submit_selenium()* will be called and submit the document just loaded.

   Later, *extract_links()* will be called then to extract all possible links from the HTML document and save such links into the requests database (c.f. *save_requests()*).

If in reboot mode, i.e. *REBOOT* is True, the function will exit after first round. If not, it will renew the Tor connections (if bootstrapped), c.f. *renew_tor_session()*, and start another round.

1.2 Web Crawlers

The *darc.crawl* module provides two types of crawlers.

- *crawler()* – crawler powered by requests
- *loader()* – crawler powered by selenium

darc.crawl.**crawler**(*url*)
Single requests crawler for a entry link.

Parameters **url** (*str*) – URL to be crawled by requests.

The function will first parse the URL using *parse_link()*, and check if need to crawl the URL (c.f. *PROXY_WHITE_LIST*, *PROXY_BLACK_LIST* , *LINK_WHITE_LIST* and *LINK_BLACK_LIST*); if true, then crawl the URL with requests.

If the URL is from a brand new host, darc will first try to fetch and save robots.txt and sitemaps of the host (c.f. *save_robots()* and *save_sitemap()*), and extract then save the links from sitemaps (c.f. *read_sitemap()*) into link database for future crawling (c.f. *save_requests()*). Also, if the submission API is provided, *submit_new_host()* will be called and submit the documents just fetched.

See also:

- *darc.proxy.null.fetch_sitemap()*

If robots.txt presented, and *FORCE* is False, darc will check if allowed to crawl the URL.

Note: The root path (e.g. / in https://www.example.com/) will always be crawled ignoring robots.txt.

At this point, darc will call the customised hook function from darc.sites to crawl and get the final response object. darc will save the session cookies and header information, using *save_headers()*.

Note: If requests.exceptions.InvalidSchema is raised, the link will be saved by *save_invalid()*. Further processing is dropped.

If the content type of response document is not ignored (c.f. *MIME_WHITE_LIST* and *MIME_BLACK_LIST*), darc will save the document using *save_html()* or *save_file()* accordingly. And if the submission API is provided, *submit_requests()* will be called and submit the document just fetched.

If the response document is HTML (text/html and application/xhtml+xml), *extract_links()* will be called then to extract all possible links from the HTML document and save such links into the database (c.f. *save_requests()*).

And if the response status code is between 400 and 600, the URL will be saved back to the link database (c.f. *save_requests()*). If **NOT**, the URL will be saved into selenium link database to proceed next steps (c.f. *save_selenium()*).

darc.crawl.**loader**(*url*)

Single selenium loader for a entry link.

Parameters **url** (*str*) – URL to be crawled by requests.

The function will first parse the URL using *parse_link()* and start loading the URL using selenium with Google Chrome.

At this point, darc will call the customised hook function from darc.sites to load and return the original selenium.webdriver.Chrome object.

If successful, the rendered source HTML document will be saved using *save_html()*, and a full-page screenshot will be taken and saved.

Note: When taking full-page screenshot, *loader()* will use document.body.scrollHeight to get the total height of web page. If the page height is *less than* **1,000 pixels**, then darc will by default set the height as **1,000 pixels**.

Later darc will tell selenium to resize the window (in *headless* mode) to **1,024 pixels** in width and **110%** of the page height in height, and take a *PNG* screenshot.

See also:

- *darc.const.SE_EMPTY*
- *darc.const.SE_WAIT*

If the submission API is provided, *submit_selenium()* will be called and submit the document just loaded.

Later, *extract_links()* will be called then to extract all possible links from the HTML document and save such links into the `requests` database (c.f. *save_requests()*).

1.3 URL Utilities

The *Link* class is the key data structure of the `darc` project, it contains all information required to identify a URL's proxy type, hostname, path prefix when saving, etc.

The *link* module also provides several wrapper function to the `urllib.parse`.

class darc.link.**Link**(*url*, *proxy*, *url_parse*, *host*, *base*, *name*)

Bases: `object`

Parsed link.

Parameters

- **url** (*str*) – original link
- **proxy** (*str*) – proxy type
- **host** (*str*) – URL's hostname
- **base** (*str*) – base folder for saving files
- **name** (*str*) – hashed link for saving files
- **url_parse** (*urllib.parse.ParseResult*) – parsed URL from `urllib.parse.urlparse()`

Returns Parsed link object.

Return type *Link*

Note: *Link* is a dataclass object. It is safely *hashable*, through `hash(url)`.

__hash__()

Provide hash support to the *Link* object.

base: str = None

base folder for saving files

host: str = None

URL's hostname

name: str = None

hashed link for saving files

proxy: str = None

proxy type

url: str = None

original link

url_parse: urllib.parse.ParseResult = None
parsed URL from `urllib.parse.urlparse()`

`darc.link.`**parse_link**(*link*, *host=None*)
Parse link.

Parameters

- **link** (*str*) – link to be parsed
- **host** (*Optional[str]*) – hostname of the link

Returns The parsed link object.

Return type *darc.link.Link*

Note: If `host` is provided, it will override the hostname of the original `link`.

The parsing process of proxy type is as follows:

0. If `host` is `None` and the parse result from `urllib.parse.urlparse()` has no `netloc` (or hostname) specified, then set `hostname` as `(null)`; else set it as is.
1. If the scheme is `data`, then the `link` is a data URI, set `hostname` as `data` and `proxy` as `data`.
2. If the scheme is `javascript`, then the link is some JavaScript codes, set `proxy` as `script`.
3. If the scheme is `bitcoin`, then the link is a Bitcoin address, set `proxy` as `bitcoin`.
4. If the scheme is `ed2k`, then the link is an ED2K magnet link, set `proxy` as `ed2k`.
5. If the scheme is `magnet`, then the link is a magnet link, set `proxy` as `magnet`.
6. If the scheme is `mailto`, then the link is an email address, set `proxy` as `mail`.
7. If the scheme is `irc`, then the link is an IRC link, set `proxy` as `irc`.
8. If the scheme is **NOT** any of `http` or `https`, then set `proxy` to the scheme.
9. If the host is `None`, set `hostname` to `(null)`, set `proxy` to `null`.
10. If the host is an onion (`.onion`) address, set `proxy` to `tor`.
11. If the host is an I2P (`.i2p`) address, or any of `localhost:7657` and `localhost:7658`, set `proxy` to `i2p`.
12. If the host is *localhost* on `ZERONET_PORT`, and the path is not `/`, i.e. **NOT** root path, set `proxy` to `zeronet`; and set the first part of its path as `hostname`.

    Example:

    For a ZeroNet address, e.g. http://127.0.0.1:43110/1HeLLo4uzjaLetFx6NH3PMwFP3qbRbTf3D, *parse_link()* will parse the `hostname` as `1HeLLo4uzjaLetFx6NH3PMwFP3qbRbTf3D`.

13. If the host is *localhost* on `FREENET_PORT`, and the path is not `/`, i.e. **NOT** root path, set `proxy` to `freenet`; and set the first part of its path as `hostname`.

    Example:

    For a Freenet address, e.g. http://127.0.0.1:8888/USK@nwa8lHa271k2QvJ8aa0Ov7IHAV-DFOCFgmDt3X6BpCI,DuQSUZiI~agF8c-6tjsFFGuZ8eICrzWCILB60nT8KKo,AQACAAE/sone/77/, *parse_link()* will parse the `hostname` as `USK@nwa8lHa271k2QvJ8aa0Ov7IHAV-DFOCFgmDt3X6BpCI,DuQSUZiI~agF8c-6tjsFFGuZ8eICrzWCILB60nT8KKo,AQACAAE`.

14. If none of the cases above satisfied, the `proxy` will be set as `null`, marking it a plain normal link.

The `base` for parsed link *`Link`* object is defined as

```
<root>/<proxy>/<scheme>/<hostname>/
```

where `root` is *`PATH_DB`*.

The `name` for parsed link *`Link`* object is the sha256 hash (c.f. `hashlib.sha256()`) of the original `link`.

`darc.link.`**`quote`**(*string*, *safe='/'*, *encoding=None*, *errors=None*)

Wrapper function for `urllib.parse.quote()`.

Parameters

- **`string`** (*`AnyStr`*) – string to be quoted
- **`safe`** (*`AnyStr`*) – charaters not to escape
- **`encoding`** (*`Optional[str]`*) – string encoding
- **`errors`** (*`Optional[str]`*) – encoding error handler

Returns The quoted string.

Return type str

Note: The function suppressed possible errors when calling `urllib.parse.quote()`. If any, it will return the original string.

`darc.link.`**`unquote`**(*string*, *encoding='utf-8'*, *errors='replace'*)

Wrapper function for `urllib.parse.unquote()`.

Parameters

- **`string`** (*`AnyStr`*) – string to be unquoted
- **`encoding`** (*`str`*) – string encoding
- **`errors`** (*`str`*) – encoding error handler

Returns The quoted string.

Return type str

Note: The function suppressed possible errors when calling `urllib.parse.unquote()`. If any, it will return the original string.

`darc.link.`**`urljoin`**(*base*, *url*, *allow_fragments=True*)

Wrapper function for `urllib.parse.urljoin()`.

Parameters

- **`base`** (*`AnyStr`*) – base URL
- **`url`** (*`AnyStr`*) – URL to be joined
- **`allow_fragments`** (*`bool`*) – if allow fragments

Returns The joined URL.

Return type str

Note: The function suppressed possible errors when calling `urllib.parse.urljoin()`. If any, it will return `base/url` directly.

`darc.link.`**`urlparse`**(*url*, *scheme=''*, *allow_fragments=True*)
Wrapper function for `urllib.parse.urlparse()`.

Parameters

- **`url`** (*`str`*) – URL to be parsed
- **`scheme`** (*`str`*) – URL scheme
- **`allow_fragments`** (*`bool`*) – if allow fragments

Returns The parse result.

Return type urllib.parse.ParseResult

Note: The function suppressed possible errors when calling `urllib.parse.urlparse()`. If any, it will return `urllib.parse.ParseResult(scheme=scheme, netloc='', path=url, params='', query='', fragment='')` directly.

1.4 Source Parsing

The *`darc.parse`* module provides auxiliary functions to read `robots.txt`, sitemaps and HTML documents. It also contains utility functions to check if the proxy type, hostname and content type if in any of the black and white lists.

`darc.parse.`**`_check`**(*temp_list*)
Check hostname and proxy type of links.

Parameters **`temp_list`** (*`List[str]`*) – List of links to be checked.

Returns List of links matches the requirements.

Return type List[str]

Note: If *`CHECK_NG`* is `True`, the function will directly call *`_check_ng()`* instead.

See also:

- *`darc.parse.match_host()`*
- *`darc.parse.match_proxy()`*

`darc.parse.`**`_check_ng`**(*temp_list*)
Check content type of links through `HEAD` requests.

Parameters **`temp_list`** (*`List[str]`*) – List of links to be checked.

Returns List of links matches the requirements.

Return type List[str]

See also:

- *darc.parse.match_host()*
- *darc.parse.match_proxy()*
- *darc.parse.match_mime()*

darc.parse.**check_robots**(*link*)
Check if `link` is allowed in `robots.txt`.

Parameters **link** (*darc.link.Link*) – The link object to be checked.

Returns If `link` is allowed in `robots.txt`.

Return type bool

Note: The root path of a URL will always return `True`.

---

darc.parse.**extract_links**(*link*, *html*, *check=False*)
Extract links from HTML document.

Parameters

- **link** (*str*) – Original link of the HTML document.
- **html** (*Union[str, bytes]*) – Content of the HTML document.
- **check** (*bool*) – If perform checks on extracted links, default to *CHECK*.

Returns An iterator of extracted links.

Return type Iterator[str]

See also:

- *darc.parse._check()*
- *darc.parse._check_ng()*

darc.parse.**get_content_type**(*response*)
Get content type from `response`.

Parameters **response** (*requests.Response.*) – Response object.

Returns The content type from `response`.

Return type str

Note: If the `Content-Type` header is not defined in `response`, the function will utilise `magic` to detect its content type.

---

darc.parse.**get_sitemap**(*link*, *text*, *host=None*)
Fetch link to other sitemaps from a sitemap.

Parameters

- **link** (*str*) – Original link to the sitemap.
- **text** (*str*) – Content of the sitemap.
- **host** (*Optional[str]*) – Hostname of the URL to the sitemap, the value may not be same as in `link`.

Returns List of link to sitemaps.

Return type List[*darc.link.Link*]

Note: As specified in the sitemap protocol, it may contain links to other sitemaps.*[0]

darc.parse.**match_host**(*host*)
Check if hostname in black list.

Parameters **host** (*str*) – Hostname to be checked.

Returns If `host` in black list.

Return type bool

Note: If `host` is `None`, then it will always return `True`.

See also:

- *darc.const.LINK_WHITE_LIST*
- *darc.const.LINK_BLACK_LIST*
- *darc.const.LINK_FALLBACK*

darc.parse.**match_mime**(*mime*)
Check if content type in black list.

Parameters **mime** (*str*) – Content type to be checked.

Returns If `mime` in black list.

Return type bool

See also:

- *darc.const.MIME_WHITE_LIST*
- *darc.const.MIME_BLACK_LIST*
- *darc.const.MIME_FALLBACK*

darc.parse.**match_proxy**(*proxy*)
Check if proxy type in black list.

Parameters **proxy** (*str*) – Proxy type to be checked.

Returns If `proxy` in black list.

Return type bool

Note: If `proxy` is `script`, then it will always return `True`.

See also:

- *darc.const.PROXY_WHITE_LIST*

[0] https://www.sitemaps.org/protocol.html#index

- *darc.const.PROXY_BLACK_LIST*
- *darc.const.PROXY_FALLBACK*

darc.parse.**read_robots**(*link*, *text*, *host=None*)

Read `robots.txt` to fetch link to sitemaps.

Parameters

- **link** (*str*) – Original link to `robots.txt`.
- **text** (*str*) – Content of `robots.txt`.
- **host** (*Optional[str]*) – Hostname of the URL to `robots.txt`, the value may not be same as in `link`.

Returns List of link to sitemaps.

Return type List[*darc.link.Link*]

Note: If the link to sitemap is not specified in `robots.txt`†[0], the fallback link `/sitemap.xml` will be used.

---

darc.parse.**read_sitemap**(*link*, *text*, *check=False*)

Read sitemap.

Parameters

- **link** (*str*) – Original link to the sitemap.
- **text** (*str*) – Content of the sitemap.
- **check** (*bool*) – If perform checks on extracted links, default to *CHECK*.

Returns List of links extracted.

Return type Iterator[str]

See also:

- *darc.parse._check()*
- *darc.parse._check_ng()*

1.5 Source Saving

The *darc.save* module contains the core utilities for managing fetched files and documents.

The data storage under the root path (*PATH_DB*) is typically as following:

```
data
├── _queue_requests.txt
├── _queue_requests.txt.tmp
├── _queue_selenium.txt
├── _queue_selenium.txt.tmp
├── api
│   └── <proxy>
```

(continues on next page)

[0] https://www.sitemaps.org/protocol.html#submit_robots

(continued from previous page)

```
│       └── <scheme>
│           └── <hostname>
│               ├── new_host
│               │   └── <hash>_<timestamp>.json
│               ├── requests
│               │   └── <hash>_<timestamp>.json
│               └── selenium
│                   └── <hash>_<timestamp>.json
├── link.csv
├── misc
│   ├── bitcoin.txt
│   ├── data
│   │   └── <hash>_<timestamp>.<ext>
│   ├── ed2k.txt
│   ├── invalid.txt
│   ├── irc.txt
│   ├── magnet.txt
│   └── mail.txt
└── <proxy>
    └── <scheme>
        └── <hostname>
            ├── <hash>_<timestamp>.dat
            ├── <hash>_<timestamp>.json
            ├── <hash>_<timestamp>_raw.html
            ├── <hash>_<timestamp>.html
            ├── <hash>_<timestamp>.png
            ├── robots.txt
            └── sitemap_<hash>.xml
```

darc.save.**has_folder**(*link*)

Check if is a new host.

Parameters **link** (darc.link.Link) – Link object to check if is a new host.

Returns

- If `link` is a new host, return *link.base*.
- If not, return `None`.

Return type Optional[str]

darc.save.**has_html**(*time*, *link*)

Check if we need to re-craw the link by `selenium`.

Parameters

- **link** (darc.link.Link) – Link object to check if we need to re-craw the link by `selenium`.
- **time** (*NewType.<locals>.new_type*) –

Returns

- If no need, return the path to the document, i.e. `<root>/<proxy>/<scheme>/<hostname>/<hash>_<timestamp>.html`.
- If needed, return `None`.

Return type Optional[str]

See also:

- *darc.const.TIME_CACHE*

darc.save.**has_raw**(*time*, *link*)
Check if we need to re-craw the link by requests.

Parameters

- **link** (darc.link.Link) – Link object to check if we need to re-craw the link by requests.
- **time** (*NewType.<locals>.new_type*) –

Returns

- If no need, return the path to the document, i.e. <root>/<proxy>/<scheme>/<hostname>/<hash>_<timestamp>_raw.html, or <root>/<proxy>/<scheme>/<hostname>/<hash>_<timestamp>.dat.
- If needed, return None.

Return type Optional[str]

See also:

- *darc.const.TIME_CACHE*

darc.save.**has_robots**(*link*)
Check if robots.txt already exists.

Parameters **link** (darc.link.Link) – Link object to check if robots.txt already exists.

Returns

- If robots.txt exists, return the path to robots.txt, i.e. <root>/<proxy>/<scheme>/<hostname>/robots.txt.
- If not, return None.

Return type Optional[str]

darc.save.**has_sitemap**(*link*)
Check if sitemap already exists.

Parameters **link** (darc.link.Link) – Link object to check if sitemap already exists.

Returns

- If sitemap exists, return the path to the sitemap, i.e. <root>/<proxy>/<scheme>/<hostname>/sitemap_<hash>.xml.
- If not, return None.

Return type Optional[str]

darc.save.**sanitise**(*link*, *time=None*, *raw=False*, *data=False*, *headers=False*, *screenshot=False*)
Sanitise link to path.

Parameters

- **link** (darc.link.Link) – Link object to sanitise the path
- **time** (*datetime*) – Timestamp for the path.
- **raw** (*bool*) – If this is a raw HTML document from requests.
- **data** (*bool*) – If this is a generic content type document.

- **headers** (*bool*) – If this is response headers from `requests`.
- **screenshot** (*bool*) – If this is the screenshot from `selenium`.

Returns

- If `raw` is `True`, `<root>/<proxy>/<scheme>/<hostname>/<hash>_<timestamp>_raw.html`.
- If `data` is `True`, `<root>/<proxy>/<scheme>/<hostname>/<hash>_<timestamp>.dat`.
- If `headers` is `True`, `<root>/<proxy>/<scheme>/<hostname>/<hash>_<timestamp>.json`.
- If `screenshot` is `True`, `<root>/<proxy>/<scheme>/<hostname>/<hash>_<timestamp>.png`.
- If none above, `<root>/<proxy>/<scheme>/<hostname>/<hash>_<timestamp>.html`.

Return type str

See also:

- *darc.crawl.crawler()*
- *darc.crawl.loader()*

darc.save.**save_file**(*time*, *link*, *content*)

Save file.

The function will also try to make symbolic links from the saved file standard path to the relative path as in the URL.

Parameters

- **time** (*datetime*) – Timestamp of generic file.
- **link** (`darc.link.Link`) – Link object of original URL.
- **content** (*bytes*) – Content of generic file.

Returns Saved path to generic content type file, `<root>/<proxy>/<scheme>/<hostname>/<hash>_<timestamp>.dat`.

Return type str

See also:

- *darc.save.sanitise()*
- *darc.crawl.crawler()*

darc.save.**save_headers**(*time*, *link*, *response*, *session*)

Save HTTP response headers.

Parameters

- **time** (*datetime*) – Timestamp of response.
- **link** (`darc.link.Link`) – Link object of response.
- **response** (`requests.Response`) – Response object to be saved.
- **session** (`requests.Session`) – Session object of response.

Returns Saved path to response headers, i.e. `<root>/<proxy>/<scheme>/<hostname>/<hash>_<timestamp>.json`.

Return type str

The JSON data saved is as following:

```
{
    "[metadata]": {
        "url": "...",
        "proxy": "...",
        "host": "...",
        "base": "...",
        "name": "..."
    },
    "Timestamp": "...",
    "URL": "...",
    "Method": "GET",
    "Status-Code": "...",
    "Reason": "...",
    "Cookies": {
        "...": "..."
    },
    "Session": {
        "...": "..."
    },
    "Request": {
        "...": "..."
    },
    "Response": {
        "...": "..."
    }
}
```

See also:

- *darc.save.sanitise()*
- *darc.crawl.crawler()*

darc.save.**save_html**(*time*, *link*, *html*, *raw=False*)

Save response.

Parameters

- **time** (*datetime*) – Timestamp of HTML document.
- **link** (darc.link.Link) – Link object of original URL.
- **html** (*Union[str, bytes]*) – Content of HTML document.
- **raw** (*bool*) – If is fetched from requests.

Returns

Saved path to HTML document.

- If raw is True, `<root>/<proxy>/<scheme>/<hostname>/<hash>_<timestamp>_raw.html`.
- If not, `<root>/<proxy>/<scheme>/<hostname>/<hash>_<timestamp>.html`.

Return type str

See also:

- *darc.save.sanitise()*
- *darc.crawl.crawler()*
- *darc.crawl.loader()*

darc.save.**save_link**(*link*)
Save link hash database `link.csv`.

The CSV file has following fields:

- proxy type: *link.proxy*
- URL scheme: *link.url_parse.scheme*
- hostname: *link.base*
- link hash: *link.name*
- original URL: *link.url*

Parameters **link** (darc.link.Link) – Link object to be saved.

See also:

- *darc.const.PATH_LN*
- *darc.save._SAVE_LOCK*

darc.save.**save_robots**(*link*, *text*)
Save `robots.txt`.

Parameters

- **link** (darc.link.Link) – Link object of `robots.txt`.
- **text** (*str*) – Content of `robots.txt`.

Returns Saved path to `robots.txt`, i.e. `<root>/<proxy>/<scheme>/<hostname>/robots.txt`.

Return type str

See also:

- *darc.save.sanitise()*

darc.save.**save_sitemap**(*link*, *text*)
Save sitemap.

Parameters

- **link** (darc.link.Link) – Link object of sitemap.
- **text** (*str*) – Content of sitemap.

Returns Saved path to sitemap, i.e. `<root>/<proxy>/<scheme>/<hostname>/sitemap_<hash>.xml`.

Return type str

See also:

- *darc.save.sanitise()*

darc.save.**_SAVE_LOCK: multiprocessing.Lock**
I/O lock for saving link hash database `link.csv`.

See also:

- *darc.save.save_link()*

1.6 Link Database

The `darc` project utilises file system based database to provide tele-process communication.

Note: In its first implementation, the `darc` project used `multiprocessing.Queue` to support such communication. However, as noticed when runtime, the `multiprocessing.Queue` object will be much affected by the lack of memory.

There will be two databases, both locate at root of the data storage path *PATH_DB*:

- the `requests` database – `queue_requests.txt`
- the `selenium` database – `queue_selenium.txt`

At runtime, after reading such database, `darc` will keep a backup of the database with `.tmp` suffix to its file extension.

darc.db.**load_requests**()
Load link from the `requests` database.

After loading, `darc` will backup the original database `queue_requests.txt` as `queue_requests.txt.tmp` and empty the loaded database.

Returns List of loaded links from the `requests` database.

Return type List[str]

Note: Lines start with # will be considered as comments. Empty lines and comment lines will be ignored when loading.

---

darc.db.**load_selenium**()
Load link from the `selenium` database.

After loading, `darc` will backup the original database `queue_selenium.txt` as `queue_selenium.txt.tmp` and empty the loaded database.

Returns List of loaded links from the `selenium` database.

Return type List[str]

Note: Lines start with # will be considered as comments. Empty lines and comment lines will be ignored when loading.

---

darc.db.**save_requests**(*entries*, *single=False*)
Save link to the requests database.

Parameters

- **entries** (*Iterable[str]*) – Links to be added to the requests database. It can be either an *iterable* of links, or a single link string (if single set as True).
- **single** (*bool*) – Indicate if entries is an *iterable* of links or a single link string.

darc.db.**save_selenium**(*entries*, *single=False*)
Save link to the selenium database.

Parameters

- **entries** (*Iterable[str]*) – Links to be added to the selenium database. It can be either an *iterable* of links, or a single link string (if single set as True).
- **single** (*bool*) – Indicate if entries is an *iterable* of links or a single link string.

darc.db.**QR_LOCK: multiprocessing.Lock**
I/O lock for the requests database _queue_requests.txt.

See also:

- *darc.db.save_requests()*

darc.db.**QS_LOCK: Union[multiprocessing.Lock, threading.Lock, contextlib.nullcontext]**
I/O lock for the selenium database _queue_selenium.txt.

If *FLAG_MP* is True, it will be an instance of multiprocessing.Lock. If *FLAG_TH* is True, it will be an instance of threading.Lock. If none above, it will be an instance of contextlib.nullcontext.

See also:

- *darc.db.save_selenium()*
- *darc.const.FLAG_MP*
- *darc.const.FLAG_TH*

1.7 Data Submission

The darc project integrates the capability of submitting fetched data and information to a web server, to support real-time cross-analysis and status display.

There are three submission events:

1. New Host Submission – *API_NEW_HOST*

   Submitted in *crawler()* function call, when the crawling URL is marked as a new host.

2. Requests Submission – *API_REQUESTS*

   Submitted in *crawler()* function call, after the crawling process of the URL using requests.

3. Selenium Submission – *API_SELENIUM*

   Submitted in *loader()* function call, after the loading process of the URL using selenium.

darc.submit.**get_html**(*link*, *time*)
Read HTML document.

Parameters

- **link** (darc.link.Link) – Link object to read document from selenium.
- **time** (*str*) –

Returns

- If document exists, return the data from document.
  - path – relative path from document to root of data storage *PATH_DB*, <proxy>/<scheme>/<hostname>/<hash>_<timestamp>.html
  - data – *base64* encoded content of document
- If not, return None.

Return type Optional[Dict[str, Union[str, ByteString]]]

See also:

- *darc.crawl.loader()*
- *darc.save.save_html()*

darc.submit.**get_metadata**(*link*)
Generate metadata field.

Parameters **link** (darc.link.Link) – Link object to generate metadata.

Returns

The metadata from link.

- url – original URL, *link.url*
- proxy – proxy type, *link.proxy*
- host – hostname, *link.host*
- base – base path, *link.base*
- name – link hash, *link.name*

Return type Dict[str, str]

darc.submit.**get_raw**(*link*, *time*)
Read raw document.

Parameters

- **link** (darc.link.Link) – Link object to read document from requests.
- **time** (*str*) –

Returns

- If document exists, return the data from document.
  - path – relative path from document to root of data storage *PATH_DB*, <proxy>/<scheme>/<hostname>/<hash>_<timestamp>_raw.html or <proxy>/<scheme>/<hostname>/<hash>_<timestamp>.dat
  - data – *base64* encoded content of document
- If not, return None.

Return type Optional[Dict[str, Union[str, ByteString]]]

See also:

- *darc.crawl.crawler()*
- *darc.save.save_html()*
- *darc.save.save_file()*

darc.submit.**get_robots**(*link*)
Read robots.txt.

Parameters **link** (darc.link.Link) – Link object to read robots.txt.

Returns

- If robots.txt exists, return the data from robots.txt.
  - path – relative path from robots.txt to root of data storage *PATH_DB*, <proxy>/<scheme>/<hostname>/robots.txt
  - data – *base64* encoded content of robots.txt
- If not, return None.

Return type Optional[Dict[str, Union[str, ByteString]]]

See also:

- *darc.crawl.crawler()*
- *darc.save.save_robots()*

darc.submit.**get_screenshot**(*link*, *time*)
Read screenshot picture.

Parameters

- **link** (darc.link.Link) – Link object to read screenshot from selenium.
- **time** (*str*) –

Returns

- If screenshot exists, return the data from screenshot.
  - path – relative path from screenshot to root of data storage *PATH_DB*, <proxy>/<scheme>/<hostname>/<hash>_<timestamp>.png
  - data – *base64* encoded content of screenshot
- If not, return None.

Return type Optional[Dict[str, Union[str, ByteString]]]

See also:

- *darc.crawl.loader()*

darc.submit.**get_sitemap**(*link*)
Read sitemaps.

Parameters **link** (darc.link.Link) – Link object to read sitemaps.

Returns

- If sitemaps exist, return list of the data from sitemaps.

- `path` – relative path from sitemap to root of data storage *PATH_DB*, `<proxy>/<scheme>/<hostname>/sitemap_<hash>.xml`
- `data` – *base64* encoded content of sitemap

- If not, return `None`.

Return type Optional[List[Dict[str, Union[str, ByteString]]]]

See also:

- *darc.crawl.crawler()*
- *darc.save.save_sitemap()*

`darc.submit.`**`save_submit`**(*domain*, *data*)
Save failed submit data.

Parameters

- **domain** (`'new_host'`, `'requests'` or `'selenium'`) – Domain of the submit data.
- **data** (*Dict[str, Any]*) – Submit data.

See also:

- *darc.submit.PATH_API*
- *darc.submit.submit()*
- *darc.submit.submit_new_host()*
- *darc.submit.submit_requests()*
- *darc.submit.submit_selenium()*

`darc.submit.`**`submit`**(*api*, *domain*, *data*)
Submit data.

Parameters

- **api** (*str*) – API URL.
- **domain** (`'new_host'`, `'requests'` or `'selenium'`) – Domain of the submit data.
- **data** (*Dict[str, Any]*) – Submit data.

See also:

- *darc.submit.API_RETRY*
- *darc.submit.save_submit()*
- *darc.submit.submit_new_host()*
- *darc.submit.submit_requests()*
- *darc.submit.submit_selenium()*

`darc.submit.`**`submit_new_host`**(*time*, *link*)
Submit new host.

When a new host is discovered, the `darc` crawler will submit the host information. Such includes `robots.txt` (if exists) and `sitemap.xml` (if any).

Parameters

- **time** (*datetime.datetime*) – Timestamp of submission.
- **link** (darc.link.Link) – Link object of submission.

If *API_NEW_HOST* is None, the data for submission will directly be save through *save_submit()*.

The data submitted should have following format:

```
{
    // metadata of URL
    "[metadata]": {
        // original URL - <scheme>://<netloc>/<path>;<params>?<query>#<fragment>
        "url": ...,
        // proxy type - null / tor / i2p / zeronet / freenet
        "proxy": ...,
        // hostname / netloc, c.f. ``urllib.parse.urlparse``
        "host": ...,
        // base folder, relative path (to data root path ``PATH_DATA``) in␣
↪container - <proxy>/<scheme>/<host>
        "base": ...,
        // sha256 of URL as name for saved files (timestamp is in ISO format)
        //   JSON log as this one - <base>/<name>_<timestamp>.json
        //   HTML from requests - <base>/<name>_<timestamp>_raw.html
        //   HTML from selenium - <base>/<name>_<timestamp>.html
        //   generic data files - <base>/<name>_<timestamp>.dat
        "name": ...
    },
    // requested timestamp in ISO format as in name of saved file
    "Timestamp": ...,
    // original URL
    "URL": ...,
    // robots.txt from the host (if not exists, then ``null``)
    "Robots": {
        // path of the file, relative path (to data root path ``PATH_DATA``) in␣
↪container
        //   - <proxy>/<scheme>/<host>/robots.txt
        "path": ...,
        // content of the file (**base64** encoded)
        "data": ...,
    },
    // sitemaps from the host (if none, then ``null``)
    "Sitemaps": [
        {
            // path of the file, relative path (to data root path ``PATH_DATA``)␣
↪in container
            //   - <proxy>/<scheme>/<host>/sitemap_<name>.txt
            "path": ...,
            // content of the file (**base64** encoded)
            "data": ...,
        },
        ...
    ],
    // hosts.txt from the host (if proxy type is ``i2p``; if not exists, then␣
↪``null``)
    "Hosts": {
        // path of the file, relative path (to data root path ``PATH_DATA``) in␣
↪container
        //   - <proxy>/<scheme>/<host>/hosts.txt
        "path": ...,
```

(continues on next page)

(continued from previous page)

```
        // content of the file (**base64** encoded)
        "data": ...,
    }
}
```

See also:

- *darc.submit.API_NEW_HOST*
- *darc.submit.submit()*
- *darc.submit.save_submit()*
- *darc.submit.get_metadata()*
- *darc.submit.get_robots()*
- *darc.proxy.i2p.get_hosts()*

darc.submit.**submit_requests**(*time*, *link*, *response*, *session*)

Submit requests data.

When crawling, we'll first fetch the URl using `requests`, to check its availability and to save its HTTP headers information. Such information will be submitted to the web UI.

Parameters

- **time** (*datetime.datetime*) – Timestamp of submission.
- **link** (darc.link.Link) – Link object of submission.
- **response** (requests.Response) – Response object of submission.
- **session** (requests.Session) – Session object of submission.

If *API_REQUESTS* is `None`, the data for submission will directly be save through *save_submit()*.

The data submitted should have following format:

```
{
    // metadata of URL
    "[metadata]": {
        // original URL - <scheme>://<netloc>/<path>;<params>?<query>#<fragment>
        "url": ...,
        // proxy type - null / tor / i2p / zeronet / freenet
        "proxy": ...,
        // hostname / netloc, c.f. ``urllib.parse.urlparse``
        "host": ...,
        // base folder, relative path (to data root path ``PATH_DATA``) in␣
↪containter - <proxy>/<scheme>/<host>
        "base": ...,
        // sha256 of URL as name for saved files (timestamp is in ISO format)
        //   JSON log as this one - <base>/<name>_<timestamp>.json
        //   HTML from requests - <base>/<name>_<timestamp>_raw.html
        //   HTML from selenium - <base>/<name>_<timestamp>.html
        //   generic data files - <base>/<name>_<timestamp>.dat
        "name": ...
    },
    // requested timestamp in ISO format as in name of saved file
    "Timestamp": ...,
    // original URL
```

(continues on next page)

(continued from previous page)

```
    "URL": ...,
    // request method
    "Method": "GET",
    // response status code
    "Status-Code": ...,
    // response reason
    "Reason": ...,
    // response cookies (if any)
    "Cookies": {
        ...
    },
    // session cookies (if any)
    "Session": {
        ...
    },
    // request headers (if any)
    "Request": {
        ...
    },
    // response headers (if any)
    "Response": {
        ...
    },
    // requested file (if not exists, then ``null``)
    "Document": {
        // path of the file, relative path (to data root path ``PATH_DATA``) in␣
↪container
        //   - <proxy>/<scheme>/<host>/<name>_<timestamp>_raw.html
        // or if the document is of generic content type, i.e. not HTML
        //   - <proxy>/<scheme>/<host>/<name>_<timestamp>.dat
        "path": ...,
        // content of the file (**base64** encoded)
        "data": ...,
    }
}
```

See also:

- *darc.submit.API_REQUESTS*
- *darc.submit.submit()*
- *darc.submit.save_submit()*
- *darc.submit.get_metadata()*
- *darc.submit.get_raw()*
- *darc.crawl.crawler()*

darc.submit.**submit_selenium**(*time*, *link*)

Submit selenium data.

After crawling with `requests`, we'll then render the URl using `selenium` with Google Chrome and its web driver, to provide a fully rendered web page. Such information will be submitted to the web UI.

Parameters

- **time** (*datetime.datetime*) – Timestamp of submission.

- **link** (darc.link.Link) – Link object of submission.

If *API_SELENIUM* is None, the data for submission will directly be save through *save_submit()*.

Note: This information is optional, only provided if the content type from requests is HTML, status code not between 400 and 600, and HTML data not empty.

The data submitted should have following format:

```
{
    // metadata of URL
    "[metadata]": {
        // original URL - <scheme>://<netloc>/<path>;<params>?<query>#<fragment>
        "url": ...,
        // proxy type - null / tor / i2p / zeronet / freenet
        "proxy": ...,
        // hostname / netloc, c.f. ``urllib.parse.urlparse``
        "host": ...,
        // base folder, relative path (to data root path ``PATH_DATA``) in␣
↪container - <proxy>/<scheme>/<host>
        "base": ...,
        // sha256 of URL as name for saved files (timestamp is in ISO format)
        //   JSON log as this one - <base>/<name>_<timestamp>.json
        //   HTML from requests - <base>/<name>_<timestamp>_raw.html
        //   HTML from selenium - <base>/<name>_<timestamp>.html
        //   generic data files - <base>/<name>_<timestamp>.dat
        "name": ...
    },
    // requested timestamp in ISO format as in name of saved file
    "Timestamp": ...,
    // original URL
    "URL": ...,
    // rendered HTML document (if not exists, then ``null``)
    "Document": {
        // path of the file, relative path (to data root path ``PATH_DATA``) in␣
↪container
        //   - <proxy>/<scheme>/<host>/<name>_<timestamp>.html
        "path": ...,
        // content of the file (**base64** encoded)
        "data": ...,
    },
    // web page screenshot (if not exists, then ``null``)
    "Screenshot": {
        // path of the file, relative path (to data root path ``PATH_DATA``) in␣
↪container
        //   - <proxy>/<scheme>/<host>/<name>_<timestamp>.png
        "path": ...,
        // content of the file (**base64** encoded)
        "data": ...,
    }
}
```

See also:

- *darc.submit.API_SELENIUM*
- *darc.submit.submit()*

- *darc.submit.save_submit()*
- *darc.submit.get_metadata()*
- *darc.submit.get_html()*
- *darc.submit.get_screenshot()*
- *darc.crawl.loader()*

darc.submit.**PATH_API = '{PATH_DB}/api/'**
Path to the API submittsion records, i.e. `api` folder under the root of data storage.

See also:

- *darc.const.PATH_DB*

darc.submit.**API_RETRY: int**
Retry times for API submission when failure.

Default `3`

Environ *API_RETRY*

darc.submit.**API_NEW_HOST: str**
API URL for *submit_new_host()*.

Default `None`

Environ *API_NEW_HOST*

darc.submit.**API_REQUESTS: str**
API URL for *submit_requests()*.

Default `None`

Environ *API_REQUESTS*

darc.submit.**API_SELENIUM: str**
API URL for *submit_selenium()*.

Default `None`

Environ *API_SELENIUM*

Note: If *API_NEW_HOST*, *API_REQUESTS* and *API_SELENIUM* is `None`, the corresponding submit function will save the JSON data in the path specified by *PATH_API*.

See also:

The `darc` provides a demo on how to implement a `darc`-compliant web backend for the data submission module. See the *demo* page for more information.

1.8 Requests Wrapper

The *darc.requests* module wraps around the requests module, and provides some simple interface for the darc project.

darc.requests.**i2p_session**(*futures=False*)
I2P (.i2p) session.

Parameters **futures** (*bool*) – If returns a requests_futures.FuturesSession.

Returns The session object with I2P proxy settings.

Return type Union[requests.Session, requests_futures.FuturesSession]

See also:

- darc.proxy.i2p.I2P_REQUESTS_PROXY

darc.requests.**null_session**(*futures=False*)
No proxy session.

Parameters **futures** (*bool*) – If returns a requests_futures.FuturesSession.

Returns The session object with no proxy settings.

Return type Union[requests.Session, requests_futures.FuturesSession]

darc.requests.**request_session**(*link*, *futures=False*)
Get requests session.

Parameters

- **link** (darc.link.Link) – Link requesting for requests.Session.
- **futures** (*bool*) – If returns a requests_futures.FuturesSession.

Returns The session object with corresponding proxy settings.

Return type Union[requests.Session, requests_futures.FuturesSession]

Raises ***UnsupportedLink*** – If the proxy type of link if not specified in the *LINK_MAP*.

See also:

- *darc.proxy.LINK_MAP*

darc.requests.**tor_session**(*futures=False*)
Tor (.onion) session.

Parameters **futures** (*bool*) – If returns a requests_futures.FuturesSession.

Returns The session object with Tor proxy settings.

Return type Union[requests.Session, requests_futures.FuturesSession]

See also:

- darc.proxy.tor.TOR_REQUESTS_PROXY

1.9 Selenium Wrapper

The *darc.selenium* module wraps around the selenium module, and provides some simple interface for the darc project.

darc.selenium.**get_capabilities**(*type='null'*)
Generate desied capabilities.

Parameters **type** (*str*) – Proxy type for capabilities.

Returns The desied capabilities for the web driver selenium.webdriver.Chrome.

Raises ***UnsupportedProxy*** – If the proxy type is **NOT** null, tor or i2p.

Return type dict

See also:

- darc.proxy.tor.TOR_SELENIUM_PROXY
- darc.proxy.i2p.I2P_SELENIUM_PROXY

darc.selenium.**get_options**(*type='null'*)
Generate options.

Parameters **type** (*str*) – Proxy type for options.

Returns The options for the web driver selenium.webdriver.Chrome.

Return type selenium.webdriver.ChromeOptions

Raises

- ***UnsupportedPlatform*** – If the operation system is **NOT** macOS or Linux.
- ***UnsupportedProxy*** – If the proxy type is **NOT** null, tor or i2p.

See also:

- darc.proxy.tor.TOR_PORT
- darc.proxy.i2p.I2P_PORT

References

- Google Chrome command line switches
- Disable sandbox (--no-sandbox) when running as root user
  - https://crbug.com/638180
  - https://stackoverflow.com/a/50642913/7218152
- Disable usage of /dev/shm
  - http://crbug.com/715363
- Using Socks proxy

---

darc.selenium.**i2p_driver**()
I2P (.i2p) driver.

Returns The web driver object with I2P proxy settings.

Return type selenium.webdriver.Chrome

See also:

- *darc.selenium.get_options()*
- *darc.selenium.get_capabilities()*

darc.selenium.**null_driver**()
No proxy driver.

Returns The web driver object with no proxy settings.

Return type selenium.webdriver.Chrome

See also:

- *darc.selenium.get_options()*
- *darc.selenium.get_capabilities()*

darc.selenium.**request_driver**(*link*)
Get selenium driver.

Parameters **link** (darc.link.Link) – Link requesting for selenium.webdriver.Chrome.

Returns The web driver object with corresponding proxy settings.

Return type selenium.webdriver.Chrome

Raises ***UnsupportedLink*** – If the proxy type of link if not specified in the *LINK_MAP*.

See also:

- *darc.proxy.LINK_MAP*

darc.selenium.**tor_driver**()
Tor (.onion) driver.

Returns The web driver object with Tor proxy settings.

Return type selenium.webdriver.Chrome

See also:

- *darc.selenium.get_options()*
- *darc.selenium.get_capabilities()*

1.10 Proxy Utilities

The darc.proxy module provides various proxy support to the darc project.

1.10.1 Bitcoin Addresses

The *darc.proxy.bitcoin* module contains the auxiliary functions around managing and processing the bitcoin addresses.

Currently, the darc project directly save the bitcoin addresses extracted to the data storage file *PATH* without further processing.

darc.proxy.bitcoin.**save_bitcoin**(*link*)
Save bitcoin address.

The function will save bitcoin address to the file as defined in *PATH*.

Parameters **link** (darc.link.Link) – Link object representing the bitcoin address.

darc.proxy.bitcoin.**PATH = '{PATH_MISC}/bitcoin.txt'**
Path to the data storage of bitcoin addresses.

See also:

- *darc.const.PATH_MISC*

darc.proxy.bitcoin.**LOCK: multiprocessing.Lock**
I/O lock for saving bitcoin addresses *PATH*.

1.10.2 Data URI Schemes

The *darc.proxy.data* module contains the auxiliary functions around managing and processing the data URI schemes.

Currently, the darc project directly save the data URI schemes extracted to the data storage path *PATH* without further processing.

darc.proxy.data.**save_data**(*link*)
Save data URI.

The function will save data URIs to the data storage as defined in *PATH*.

Parameters **link** (darc.link.Link) – Link object representing the data URI.

darc.proxy.data.**PATH = '{PATH_MISC}/data/'**
Path to the data storage of data URI schemes.

See also:

- *darc.const.PATH_MISC*

1.10.3 ED2K Magnet Links

The *darc.proxy.ed2k* module contains the auxiliary functions around managing and processing the ED2K magnet links.

Currently, the darc project directly save the ED2K magnet links extracted to the data storage file *PATH* without further processing.

darc.proxy.ed2k.**save_ed2k**(*link*)
Save ed2k magnet link.

The function will save ED2K magnet link to the file as defined in *PATH*.

Parameters **link** (darc.link.Link) – Link object representing the ED2K magnet links.

darc.proxy.ed2k.**PATH = '{PATH_MISC}/ed2k.txt'**
Path to the data storage of bED2K magnet links.

See also:

- *darc.const.PATH_MISC*

darc.proxy.ed2k.**LOCK: multiprocessing.Lock**
I/O lock for saving ED2K magnet links *PATH*.

1.10.4 Freenet Proxy

The *darc.proxy.freenet* module contains the auxiliary functions around managing and processing the Freenet proxy.

darc.proxy.freenet.**_freenet_bootstrap**()
Freenet bootstrap.

The bootstrap arguments are defined as *_FREENET_ARGS*.

Raises **subprocess.CalledProcessError** – If the return code of *_FREENET_PROC* is non-zero.

See also:

- *darc.proxy.freenet.freenet_bootstrap()*
- *darc.proxy.freenet.BS_WAIT*
- *darc.proxy.freenet._FREENET_BS_FLAG*
- *darc.proxy.freenet._FREENET_PROC*

darc.proxy.freenet.**freenet_bootstrap**()
Bootstrap wrapper for Freenet.

The function will bootstrap the Freenet proxy. It will retry for *FREENET_RETRY* times in case of failure.

Also, it will **NOT** re-bootstrap the proxy as is guaranteed by *_FREENET_BS_FLAG*.

Warns **FreenetBootstrapFailed** – If failed to bootstrap Freenet proxy.

Raises ***UnsupportedPlatform*** – If the system is not supported, i.e. not macOS or Linux.

See also:

- *darc.proxy.freenet._freenet_bootstrap()*
- *darc.proxy.freenet.FREENET_RETRY*
- *darc.proxy.freenet._FREENET_BS_FLAG*

darc.proxy.freenet.**has_freenet**(*link_pool*)
Check if contain Freenet links.

Parameters **link_pool** (*Iterable[str]*) – Link pool to check.

Returns If the link pool contains Freenet links.

Return type bool

See also:

- *darc.link.parse_link()*
- *darc.link.urlparse()*
- *darc.proxy.freenet.FREENET_PORT*

The following constants are configuration through environment variables:

darc.proxy.freenet.**FREENET_PORT: int**
Port for Freenet proxy connection.

Default 8888

Environ *FREENET_PORT*

darc.proxy.freenet.**FREENET_RETRY: int**
Retry times for Freenet bootstrap when failure.

Default 3

Environ *FREENET_RETRY*

darc.proxy.freenet.**BS_WAIT: float**
Time after which the attempt to start Freenet is aborted.

Default 90

Environ FREENET_WAIT

Note: If not provided, there will be **NO** timeouts.

darc.proxy.freenet.**FREENET_PATH: str**
Path to the Freenet project.

Default /usr/local/src/freenet

Environ *FREENET_PATH*

darc.proxy.freenet.**FREENET_ARGS: List[str]**
Freenet bootstrap arguments for `run.sh start`.

If provided, it should be parsed as command line arguments (c.f. `shlex.split`).

Default ''

Environ *FREENET_ARGS*

Note: The command will be run as *DARC_USER*, if current user (c.f. `getpass.getuser()`) is *root*.

The following constants are defined for internal usage:

darc.proxy.freenet.**_FREENET_BS_FLAG: bool**
If the Freenet proxy is bootstrapped.

darc.proxy.freenet.**_FREENET_PROC: subprocess.Popen**
Freenet proxy process running in the background.

darc.proxy.freenet.**_FREENET_ARGS: List[str]**
Freenet proxy bootstrap arguments.

1.10.5 I2P Proxy

The *darc.proxy.i2p* module contains the auxiliary functions around managing and processing the I2P proxy.

darc.proxy.i2p.**_i2p_bootstrap**()
I2P bootstrap.

The bootstrap arguments are defined as *_I2P_ARGS*.

Raises **subprocess.CalledProcessError** – If the return code of *_I2P_PROC* is non-zero.

See also:

- *darc.proxy.i2p.i2p_bootstrap()*
- *darc.proxy.i2p.BS_WAIT*
- *darc.proxy.i2p._I2P_BS_FLAG*
- *darc.proxy.i2p._I2P_PROC*

darc.proxy.i2p.**fetch_hosts**(*link*)
Fetch `hosts.txt`.

Parameters **link** (darc.link.Link) – Link object to fetch for its `hosts.txt`.

darc.proxy.i2p.**get_hosts**(*link*)
Read `hosts.txt`.

Parameters **link** (darc.link.Link) – Link object to read `hosts.txt`.

Returns

- If `hosts.txt` exists, return the data from `hosts.txt`.
  - `path` – relative path from `hosts.txt` to root of data storage *PATH_DB*, `<proxy>/<scheme>/<hostname>/hosts.txt`
  - `data` – *base64* encoded content of `hosts.txt`
- If not, return `None`.

Return type Optional[Dict[str, Union[str, ByteString]]]

See also:

- *darc.submit.submit_new_host()*
- *darc.proxy.i2p.save_hosts()*

darc.proxy.i2p.**has_hosts**(*link*)
Check if `hosts.txt` already exists.

Parameters **link** (darc.link.Link) – Link object to check if `hosts.txt` already exists.

Returns

- If `hosts.txt` exists, return the path to `hosts.txt`, i.e. `<root>/<proxy>/<scheme>/<hostname>/hosts.txt`.
- If not, return `None`.

Return type Optional[str]

darc.proxy.i2p.**has_i2p**(*link_pool*)
Check if contain I2P links.

Parameters **link_pool** (*Set[str]*) – Link pool to check.

Returns If the link pool contains I2P links.

Return type bool

See also:

- *darc.link.parse_link()*
- *darc.link.urlparse()*

darc.proxy.i2p.**i2p_bootstrap**()
Bootstrap wrapper for I2P.

The function will bootstrap the I2P proxy. It will retry for *I2P_RETRY* times in case of failure.

Also, it will **NOT** re-bootstrap the proxy as is guaranteed by *_I2P_BS_FLAG*.

Warns **I2PBootstrapFailed** – If failed to bootstrap I2P proxy.

Raises ***UnsupportedPlatform*** – If the system is not supported, i.e. not macOS or Linux.

See also:

- *darc.proxy.i2p._i2p_bootstrap()*
- *darc.proxy.i2p.I2P_RETRY*
- *darc.proxy.i2p._I2P_BS_FLAG*

darc.proxy.i2p.**read_hosts**(*text*, *check=False*)
Read `hosts.txt`.

Parameters

- **text** (*Iterable[str]*) – Content of `hosts.txt`.
- **check** (*bool*) – If perform checks on extracted links, default to *CHECK*.

Returns List of links extracted.

Return type Iterable[str]

darc.proxy.i2p.**save_hosts**(*link*, *text*)
Save `hosts.txt`.

Parameters

- **link** (`darc.link.Link`) – Link object of `hosts.txt`.
- **text** (*str*) – Content of `hosts.txt`.

Returns Saved path to `hosts.txt`, i.e. `<root>/<proxy>/<scheme>/<hostname>/hosts.txt`.

Return type str

See also:

- *darc.save.sanitise()*

darc.proxy.i2p.**I2P_REQUESTS_PROXY: Dict[str, Any]**
Proxy for I2P sessions.

See also:

- *darc.requests.i2p_session()*

darc.proxy.i2p.**I2P_SELENIUM_PROXY: selenium.webdriver.Proxy**
Proxy (selenium.webdriver.Proxy) for I2P web drivers.

See also:

- *darc.selenium.i2p_driver()*

The following constants are configuration through environment variables:

darc.proxy.i2p.**I2P_PORT: int**
Port for I2P proxy connection.

Default 4444

Environ *I2P_PORT*

darc.proxy.i2p.**I2P_RETRY: int**
Retry times for I2P bootstrap when failure.

Default 3

Environ *I2P_RETRY*

darc.proxy.i2p.**BS_WAIT: float**
Time after which the attempt to start I2P is aborted.

Default 90

Environ I2P_WAIT

Note: If not provided, there will be **NO** timeouts.

---

darc.proxy.i2p.**I2P_ARGS: List[str]**
I2P bootstrap arguments for i2prouter start.

If provided, it should be parsed as command line arguments (c.f. shlex.split).

Default ''

Environ *I2P_ARGS*

Note: The command will be run as *DARC_USER*, if current user (c.f. getpass.getuser()) is *root*.

The following constants are defined for internal usage:

darc.proxy.i2p.**_I2P_BS_FLAG: bool**
If the I2P proxy is bootstrapped.

darc.proxy.i2p.**_I2P_PROC: subprocess.Popen**
I2P proxy process running in the background.

darc.proxy.i2p.**_I2P_ARGS: List[str]**
I2P proxy bootstrap arguments.

1.10.6 IRC Addresses

The *darc.proxy.irc* module contains the auxiliary functions around managing and processing the IRC addresses.

Currently, the darc project directly save the IRC addresses extracted to the data storage file *PATH* without further processing.

darc.proxy.irc.**save_irc**(*link*)
Save IRC address.

The function will save IRC address to the file as defined in *PATH*.

Parameters **link** (darc.link.Link) – Link object representing the IRC address.

darc.proxy.irc.**PATH = '{PATH_MISC}/irc.txt'**
Path to the data storage of IRC addresses.

See also:

- *darc.const.PATH_MISC*

darc.proxy.irc.**LOCK: multiprocessing.Lock**
I/O lock for saving IRC addresses *PATH*.

1.10.7 Magnet Links

The *darc.proxy.magnet* module contains the auxiliary functions around managing and processing the magnet links.

Currently, the darc project directly save the magnet links extracted to the data storage file *PATH* without further processing.

darc.proxy.magnet.**save_magnet**(*link*)
Save magnet link.

The function will save magnet link to the file as defined in *PATH*.

Parameters **link** (darc.link.Link) – Link object representing the magnet link

darc.proxy.magnet.**PATH = '{PATH_MISC}/magnet.txt'**
Path to the data storage of magnet links.

See also:

- *darc.const.PATH_MISC*

darc.proxy.magnet.**LOCK: multiprocessing.Lock**
I/O lock for saving magnet links *PATH*.

1.10.8 Email Addresses

The *darc.proxy.mail* module contains the auxiliary functions around managing and processing the email addresses.

Currently, the darc project directly save the email addresses extracted to the data storage file *PATH* without further processing.

darc.proxy.mail.**save_mail**(*link*)
Save email address.

The function will save email address to the file as defined in *PATH*.

Parameters **link** (darc.link.Link) – Link object representing the email address.

darc.proxy.mail.**PATH = '{PATH_MISC}/mail.txt'**
Path to the data storage of email addresses.

See also:

- *darc.const.PATH_MISC*

darc.proxy.mail.**LOCK: multiprocessing.Lock**
I/O lock for saving email addresses *PATH*.

1.10.9 No Proxy

The *darc.proxy.null* module contains the auxiliary functions around managing and processing normal websites with no proxy.

darc.proxy.null.**fetch_sitemap**(*link*)
Fetch sitemap.

The function will first fetch the robots.txt, then fetch the sitemaps accordingly.

Parameters **link** (darc.link.Link) – Link object to fetch for its sitemaps.

See also:

- *darc.parse.read_robots()*
- *darc.parse.read_sitemap()*
- *darc.parse.get_sitemap()*

darc.proxy.null.**save_invalid**(*link*)
Save link with invalid scheme.

The function will save link with invalid scheme to the file as defined in *PATH*.

Parameters **link** (darc.link.Link) – Link object representing the link with invalid scheme.

darc.proxy.null.**PATH = '{PATH_MISC}/invalid.txt'**
Path to the data storage of links with invalid scheme.

See also:

- *darc.const.PATH_MISC*

darc.proxy.null.**LOCK: multiprocessing.Lock**
I/O lock for saving links with invalid scheme *PATH*.

1.10.10 Tor Proxy

The *darc.proxy.tor* module contains the auxiliary functions around managing and processing the Tor proxy.

darc.proxy.tor.**_tor_bootstrap**()
Tor bootstrap.

The bootstrap configuration is defined as *_TOR_CONFIG*.

If *TOR_PASS* not provided, the function will request for it.

See also:

- *darc.proxy.tor.tor_bootstrap()*
- *darc.proxy.tor.BS_WAIT*
- *darc.proxy.tor.TOR_PASS*
- *darc.proxy.tor._TOR_BS_FLAG*
- *darc.proxy.tor._TOR_PROC*
- *darc.proxy.tor._TOR_CTRL*

darc.proxy.tor.**has_tor**(*link_pool*)
Check if contain Tor links.

Parameters **link_pool** (*Set[str]*) – Link pool to check.

Returns If the link pool contains Tor links.

Return type bool

See also:

- *darc.link.parse_link()*
- *darc.link.urlparse()*

darc.proxy.tor.**print_bootstrap_lines**(*line*)
Print Tor bootstrap lines.

Parameters **line** (*str*) – Tor bootstrap line.

darc.proxy.tor.**renew_tor_session**()
Renew Tor session.

darc.proxy.tor.**tor_bootstrap**()
Bootstrap wrapper for Tor.

The function will bootstrap the Tor proxy. It will retry for *TOR_RETRY* times in case of failure.

Also, it will **NOT** re-bootstrap the proxy as is guaranteed by *_TOR_BS_FLAG*.

Warns **TorBootstrapFailed** – If failed to bootstrap Tor proxy.

See also:

- *darc.proxy.tor._tor_bootstrap()*
- *darc.proxy.tor.TOR_RETRY*
- *darc.proxy.tor._TOR_BS_FLAG*

darc.proxy.tor.**TOR_REQUESTS_PROXY: Dict[str, Any]**
Proxy for Tor sessions.

See also:

- *darc.requests.tor_session()*

darc.proxy.tor.**TOR_SELENIUM_PROXY: selenium.webdriver.Proxy**
Proxy (selenium.webdriver.Proxy) for Tor web drivers.

See also:

- *darc.selenium.tor_driver()*

The following constants are configuration through environment variables:

darc.proxy.tor.**TOR_PORT: int**
Port for Tor proxy connection.

Default 9050

Environ *TOR_PORT*

darc.proxy.tor.**TOR_CTRL: int**
Port for Tor controller connection.

Default 9051

Environ *TOR_CTRL*

darc.proxy.tor.**TOR_STEM: bool**
If manage the Tor proxy through stem.

Default True

Environ *TOR_STEM*

darc.proxy.tor.**TOR_PASS: str**
Tor controller authentication token.

Default None

Environ *TOR_PASS*

Note: If not provided, it will be requested at runtime.

---

darc.proxy.tor.**TOR_RETRY: int**
Retry times for Tor bootstrap when failure.

Default 3

Environ *TOR_RETRY*

darc.proxy.tor.**BS_WAIT: float**
Time after which the attempt to start Tor is aborted.

Default 90

Environ TOR_WAIT

Note: If not provided, there will be **NO** timeouts.

---

darc.proxy.tor.**TOR_CFG: Dict[str, Any]**
Tor bootstrap configuration for stem.process.launch_tor_with_config().

Default {}

Environ *TOR_CFG*

Note: If provided, it will be parsed from a JSON encoded string.

The following constants are defined for internal usage:

darc.proxy.tor.**_TOR_BS_FLAG: bool**
If the Tor proxy is bootstrapped.

darc.proxy.tor.**_TOR_PROC: subprocess.Popen**
Tor proxy process running in the background.

darc.proxy.tor.**_TOR_CTRL: stem.control.Controller**
Tor controller process (stem.control.Controller) running in the background.

darc.proxy.tor.**_TOR_CONFIG: List[str]**
Tor bootstrap configuration for stem.process.launch_tor_with_config().

1.10.11 ZeroNet Proxy

The *darc.proxy.zeronet* module contains the auxiliary functions around managing and processing the ZeroNet proxy.

darc.proxy.zeronet.**_zeronet_bootstrap**()
ZeroNet bootstrap.

The bootstrap arguments are defined as *_ZERONET_ARGS*.

Raises **subprocess.CalledProcessError** – If the return code of *_ZERONET_PROC* is non-zero.

See also:

- *darc.proxy.zeronet.zeronet_bootstrap()*
- *darc.proxy.zeronet.BS_WAIT*
- *darc.proxy.zeronet._ZERONET_BS_FLAG*
- *darc.proxy.zeronet._ZERONET_PROC*

darc.proxy.zeronet.**has_zeronet**(*link_pool*)
Check if contain ZeroNet links.

Parameters **link_pool** (*Set[str]*) – Link pool to check.

Returns If the link pool contains ZeroNet links.

Return type bool

See also:

- *darc.link.parse_link()*
- *darc.link.urlparse()*
- *darc.proxy.zeronet.ZERONET_PORT*

darc.proxy.zeronet.**zeronet_bootstrap**()
Bootstrap wrapper for ZeroNet.

The function will bootstrap the ZeroNet proxy. It will retry for *ZERONET_RETRY* times in case of failure.

Also, it will **NOT** re-bootstrap the proxy as is guaranteed by *_ZERONET_BS_FLAG*.

Warns **ZeroNetBootstrapFailed** – If failed to bootstrap ZeroNet proxy.

Raises ***UnsupportedPlatform*** – If the system is not supported, i.e. not macOS or Linux.

See also:

- *darc.proxy.zeronet._zeronet_bootstrap()*
- *darc.proxy.zeronet.ZERONET_RETRY*
- *darc.proxy.zeronet._ZERONET_BS_FLAG*

The following constants are configuration through environment variables:

darc.proxy.zeronet.**ZERONET_PORT: int**
Port for ZeroNet proxy connection.

Default 43110

Environ *ZERONET_PORT*

darc.proxy.zeronet.**ZERONET_RETRY: int**
Retry times for ZeroNet bootstrap when failure.

Default 3

Environ *ZERONET_RETRY*

darc.proxy.zeronet.**BS_WAIT: float**
Time after which the attempt to start ZeroNet is aborted.

Default 90

Environ ZERONET_WAIT

Note: If not provided, there will be **NO** timeouts.

---

darc.proxy.zeronet.**ZERONET_PATH: str**
Path to the ZeroNet project.

Default /usr/local/src/zeronet

Environ *ZERONET_PATH*

darc.proxy.zeronet.**ZERONET_ARGS: List[str]**
ZeroNet bootstrap arguments for `run.sh start`.

If provided, it should be parsed as command line arguments (c.f. `shlex.split`).

Default ''

Environ *ZERONET_ARGS*

Note: The command will be run as *DARC_USER*, if current user (c.f. `getpass.getuser()`) is *root*.

The following constants are defined for internal usage:

darc.proxy.zeronet.**_ZERONET_BS_FLAG: bool**
If the ZeroNet proxy is bootstrapped.

darc.proxy.zeronet.**_ZERONET_PROC: subprocess.Popen**
ZeroNet proxy process running in the background.

darc.proxy.zeronet.**_ZERONET_ARGS: List[str]**
ZeroNet proxy bootstrap arguments.

To tell the darc project which proxy settings to be used for the requests.Session objects and selenium.webdriver.Chrome objects, you can specify such information in the *darc.proxy.LINK_MAP* mapping dictionarty.

darc.proxy.**LINK_MAP: DefaultDict[str, Tuple[types.FunctionType, types.FunctionType]]**

```
LINK_MAP = collections.defaultdict(
    lambda: (darc.requests.null_session, darc.selenium.null_driver),
    dict(
        tor=(darc.requests.tor_session, darc.selenium.tor_driver),
        i2p=(darc.requests.i2p_session, darc.selenium.i2p_driver),
    )
)
```

The mapping dictionary for proxy type to its corresponding requests.Session factory function and selenium.webdriver.Chrome factory function.

The fallback value is the no proxy requests.Session object (*null_session()*) and selenium.webdriver.Chrome object (*null_driver()*).

See also:

- *darc.requests* – requests.Session factory functions
- *darc.selenium* – selenium.webdriver.Chrome factory functions

1.11 Sites Customisation

As websites may have authentication requirements, etc., over its content, the darc.sites module provides sites customisation hooks to both requests and selenium crawling processes.

1.11.1 Default Hooks

The *darc.sites.default* module is the fallback for sites customisation.

darc.sites.default.**crawler**(*session*, *link*)
Default crawler hook.

Parameters

- **session** (requests.Session) – Session object with proxy settings.
- **link** (darc.link.Link) – Link object to be crawled.

Returns The final response object with crawled data.

Return type requests.Response

See also:

- *darc.crawl.crawler()*

darc.sites.default.**loader**(*driver*, *link*)
Default loader hook.

When loading, if *SE_WAIT* is a valid time lapse, the function will sleep for such time to wait for the page to finish loading contents.

Parameters

- **driver** (selenium.webdriver.Chrome) – Web driver object with proxy settings.
- **link** (darc.link.Link) – Link object to be loaded.

Returns The web driver object with loaded data.

Return type selenium.webdriver.Chrome

Note: Internally, selenium will wait for the browser to finish loading the pages before return (i.e. the web API event DOMContentLoaded). However, some extra scripts may take more time running after the event.

See also:

- *darc.crawl.loader()*
- *darc.const.SE_WAIT*

To customise behaviours over requests, you sites customisation module should have a crawler() function, e.g. *crawler()*.

The function takes the requests.Session object with proxy settings and a *Link* object representing the link to be crawled, then returns a requests.Response object containing the final data of the crawling process.

darc.sites.**crawler_hook**(*link*, *session*)
Customisation as to requests sessions.

Parameters

- **link** (darc.link.Link) – Link object to be crawled.
- **session** (requests.Session) – Session object with proxy settings.

Returns The final response object with crawled data.

Return type requests.Response

See also:

- darc.sites.SITE_MAP
- *darc.sites._get_spec()*
- *darc.crawl.crawler()*

To customise behaviours over selenium, you sites customisation module should have a loader() function, e.g. *loader()*.

The function takes the selenium.webdriver.Chrome object with proxy settings and a *Link* object representing the link to be loaded, then returns the selenium.webdriver.Chrome object containing the final data of the loading process.

darc.sites.**loader_hook**(*link*, *driver*)
Customisation as to selenium drivers.

Parameters

- **link** (darc.link.Link) – Link object to be loaded.
- **driver** (selenium.webdriver.Chrome) – Web driver object with proxy settings.

Returns The web driver object with loaded data.

Return type selenium.webdriver.Chrome

See also:

- darc.sites.SITE_MAP
- *darc.sites._get_spec()*
- *darc.crawl.loader()*

To tell the darc project which sites customisation module it should use for a certain hostname, you can register such module to the *SITEMAP* mapping dictionary.

darc.sites.**SITEMAP: DefaultDict[str, str]**

```
SITEMAP = collections.defaultdict(lambda: 'default', {
    # 'www.sample.com': 'sample',  # darc.sites.sample
})
```

The mapping dictionary for hostname to sites customisation modules.

The fallback value is default, c.f. *darc.sites.default*.

darc.sites.**_get_spec**(*link*)
Load spec if any.

If the sites customisation failed to import, it will fallback to the default hooks, *default*.

Parameters **link** (darc.link.Link) – Link object to fetch sites customisation module.

Returns The sites customisation module.

Return type types.ModuleType

Warns **SiteNotFoundWarning** – If the sites customisation failed to import.

See also:

- *darc.sites.SITEMAP*

1.12 Module Constants

1.12.1 Auxiliary Function

darc.const.**getpid**()
Get process ID.

The process ID will be saved under the *PATH_DB* folder, in a file named darc.pid. If no such file exists, -1 will be returned.

Returns The process ID.

Return type int

See also:

- *darc.const.PATH_ID*

1.12.2 General Configurations

darc.const.**REBOOT: bool**
If exit the program after first round, i.e. crawled all links from the `requests` link database and loaded all links from the `selenium` link database.

This can be useful especially when the capacity is limited and you wish to save some space before continuing next round. See *Docker integration* for more information.

Default `False`

Environ *DARC_REBOOT*

darc.const.**DEBUG: bool**
If run the program in debugging mode.

Default `False`

Environ *DARC_DEBUG*

darc.const.**VERBOSE: bool**
If run the program in verbose mode. If *DEBUG* is `True`, then the verbose mode will be always enabled.

Default `False`

Environ *DARC_VERBOSE*

darc.const.**FORCE: bool**
If ignore `robots.txt` rules when crawling (c.f. *crawler()*).

Default `False`

Environ *DARC_FORCE*

darc.const.**CHECK: bool**
If check proxy and hostname before crawling (when calling *extract_links()*, *read_sitemap()* and *read_hosts()*).

If *CHECK_NG* is `True`, then this environment variable will be always set as `True`.

Default `False`

Environ *DARC_CHECK*

darc.const.**CHECK_NG: bool**
If check content type through `HEAD` requests before crawling (when calling *extract_links()*, *read_sitemap()* and *read_hosts()*).

Default `False`

Environ *DARC_CHECK_CONTENT_TYPE*

darc.const.**ROOT: str**
The root folder of the project.

darc.const.**CWD = '.'**
The current working direcory.

darc.const.**DARC_CPU: int**
Number of concurrent processes. If not provided, then the number of system CPUs will be used.

Default `None`

Environ *DARC_CPU*

darc.const.**FLAG_MP: bool**
If enable *multiprocessing* support.

Default `True`

Environ *DARC_MULTIPROCESSING*

darc.const.**FLAG_TH: bool**
If enable *multithreading* support.

Default `False`

Environ *DARC_MULTITHREADING*

Note: *FLAG_MP* and *FLAG_TH* can **NOT** be toggled at the same time.

---

darc.const.**DARC_USER: str**
Non-root user for proxies.

Default current login user (c.f. `getpass.getuser()`)

Environ *DARC_USER*

1.12.3 Data Storage

darc.const.**PATH_DB: str**
Path to data storage.

Default `data`

Environ *PATH_DATA*

See also:

See *darc.save* for more information about source saving.

darc.const.**PATH_MISC = '{PATH_DB}/misc/'**
Path to miscellaneous data storage, i.e. `misc` folder under the root of data storage.

See also:

- *darc.const.PATH_DB*

darc.const.**PATH_LN = '{PATH_DB}/link.csv'**
Path to the link CSV file, `link.csv`.

See also:

- *darc.const.PATH_DB*
- *darc.save.save_link*

darc.const.**PATH_QR = '{PATH_DB}/_queue_requests.txt'**
Path to the `requests` database, `_queue_requests.txt`.

See also:

- *darc.const.PATH_DB*
- *darc.db.load_requests()*
- *darc.db.save_requests()*

darc.const.**PATH_QS = '{PATH_DB}/_queue_selenium.txt'**
Path to the `selenium` database, `_queue_selenium.txt`.

See also:

- *darc.const.PATH_DB*
- *darc.db.load_selenium()*
- *darc.db.save_selenium()*

darc.const.**PATH_ID = '{PATH_DB}/darc.pid'**
Path to the process ID file, `darc.pid`.

See also:

- *darc.const.PATH_DB*
- *darc.const.getpid()*

1.12.4 Web Crawlers

darc.const.**SAVE: bool**
If save processed link back to database.

Note: If *SAVE* is `True`, then *SAVE_REQUESTS* and *SAVE_SELENIUM* will be forced to be `True`.

Default `False`

Environ *DARC_SAVE*

See also:

See *darc.db* for more information about link database.

darc.const.**SAVE_REQUESTS: bool**
If save *crawler()* crawled link back to `requests` database.

Default `False`

Environ *DARC_SAVE_REQUESTS*

See also:

See *darc.db* for more information about link database.

darc.const.**SAVE_SELENIUM: bool**
If save *loader()* crawled link back to `selenium` database.

Default `False`

Environ *DARC_SAVE_SELENIUM*

See also:

See *darc.db* for more information about link database.

darc.const.**TIME_CACHE: float**
Time delta for caches in seconds.

The `darc` project supports *caching* for fetched files. *TIME_CACHE* will specify for how log the fetched files will be cached and **NOT** fetched again.

Note: If *TIME_CACHE* is `None` then caching will be marked as *forever*.

Default `60`

Environ *TIME_CACHE*

darc.const.**SE_WAIT: float**
Time to wait for `selenium` to finish loading pages.

Note: Internally, `selenium` will wait for the browser to finish loading the pages before return (i.e. the web API event `DOMContentLoaded`). However, some extra scripts may take more time running after the event.

Default `60`

Environ *SE_WAIT*

darc.const.**SE_EMPTY = '<html><head></head><body></body></html>'**
The empty page from `selenium`.

See also:

- *darc.crawl.loader()*

1.12.5 White / Black Lists

darc.const.**LINK_WHITE_LIST: List[re.Pattern]**
White list of hostnames should be crawled.

Default `[]`

Environ *LINK_WHITE_LIST*

Note: Regular expressions are supported.

darc.const.**LINK_BLACK_LIST: List[re.Pattern]**
Black list of hostnames should be crawled.

Default `[]`

Environ *LINK_BLACK_LIST*

Note: Regular expressions are supported.

darc.const.**LINK_FALLBACK: bool**
Fallback value for *match_host()*.

Default `False`

Environ *LINK_FALLBACK*

darc.const.**MIME_WHITE_LIST: List[re.Pattern]**
White list of content types should be crawled.

Default `[]`

Environ *MIME_WHITE_LIST*

Note: Regular expressions are supported.

darc.const.**MIME_BLACK_LIST: List[re.Pattern]**
Black list of content types should be crawled.

Default `[]`

Environ *MIME_BLACK_LIST*

Note: Regular expressions are supported.

darc.const.**MIME_FALLBACK: bool**
Fallback value for *match_mime()*.

Default `False`

Environ *MIME_FALLBACK*

darc.const.**PROXY_WHITE_LIST: List[str]**
White list of proxy types should be crawled.

Default `[]`

Environ *PROXY_WHITE_LIST*

Note: The proxy types are **case insensitive**.

darc.const.**PROXY_BLACK_LIST: List[str]**
Black list of proxy types should be crawled.

Default `[]`

Environ *PROXY_BLACK_LIST*

Note: The proxy types are **case insensitive**.

darc.const.**PROXY_FALLBACK: bool**
Fallback value for *match_proxy()*.

Default `False`

Environ *PROXY_FALLBACK*

1.13 Custom Exceptions

The *render_error()* function can be used to render multi-line error messages with stem.util.term colours.

The darc project provides following custom exceptions:

- *UnsupportedLink*
- *UnsupportedPlatform*
- *UnsupportedProxy*

The darc project provides following custom exceptions:

- *TorBootstrapFailed*
- *I2PBootstrapFailed*
- *ZeroNetBootstrapFailed*
- *FreenetBootstrapFailed*
- *APIRequestFailed*
- *SiteNotFoundWarning*

exception darc.error.**APIRequestFailed**
Bases: Warning

API submit failed.

exception darc.error.**FreenetBootstrapFailed**
Bases: Warning

Freenet bootstrap process failed.

exception darc.error.**I2PBootstrapFailed**
Bases: Warning

I2P bootstrap process failed.

exception darc.error.**SiteNotFoundWarning**
Bases: ImportWarning

Site customisation not found.

exception darc.error.**TorBootstrapFailed**
Bases: Warning

Tor bootstrap process failed.

exception darc.error.**UnsupportedLink**
Bases: Exception

The link is not supported.

exception darc.error.**UnsupportedPlatform**
Bases: Exception

The platform is not supported.

exception darc.error.**UnsupportedProxy**
Bases: Exception

The proxy is not supported.

exception darc.error.**ZeroNetBootstrapFailed**
Bases: Warning

ZeroNet bootstrap process failed.

darc.error.**render_error**(*message*, *colour*)
Render error message.

The function wraps the stem.util.term.format() function to provide multi-line formatting support.

Parameters

- **message** (*str*) – Multi-line message to be rendered with colour.
- **colour** (*stem.util.term.Color*) – Front colour of text, c.f. stem.util.term.Color.

Returns The rendered error message.

Return type str

As the websites can be sometimes irritating for their anti-robots verification, login requirements, etc., the darc project also privides hooks to customise crawling behaviours around both requests and selenium.

See also:

Such customisation, as called in the darc project, site hooks, is site specific, user can set up your own hooks unto a certain site, c.f. darc.sites for more information.

Still, since the network is a world full of mysteries and miracles, the speed of crawling will much depend on the response speed of the target website. To boost up, as well as meet the system capacity, the darc project introduced multiprocessing, multithreading and the fallback slowest single-threaded solutions when crawling.

Note: When rendering the target website using selenium powered by the renown Google Chrome, it will require much memory to run. Thus, the three solutions mentioned above would only toggle the behaviour around the use of selenium.

To keep the darc project as it is a swiss army knife, only the main entrypoint function *darc.process.process()* is exported in global namespace (and renamed to *darc.darc()*), see below:

darc.**darc**()
Main process.

The function will register *_signal_handler()* for SIGTERM, and start the main process of the darc darkweb crawlers.

The general process can be described as following:

0. *process()*: obtain URLs from the requests link database (c.f. *load_requests()*), and feed such URLs to *crawler()* with *multiprocessing* support.
1. *crawler()*: parse the URL using *parse_link()*, and check if need to crawl the URL (c.f. *PROXY_WHITE_LIST*, *PROXY_BLACK_LIST* , *LINK_WHITE_LIST* and *LINK_BLACK_LIST*); if true, then crawl the URL with requests.

   If the URL is from a brand new host, darc will first try to fetch and save robots.txt and sitemaps of the host (c.f. *save_robots()* and *save_sitemap()*), and extract then save the links from sitemaps (c.f. *read_sitemap()*) into link database for future crawling (c.f. *save_requests()*). Also, if

the submission API is provided, *submit_new_host()* will be called and submit the documents just fetched.

If robots.txt presented, and *FORCE* is False, darc will check if allowed to crawl the URL.

Note: The root path (e.g. / in https://www.example.com/) will always be crawled ignoring robots.txt.

At this point, darc will call the customised hook function from darc.sites to crawl and get the final response object. darc will save the session cookies and header information, using *save_headers()*.

Note: If requests.exceptions.InvalidSchema is raised, the link will be saved by save_invalid(). Further processing is dropped.

If the content type of response document is not ignored (c.f. *MIME_WHITE_LIST* and *MIME_BLACK_LIST*), darc will save the document using *save_html()* or *save_file()* accordingly. And if the submission API is provided, *submit_requests()* will be called and submit the document just fetched.

If the response document is HTML (text/html and application/xhtml+xml), *extract_links()* will be called then to extract all possible links from the HTML document and save such links into the database (c.f. *save_requests()*).

And if the response status code is between 400 and 600, the URL will be saved back to the link database (c.f. *save_requests()*). If **NOT**, the URL will be saved into selenium link database to proceed next steps (c.f. *save_selenium()*).

2. *process()*: after the obtained URLs have all been crawled, darc will obtain URLs from the selenium link database (c.f. *load_selenium()*), and feed such URLs to *loader()*.

   Note: If *FLAG_MP* is True, the function will be called with *multiprocessing* support; if *FLAG_TH* if True, the function will be called with *multithreading* support; if none, the function will be called in single-threading.

   ---

3. *loader()*: parse the URL using *parse_link()* and start loading the URL using selenium with Google Chrome.

   At this point, darc will call the customised hook function from darc.sites to load and return the original selenium.webdriver.Chrome object.

   If successful, the rendered source HTML document will be saved using *save_html()*, and a full-page screenshot will be taken and saved.

   If the submission API is provided, *submit_selenium()* will be called and submit the document just loaded.

   Later, *extract_links()* will be called then to extract all possible links from the HTML document and save such links into the requests database (c.f. *save_requests()*).

If in reboot mode, i.e. *REBOOT* is True, the function will exit after first round. If not, it will renew the Tor connections (if bootstrapped), c.f. *renew_tor_session()*, and start another round.

CHAPTER

TWO

WEB BACKEND DEMO

This is a demo of API for communication between the `darc` crawlers (*`darc.submit`*) and web UI.

Assuming the web UI is developed using the `Flask` microframework.

```python
# -*- coding: utf-8 -*-

import flask  # pylint: disable=import-error

# Flask application
app = flask.Flask(__file__)


@app.route('/api/new_host', methods=['POST'])
def new_host():
    """When a new host is discovered, the :mod:`darc` crawler will submit the
    host information. Such includes ``robots.txt`` (if exists) and
    ``sitemap.xml`` (if any).

    Data format::

        {
            // metadata of URL
            "[metadata]": {
                // original URL - <scheme>://<netloc>/<path>;<params>?<query>#
↪<fragment>
                "url": ...,
                // proxy type - null / tor / i2p / zeronet / freenet
                "proxy": ...,
                // hostname / netloc, c.f. ``urllib.parse.urlparse``
                "host": ...,
                // base folder, relative path (to data root path ``PATH_DATA``) in␣
↪containter - <proxy>/<scheme>/<host>
                "base": ...,
                // sha256 of URL as name for saved files (timestamp is in ISO format)
                //   JSON log as this one - <base>/<name>_<timestamp>.json
                //   HTML from requests - <base>/<name>_<timestamp>_raw.html
                //   HTML from selenium - <base>/<name>_<timestamp>.html
                //   generic data files - <base>/<name>_<timestamp>.dat
                "name": ...
            },
            // requested timestamp in ISO format as in name of saved file
            "Timestamp": ...,
            // original URL
            "URL": ...,
```

(continues on next page)

(continued from previous page)

```
            // robots.txt from the host (if not exists, then ``null``)
            "Robots": {
                // path of the file, relative path (to data root path ``PATH_DATA``)␣
↪in container
                //   - <proxy>/<scheme>/<host>/robots.txt
                "path": ...,
                // content of the file (**base64** encoded)
                "data": ...,
            },
            // sitemaps from the host (if none, then ``null``)
            "Sitemaps": [
                {
                    // path of the file, relative path (to data root path ``PATH_
↪DATA``) in container
                    //   - <proxy>/<scheme>/<host>/sitemap_<name>.txt
                    "path": ...,
                    // content of the file (**base64** encoded)
                    "data": ...,
                },
                ...
            ],
            // hosts.txt from the host (if proxy type is ``i2p``; if not exists, then␣
↪``null``)
            "Hosts": {
                // path of the file, relative path (to data root path ``PATH_DATA``)␣
↪in container
                //   - <proxy>/<scheme>/<host>/hosts.txt
                "path": ...,
                // content of the file (**base64** encoded)
                "data": ...,
            }
        }

    """
    # JSON data from the request
    data = flask.request.json  # pylint: disable=unused-variable

    # do whatever processing needed
    ...


@app.route('/api/requests', methods=['POST'])
def from_requests():
    """When crawling, we'll first fetch the URl using ``requests``, to check
    its availability and to save its HTTP headers information. Such information
    will be submitted to the web UI.

    Data format::

        {
            // metadata of URL
            "[metadata]": {
                // original URL - <scheme>://<netloc>/<path>;<params>?<query>#
↪<fragment>
                "url": ...,
                // proxy type - null / tor / i2p / zeronet / freenet
                "proxy": ...,
```

(continues on next page)

(continued from previous page)

```
                // hostname / netloc, c.f. ``urllib.parse.urlparse``
                "host": ...,
                // base folder, relative path (to data root path ``PATH_DATA``) in␣
↪containter - <proxy>/<scheme>/<host>
                "base": ...,
                // sha256 of URL as name for saved files (timestamp is in ISO format)
                //   JSON log as this one - <base>/<name>_<timestamp>.json
                //   HTML from requests - <base>/<name>_<timestamp>_raw.html
                //   HTML from selenium - <base>/<name>_<timestamp>.html
                //   generic data files - <base>/<name>_<timestamp>.dat
                "name": ...
            },
            // requested timestamp in ISO format as in name of saved file
            "Timestamp": ...,
            // original URL
            "URL": ...,
            // request method
            "Method": "GET",
            // response status code
            "Status-Code": ...,
            // response reason
            "Reason": ...,
            // response cookies (if any)
            "Cookies": {
                ...
            },
            // session cookies (if any)
            "Session": {
                ...
            },
            // request headers (if any)
            "Request": {
                ...
            },
            // response headers (if any)
            "Response": {
                ...
            },
            // requested file (if not exists, then ``null``)
            "Document": {
                // path of the file, relative path (to data root path ``PATH_DATA``)␣
↪in container
                //   - <proxy>/<scheme>/<host>/<name>_<timestamp>_raw.html
                // or if the document is of generic content type, i.e. not HTML
                //   - <proxy>/<scheme>/<host>/<name>_<timestamp>.dat
                "path": ...,
                // content of the file (**base64** encoded)
                "data": ...,
            }
        }

    """
    # JSON data from the request
    data = flask.request.json  # pylint: disable=unused-variable

    # do whatever processing needed
    ...
```

(continues on next page)

(continued from previous page)

```
@app.route('/api/selenium', methods=['POST'])
def from_selenium():
    """After crawling with ``requests``, we'll then render the URl using
    ``selenium`` with Google Chrome and its driver, to provide a fully rendered
    web page. Such information will be submitted to the web UI.

    Note:
        This information is optional, only provided if the content type from
        ``requests`` is HTML, status code < 400, and HTML data not empty.

    Data format::

        {
            // metadata of URL
            "[metadata]": {
                // original URL - <scheme>://<netloc>/<path>;<params>?<query>#
↪<fragment>
                "url": ...,
                // proxy type - null / tor / i2p / zeronet / freenet
                "proxy": ...,
                // hostname / netloc, c.f. ``urllib.parse.urlparse``
                "host": ...,
                // base folder, relative path (to data root path ``PATH_DATA``) in␣
↪containter - <proxy>/<scheme>/<host>
                "base": ...,
                // sha256 of URL as name for saved files (timestamp is in ISO format)
                //   JSON log as this one - <base>/<name>_<timestamp>.json
                //   HTML from requests - <base>/<name>_<timestamp>_raw.html
                //   HTML from selenium - <base>/<name>_<timestamp>.html
                //   generic data files - <base>/<name>_<timestamp>.dat
                "name": ...
            },
            // requested timestamp in ISO format as in name of saved file
            "Timestamp": ...,
            // original URL
            "URL": ...,
            // rendered HTML document (if not exists, then ``null``)
            "Document": {
                // path of the file, relative path (to data root path ``PATH_DATA``)␣
↪in container
                //   - <proxy>/<scheme>/<host>/<name>_<timestamp>.html
                "path": ...,
                // content of the file (**base64** encoded)
                "data": ...,
            },
            // web page screenshot (if not exists, then ``null``)
            "Screenshot": {
                // path of the file, relative path (to data root path ``PATH_DATA``)␣
↪in container
                //   - <proxy>/<scheme>/<host>/<name>_<timestamp>.png
                "path": ...,
                // content of the file (**base64** encoded)
                "data": ...,
            }
        }
```

(continues on next page)

(continued from previous page)

```
    """
    # JSON data from the request
    data = flask.request.json  # pylint: disable=unused-variable

    # do whatever processing needed
    ...


if __name__ == "__main__":
    flask.run()
```

CHAPTER

THREE

DOCKER INTEGRATION

The `darc` project is integrated with Docker and Compose. Though not published to Docker Hub, you can still build by yourself.

The Docker image is based on Ubuntu Bionic (18.04 LTS), setting up all Python dependencies for the `darc` project, installing Google Chrome (version 79.0.3945.36) and corresponding ChromeDriver, as well as installing and configuring Tor, I2P, ZeroNet, FreeNet, NoIP proxies.

Note: NoIP is currently not fully integrated in the `darc` due to misunderstanding in the configuration process. Contributions are welcome.

When building the image, there is an *optional* argument for setting up a *non-root* user, c.f. environment variable `DARC_USER` and module constant *`DARC_USER`*. By default, the username is `darc`.

```
FROM ubuntu:bionic

LABEL Name=darc \
      Version=0.1.6

STOPSIGNAL SIGINT
HEALTHCHECK --interval=1h --timeout=1m \
    CMD wget https://httpbin.org/get -O /dev/null || exit 1

ARG DARC_USER="darc"
ENV LANG="C.UTF-8" \
    LC_ALL="C.UTF-8" \
    PYTHONIOENCODING="UTF-8" \
    DEBIAN_FRONTEND="teletype" \
    DARC_USER="${DARC_USER}"
    # DEBIAN_FRONTEND="noninteractive"

COPY extra/retry.sh /usr/local/bin/retry
COPY extra/install.py /usr/local/bin/pty-install
COPY vendor/jdk-13.0.2_linux-x64_bin.tar.gz /var/cache/oracle-jdk13-installer/

RUN set -x \
 && retry apt-get update \
 && retry apt-get install --yes --no-install-recommends \
        apt-utils \
 && retry apt-get install --yes --no-install-recommends \
        gcc \
        g++ \
        libmagic1 \
```

(continues on next page)

(continued from previous page)

```
        make \
        software-properties-common \
        tar \
        unzip \
        zlib1g-dev \
 && retry add-apt-repository ppa:deadsnakes/ppa --yes \
 && retry add-apt-repository ppa:linuxuprising/java --yes \
 && retry add-apt-repository ppa:i2p-maintainers/i2p --yes
RUN retry apt-get update \
 && retry apt-get install --yes --no-install-recommends \
        python3.8 \
        python3-pip \
        python3-setuptools \
        python3-wheel \
 && ln -sf /usr/bin/python3.8 /usr/local/bin/python3
RUN retry pty-install --stdin '6\n70' apt-get install --yes --no-install-recommends \
        tzdata \
 && retry pty-install --stdin 'yes' apt-get install --yes \
        oracle-java13-installer
RUN retry apt-get install --yes --no-install-recommends \
        sudo \
 && adduser --disabled-password --gecos '' ${DARC_USER} \
 && adduser ${DARC_USER} sudo \
 && echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers

## Tor
RUN retry apt-get install --yes --no-install-recommends tor
COPY extra/torrc.bionic /etc/tor/torrc

## I2P
RUN retry apt-get install --yes --no-install-recommends i2p
COPY extra/i2p.bionic /etc/defaults/i2p

## ZeroNet
COPY vendor/ZeroNet-py3-linux64.tar.gz /tmp
RUN set -x \
 && cd /tmp \
 && tar xvpfz ZeroNet-py3-linux64.tar.gz \
 && mv ZeroNet-linux-dist-linux64 /usr/local/src/zeronet
COPY extra/zeronet.bionic.conf /usr/local/src/zeronet/zeronet.conf

## FreeNet
USER darc
COPY vendor/new_installer_offline.jar /tmp
RUN set -x \
 && cd /tmp \
 && ( pty-install --stdin '/home/darc/freenet\n1' java -jar new_installer_offline.jar␣
↪|| true ) \
 && sudo mv /home/darc/freenet /usr/local/src/freenet
USER root

## NoIP
COPY vendor/noip-duc-linux.tar.gz /tmp
RUN set -x \
 && cd /tmp \
 && tar xvpfz noip-duc-linux.tar.gz \
 && mv noip-2.1.9-1 /usr/local/src/noip \
```

(continues on next page)

(continued from previous page)

```
 && cd /usr/local/src/noip \
 && make
 # && make install

# # set up timezone
# RUN echo 'Asia/Shanghai' > /etc/timezone \
#  && rm -f /etc/localtime \
#  && ln -snf /usr/share/zoneinfo/Asia/Shanghai /etc/localtime \
#  && dpkg-reconfigure -f noninteractive tzdata

COPY vendor/chromedriver_linux64-79.0.3945.36.zip \
     vendor/google-chrome-stable_current_amd64.deb /tmp/
RUN set -x \
 ## ChromeDriver
 && unzip -d /usr/bin /tmp/chromedriver_linux64-79.0.3945.36.zip \
 && which chromedriver \
 ## Google Chrome
 && ( dpkg --install /tmp/google-chrome-stable_current_amd64.deb || true ) \
 && retry apt-get install --fix-broken --yes --no-install-recommends \
 && dpkg --install /tmp/google-chrome-stable_current_amd64.deb \
 && which google-chrome

# Using pip:
COPY requirements.txt /tmp
RUN python3 -m pip install -r /tmp/requirements.txt --no-cache-dir

RUN set -x \
 && rm -rf \
        ## APT repository lists
        /var/lib/apt/lists/* \
        ## Python dependencies
        /tmp/requirements.txt \
        /tmp/pip \
        ## ChromeDriver
        /tmp/chromedriver_linux64-79.0.3945.36.zip \
        ## Google Chrome
        /tmp/google-chrome-stable_current_amd64.deb \
        ## Vendors
        /tmp/new_installer_offline.jar \
        /tmp/noip-duc-linux.tar.gz \
        /tmp/ZeroNet-py3-linux64.tar.gz \
 #&& apt-get remove --auto-remove --yes \
 #       software-properties-common \
 #       unzip \
 && apt-get autoremove -y \
 && apt-get autoclean \
 && apt-get clean

ENTRYPOINT [ "python3", "-m", "darc" ]
#ENTRYPOINT [ "bash", "/app/run.sh" ]
CMD [ "--help" ]

WORKDIR /app
COPY darc/ /app/darc/
COPY LICENSE \
     MANIFEST.in \
     README.rst \
```

(continues on next page)

(continued from previous page)

```
    extra/run.sh \
    setup.cfg \
    setup.py \
    test_darc.py /app/
RUN python3 -m pip install -e .
```

Note:

- `retry` is a shell script for retrying the commands until success

```
#!/usr/bin/env bash

while true; do
    >&2 echo "+ $@"
    $@ && break
    >&2 echo "exit: $?"
done
>&2 echo "exit: 0"
```

- `pty-install` is a Python script simulating user input for APT package installation with `DEBIAN_FRONTEND` set as `Teletype`.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""Install packages requiring interactions."""

import argparse
import os
import subprocess
import sys
import tempfile


def get_parser():
    """Argument parser."""
    parser = argparse.ArgumentParser('install',
                                     description='pseudo-interactive package installer
↪')

    parser.add_argument('-i', '--stdin', help='content for input')
    parser.add_argument('command', nargs=argparse.REMAINDER, help='command to execute
↪')

    return parser


def main():
    """Entrypoint."""
    parser = get_parser()
    args = parser.parse_args()
    text = args.stdin.encode().decode('unicode_escape')

    path = tempfile.mktemp(prefix='install-')
    with open(path, 'w') as file:
        file.write(text)
```

(continues on next page)

(continued from previous page)

```
    with open(path, 'r') as file:
        proc = subprocess.run(args.command, stdin=file)  # pylint: disable=subprocess-
↪run-check

    os.remove(path)
    return proc.returncode


if __name__ == "__main__":
    sys.exit(main())
```

As always, you can also use Docker Compose to manage the `darc` image. Environment variables can be set as described in the configuration section.

```yaml
version: '3'

services:
  darc:
    image: darc
    build:
      context: .
      args:
        # non-root user
        DARC_USER: "darc"
    command: [ "--file", "/app/text/market.txt",
               "--file", "/app/text/i2p.txt",
               "--file", "/app/text/zeronet.txt",
               "--file", "/app/text/freenet.txt" ]
    environment:
      ## [PYTHON] force the stdout and stderr streams to be unbuffered
      PYTHONUNBUFFERED: 1
      # reboot mode
      DARC_REBOOT: 1
      # debug mode
      DARC_DEBUG: 0
      # verbose mode
      DARC_VERBOSE: 1
      # force mode (ignore robots.txt)
      DARC_FORCE: 1
      # check mode (check proxy and hostname before crawling)
      DARC_CHECK: 1
      # check mode (check content type before crawling)
      DARC_CHECK_CONTENT_TYPE: 0
      # save mode
      DARC_SAVE: 0
      # save mode (for requests)
      DAVE_SAVE_REQUESTS: 0
      # save mode (for selenium)
      DAVE_SAVE_SELENIUM: 0
      # processes
      DARC_CPU: 16
      # multiprocessing
      DARC_MULTIPROCESSING: 0
      # multithreading
      DARC_MULTITHREADING: 0
```

(continues on next page)

(continued from previous page)

```
      # time lapse
      DARC_WAIT: 60
      # data storage
      PATH_DATA: "data"
      # Tor proxy & control port
      TOR_PORT: 9050
      TOR_CTRL: 9051
      # Tor management method
      TOR_STEM: 1
      # Tor authentication
      TOR_PASS: "16:B9D36206B5374B3F609045F9609EE670F17047D88FF713EFB9157EA39F"
      # Tor bootstrap retry
      TOR_RETRY: 10
      # Tor bootstrap wait
      TOR_WAIT: 90
      # Tor bootstrap config
      TOR_CFG: "{}"
      # I2P port
      I2P_PORT: 4444
      # I2P bootstrap retry
      I2P_RETRY: 10
      # I2P bootstrap wait
      I2P_WAIT: 90
      # I2P bootstrap config
      I2P_ARGS: ""
      # ZeroNet port
      ZERONET_PORT: 43110
      # ZeroNet bootstrap retry
      ZERONET_RETRY: 10
      # ZeroNet project path
      ZERONET_PATH: "/usr/local/src/zeronet"
      # ZeroNet bootstrap wait
      ZERONET_WAIT: 90
      # ZeroNet bootstrap config
      ZERONET_ARGS: ""
      # Freenet port
      FREENET_PORT: 8888
      # Freenet bootstrap retry
      FREENET_RETRY: 0
      # Freenet project path
      FREENET_PATH: "/usr/local/src/freenet"
      # Freenet bootstrap wait
      FREENET_WAIT: 90
      # Freenet bootstrap config
      FREENET_ARGS: ""
      # time delta for caches in seconds
      TIME_CACHE: 86400  # 1 day
      # time to wait for selenium
      SE_WAIT: 5
      # extract link pattern
      LINK_WHITE_LIST: '[
        ".*?\\.onion",
        ".*?\\.i2p", "127\\.0\\.0\\.1:7657", "localhost:7657", "127\\.0\\.0\\.1:7658",
↪ "localhost:7658",
        "127\\.0\\.0\\.1:43110", "localhost:43110",
        "127\\.0\\.0\\.1:8888", "localhost:8888"
      ]'
```

(continues on next page)

(continued from previous page)

```
      # link black list
      LINK_BLACK_LIST: '[ "(.*\\.)?facebookcorewwwi\\.onion", "(.*\\.)?
↪nytimes3xbfgragh\\.onion" ]'
      # link fallback flag
      LINK_FALLBACK: 1
      # content type white list
      MIME_WHITE_LIST: '[ "text/html", "application/xhtml+xml" ]'
      # content type black list
      MIME_BLACK_LIST: '[ "text/css", "application/javascript", "text/json" ]'
      # content type fallback flag
      MIME_FALLBACK: 0
      # proxy type white list
      PROXY_WHITE_LIST: '[ "tor", "i2p", "freenet", "zeronet" ]'
      # proxy type black list
      PROXY_BLACK_LIST: '[ "null", "data" ]'
      # proxy type fallback flag
      PROXY_FALLBACK: 0
      # API retry times
      API_RETRY: 10
      # API URLs
      #API_NEW_HOST: 'https://example.com/api/new_host'
      #API_REQUESTS: 'https://example.com/api/requests'
      #API_SELENIUM: 'https://example.com/api/selenium'
    restart: "always"
    volumes:
      - ./text:/app/text
      - ./extra:/app/extra
      - /data/darc:/app/data
      # - ./cache:/app/cache
      # ## change timezone
      # - /etc:/etc
      # - /usr/share/zoneinfo:/usr/share/zoneinfo
```

Note: Should you wish to run `darc` in reboot mode, i.e. set *DARC_REBOOT* and/or *REBOOT* as `True`, you may wish to change the entrypoint to

```
bash /app/run.sh
```

where `run.sh` is a shell script wraps around `darc` especially for reboot mode.

```
#!/usr/bin/env bash

set -e

# time lapse
WAIT=${DARC_WAIT=10}

# signal handlers
trap '[ -f ${PATH_DATA}/darc.pid ] && kill -2 $(cat ${PATH_DATA}/darc.pid)' SIGINT␣
↪SIGTERM SIGKILL

# initialise
echo "+ Starting application..."
python3 -m darc $@
sleep ${WAIT}
```

(continues on next page)

(continued from previous page)

```
# mainloop
while true; do
    echo "+ Restarting application..."
    python3 -m darc
    sleep ${WAIT}
done
```

In such scenario, you can customise your `run.sh` to, for instance, archive then upload current data crawled by `darc` to somewhere else and save up some disk space.

Or you may wish to look into the `_queue_requests.txt` and `_queue_selenium.txt` databases (c.f. *darc.db*), and make some minor adjustments to, perhaps, narrow down the crawling targets.

`darc` is designed as a swiss army knife for darkweb crawling. It integrates `requests` to collect HTTP request and response information, such as cookies, header fields, etc. It also bundles `selenium` to provide a fully rendered web page and screenshot of such view.

The general process of `darc` can be described as following:

0. *process()*: obtain URLs from the `requests` link database (c.f. *load_requests()*), and feed such URLs to *crawler()* with *multiprocessing* support.

1. *crawler()*: parse the URL using *parse_link()*, and check if need to crawl the URL (c.f. *PROXY_WHITE_LIST*, *PROXY_BLACK_LIST* , *LINK_WHITE_LIST* and *LINK_BLACK_LIST*); if true, then crawl the URL with `requests`.

   If the URL is from a brand new host, `darc` will first try to fetch and save `robots.txt` and sitemaps of the host (c.f. *save_robots()* and *save_sitemap()*), and extract then save the links from sitemaps (c.f. *read_sitemap()*) into link database for future crawling (c.f. *save_requests()*). Also, if the submission API is provided, *submit_new_host()* will be called and submit the documents just fetched.

   If `robots.txt` presented, and *FORCE* is `False`, `darc` will check if allowed to crawl the URL.

   Note: The root path (e.g. `/` in https://www.example.com/) will always be crawled ignoring `robots.txt`.

   At this point, `darc` will call the customised hook function from `darc.sites` to crawl and get the final response object. `darc` will save the session cookies and header information, using *save_headers()*.

   Note: If `requests.exceptions.InvalidSchema` is raised, the link will be saved by `save_invalid()`. Further processing is dropped.

   If the content type of response document is not ignored (c.f. *MIME_WHITE_LIST* and *MIME_BLACK_LIST*), `darc` will save the document using *save_html()* or *save_file()* accordingly. And if the submission API is provided, *submit_requests()* will be called and submit the document just fetched.

   If the response document is HTML (`text/html` and `application/xhtml+xml`), *extract_links()* will be called then to extract all possible links from the HTML document and save such links into the database (c.f. *save_requests()*).

   And if the response status code is between `400` and `600`, the URL will be saved back to the link database (c.f. *save_requests()*). If **NOT**, the URL will be saved into `selenium` link database to proceed next steps (c.f. *save_selenium()*).

2. *process()*: after the obtained URLs have all been crawled, darc will obtain URLs from the selenium link database (c.f. *load_selenium()*), and feed such URLs to *loader()*.

   Note: If *FLAG_MP* is True, the function will be called with *multiprocessing* support; if *FLAG_TH* if True, the function will be called with *multithreading* support; if none, the function will be called in single-threading.

3. *loader()*: parse the URL using *parse_link()* and start loading the URL using selenium with Google Chrome.

   At this point, darc will call the customised hook function from darc.sites to load and return the original selenium.webdriver.Chrome object.

   If successful, the rendered source HTML document will be saved using *save_html()*, and a full-page screenshot will be taken and saved.

   If the submission API is provided, *submit_selenium()* will be called and submit the document just loaded.

   Later, *extract_links()* will be called then to extract all possible links from the HTML document and save such links into the requests database (c.f. *save_requests()*).

CHAPTER

FOUR

INSTALLATION

Note: `darc` supports Python all versions above and includes **3.6**. Currently, it only supports and is tested on Linux (*Ubuntu 18.04*) and macOS (*Catalina*).

When installing in Python versions below **3.8**, `darc` will use `walrus` to compile itself for backport compatibility.

```
pip install darc
```

Please make sure you have Google Chrome and corresponding version of Chrome Driver installed on your system.

However, the `darc` project is shipped with Docker and Compose support. Please see :*Docker Integration* for more information.

CHAPTER

FIVE

USAGE

The `darc` project provides a simple CLI:

```
usage: darc [-h] [-f FILE] ...

the darkweb knife crawling swiss army knife

positional arguments:
  link                  links to craw

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  read links from file
```

It can also be called through module entrypoint:

```
python -m python-darc ...
```

Note: The link files can contain **comment** lines, which should start with `#`. Empty lines and comment lines will be ignored when loading.

CHAPTER

SIX

CONFIGURATION

Though simple CLI, the `darc` project is more configurable by environment variables.

6.1 General Configurations

DARC_REBOOT

Type `bool` (`int`)

Default `0`

If exit the program after first round, i.e. crawled all links from the `requests` link database and loaded all links from the `selenium` link database.

This can be useful especially when the capacity is limited and you wish to save some space before continuing next round. See *Docker integration* for more information.

DARC_DEBUG

Type `bool` (`int`)

Default `0`

If run the program in debugging mode.

DARC_VERBOSE

Type `bool` (`int`)

Default `0`

If run the program in verbose mode. If `DARC_DEBUG` is `True`, then the verbose mode will be always enabled.

DARC_FORCE

Type `bool` (`int`)

Default `0`

If ignore `robots.txt` rules when crawling (c.f. *`crawler()`*).

DARC_CHECK

Type `bool` (`int`)

Default `0`

If check proxy and hostname before crawling (when calling *`extract_links()`*, *`read_sitemap()`* and *`read_hosts()`*).

If `DARC_CHECK_CONTENT_TYPE` is `True`, then this environment variable will be always set as `True`.

DARC_CHECK_CONTENT_TYPE

Type `bool` (`int`)

Default `0`

If check content type through `HEAD` requests before crawling (when calling *`extract_links()`*, *`read_sitemap()`* and *`read_hosts()`*).

DARC_CPU

Type `int`

Default `None`

Number of concurrent processes. If not provided, then the number of system CPUs will be used.

DARC_MULTIPROCESSING

Type `bool` (`int`)

Default `1`

If enable *multiprocessing* support.

DARC_MULTITHREADING

Type `bool` (`int`)

Default `0`

If enable *multithreading* support.

Note: `DARC_MULTIPROCESSING` and `DARC_MULTITHREADING` can **NOT** be toggled at the same time.

DARC_USER

Type `str`

Default current login user (c.f. `getpass.getuser()`)

Non-root user for proxies.

6.2 Data Storage

PATH_DATA

Type `str` (path)

Default `data`

Path to data storage.

See also:

See *`darc.save`* for more information about source saving.

6.3 Web Crawlers

DARC_SAVE

Type bool (int)

Default 0

If save processed link back to database.

Note: If *DARC_SAVE* is True, then *DARC_SAVE_REQUESTS* and *DARC_SAVE_SELENIUM* will be forced to be True.

See also:

See *darc.db* for more information about link database.

DARC_SAVE_REQUESTS

Type bool (int)

Default 0

If save *crawler()* crawled link back to requests database.

See also:

See *darc.db* for more information about link database.

DARC_SAVE_SELENIUM

Type bool (int)

Default 0

If save *loader()* crawled link back to selenium database.

See also:

See *darc.db* for more information about link database.

TIME_CACHE

Type float

Default 60

Time delta for caches in seconds.

The darc project supports *caching* for fetched files. TIME_CACHE will specify for how log the fetched files will be cached and **NOT** fetched again.

Note: If TIME_CACHE is None then caching will be marked as *forever*.

SE_WAIT

Type float

Default 60

Time to wait for selenium to finish loading pages.

Note: Internally, `selenium` will wait for the browser to finish loading the pages before return (i.e. the web API event `DOMContentLoaded`). However, some extra scripts may take more time running after the event.

6.4 White / Black Lists

LINK_WHITE_LIST

Type `List[str]` (JSON)

Default `[]`

White list of hostnames should be crawled.

Note: Regular expressions are supported.

LINK_BLACK_LIST

Type `List[str]` (JSON)

Default `[]`

Black list of hostnames should be crawled.

Note: Regular expressions are supported.

LINK_FALLBACK

Type `bool` (`int`)

Default `0`

Fallback value for *`match_host()`*.

MIME_WHITE_LIST

Type `List[str]` (JSON)

Default `[]`

White list of content types should be crawled.

Note: Regular expressions are supported.

MIME_BLACK_LIST

Type `List[str]` (JSON)

Default `[]`

Black list of content types should be crawled.

Note: Regular expressions are supported.

MIME_FALLBACK

Type bool (int)

Default 0

Fallback value for *match_mime()*.

PROXY_WHITE_LIST

Type List[str] (JSON)

Default []

White list of proxy types should be crawled.

Note: The proxy types are **case insensitive**.

PROXY_BLACK_LIST

Type List[str] (JSON)

Default []

Black list of proxy types should be crawled.

Note: The proxy types are **case insensitive**.

PROXY_FALLBACK

Type bool (int)

Default 0

Fallback value for *match_proxy()*.

Note: If provided, LINK_WHITE_LIST, LINK_BLACK_LIST, MIME_WHITE_LIST, MIME_BLACK_LIST, PROXY_WHITE_LIST and PROXY_BLACK_LIST should all be JSON encoded strings.

6.5 Data Submission

API_RETRY

Type int

Default 3

Retry times for API submission when failure.

API_NEW_HOST

Type str

Default None

API URL for *submit_new_host()*.

API_REQUESTS

Type str

Default None

API URL for *submit_requests()*.

API_SELENIUM

Type str

Default None

API URL for *submit_selenium()*.

Note: If API_NEW_HOST, API_REQUESTS and API_SELENIUM is None, the corresponding submit function will save the JSON data in the path specified by PATH_DATA.

6.6 Tor Proxy Configuration

TOR_PORT

Type int

Default 9050

Port for Tor proxy connection.

TOR_CTRL

Type int

Default 9051

Port for Tor controller connection.

TOR_STEM

Type bool (int)

Default 1

If manage the Tor proxy through stem.

TOR_PASS

Type str

Default None

Tor controller authentication token.

Note: If not provided, it will be requested at runtime.

TOR_RETRY

Type int

Default 3

Retry times for Tor bootstrap when failure.

TOR_WAIT

Type float

Default 90

Time after which the attempt to start Tor is aborted.

Note: If not provided, there will be **NO** timeouts.

TOR_CFG

Type `Dict[str, Any]` (JSON)

Default `{}`

Tor bootstrap configuration for `stem.process.launch_tor_with_config()`.

Note: If provided, it should be a JSON encoded string.

6.7 I2P Proxy Configuration

I2P_PORT

Type `int`

Default `4444`

Port for I2P proxy connection.

I2P_RETRY

Type `int`

Default `3`

Retry times for I2P bootstrap when failure.

I2P_WAIT

Type `float`

Default `90`

Time after which the attempt to start I2P is aborted.

Note: If not provided, there will be **NO** timeouts.

I2P_ARGS

Type `str` (Shell)

Default `''`

I2P bootstrap arguments for `i2prouter start`.

If provided, it should be parsed as command line arguments (c.f. `shlex.split`).

Note: The command will be run as `DARC_USER`, if current user (c.f. `getpass.getuser()`) is *root*.

6.8 ZeroNet Proxy Configuration

ZERONET_PORT

Type `int`

Default `4444`

Port for ZeroNet proxy connection.

ZERONET_RETRY

Type `int`

Default `3`

Retry times for ZeroNet bootstrap when failure.

ZERONET_WAIT

Type `float`

Default `90`

Time after which the attempt to start ZeroNet is aborted.

Note: If not provided, there will be **NO** timeouts.

ZERONET_PATH

Type `str` (path)

Default `/usr/local/src/zeronet`

Path to the ZeroNet project.

ZERONET_ARGS

Type `str` (Shell)

Default `''`

ZeroNet bootstrap arguments for `ZeroNet.sh main`.

Note: If provided, it should be parsed as command line arguments (c.f. `shlex.split`).

6.9 Freenet Proxy Configuration

FREENET_PORT

Type `int`

Default `8888`

Port for Freenet proxy connection.

FREENET_RETRY

Type `int`

Default `3`

Retry times for Freenet bootstrap when failure.

FREENET_WAIT

Type `float`

Default `90`

Time after which the attempt to start Freenet is aborted.

Note: If not provided, there will be **NO** timeouts.

FREENET_PATH

Type `str` (path)

Default `/usr/local/src/freenet`

Path to the Freenet project.

FREENET_ARGS

Type `str` (Shell)

Default `''`

Freenet bootstrap arguments for `run.sh start`.

If provided, it should be parsed as command line arguments (c.f. `shlex.split`).

Note: The command will be run as `DARC_USER`, if current user (c.f. `getpass.getuser()`) is *root*.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

Q

R

S

T

U

Z