Web Scraper Documentation

0: Format and Wanings

In this document, code snippets, classes names, variables and such will come sorrounded by triple graves and/or in a distinct background.

Personally, I like to end functions arguments with "_arg", so that they are easily distinguished from function variables.

Even though the ```brotli``` library is not used once in the script, its required to be installed, as the ```requests``` library uses it under the hood

1: Overall Structure of the Code

The process of extracting information from news websites can be split into 3 main parts:

1. Crawl through the main page, obtaining the relevant links to particular news
2. Process the html of each link to obtain the relevant text in a conveninent format
3. Analize said text with various methods

2: Class -> Disaster (and subclasses)

```Disaster```

WIP

Abstract parent class to all the disaster subclasses (Flood, Volcano, Earthquake, etc…). All the children classes must implement the save_to_database method.

```
class NoneDisaster(Disaster):
    """Debugging tool that holds an unprocessed disaster subclass"""
```

Pretty self explanatory. self.unprocessed_contents contains the data that, in the normal course of the program, would have been passed to an NLP processor or similar.

Its save_to_database method just prints the unprocessed contents to the standard output.

2: Class -> Website

```Website```

Works as a data structure that holds information about where the different targets of the scraping are locates as well as other data, like link blacklists and such.

Also contains the methods required to scrape the website

```
def __init__(self, web_name_arg: str, main_page_link_arg: str,
             news_tag_type_arg: str, news_tag_attr_arg: dict,
             new_link_tag_type_arg: str, new_link_tag_attr_arg: dict,
             title_tag_type_arg: str, title_tag_attr_arg: dict,
             body_tag_type_arg: str, body_tag_attr_arg: dict,
             next_page_tag_attr_arg: dict = None,
             news_links_blacklist_arg: [str] = None,
             news_links_whitelist_arg: [str] = None,
             base_next_page_link_arg: str = "",
             base_news_link_arg:str = "",
             scraping_method_arg: str = "generic",
             main_needs_selenium_arg: bool = False,
             news_needs_selenium_arg: bool = False,
             encoding_arg: str = "UTF-8"):
```

Most of the parameters are *""self explanatory""* but others aren't:

- web_name_arg: A string containing the name of the website
- main_page_link_arg: A string containing a link to the main page we will scrape news links from
- news_tag_type_arg: A string containing the type of tag that encapsulates the news section we want to scrape links from in the main page
- news_tag_attr_arg: A dictionary of the attributes of the tag specified in the previous variable
- new_link_tag_type_arg: Inside the tag specified in news_tag_type_arg, the type of tag the encapsulates the link to each individual new. Must not be the <a> tag that has the link, but, for example, its parent tag
- new_link_tag_attr_arg: Attributes of the tag specified above
- title_tag_type_arg: Inside the specific new, type of tag that contains the title
- title_tag_attr_arg: Attributes of the tag specified above
- body_tag_type_arg: Inside the specific new, type of tag that contains the body
- body_tag_atrr_arg: Attributes of the tag specified above
- next_page_attr_arg: A dictionary of the attributes of the <a> tag containing the link to the next page of links
- new_links_blacklist_arg: A list of strings in the form of a re regex that matches which links to ignore from the div we specified in the previous variables
- new_links_whitelist_arg: Similar to the previous parameter
- base_news_link_arg: Root link for the news. See ```parse_link``` for more details
- base_next_page_link_arg: Root link for the next page. See ```parse_link``` for more details
- scraping_method_arg: Methods of scraping the individual news. See dispatch_links for more information. Defaults to "generic"
- main_needs_selenium_arg: a bool stating if the main page has relevant java scripts that need to be loaded. Defaults to False
- news_needs_selenium_arg: a bool stating if the individual news have relevant java scripts that need to be loaded. Defaults to False
- encoding_arg: Type of encoding in the web. Defaults to UTF-8

Additionally, each instance of Website has a self.link_pipeline attribute. It's a Queue instance that contains dictionaries of links of individual news and their status:

Eg: {"link": "foo1.com", "status": "Not_Yet_Executed"}

{{"link": "foo2.com", "status": "Connection_Error"}

```
filter_link(self, link_arg: str) -> bool
```

When collecting links from the main pages of the web, some links aren't relevant, such as sources and redirect links, or maybe some links aren't disaster-related. We just care about disaster news links, so the filter methods applies the link_whitelists and link_blacklists defined in the instance of to the link passed as an argument. A return value of True means we accept the link.

Note that we only accept a link if it **simultaneously** matches at least one whitelists and matches no blacklists. If whitelist is None, all links will match the whitelist.

```
parse_link(self, scraped_tag_arg, parse_next_page_link_arg: bool = 0)
-> str:
```

Some of the links obtained from scraping can be just relative urls instead of complete urls. This method checks the links are not None values and completes the links by adding the missing part according to if the incomplete url is a link to a new or to another page.

- parse_next_page_link_arg: A Boolean value indicating where to parse the url as:
  - 0 -> as a normal news link
  - 1 -> as a next page link

Note that the padding added to the link in the scraped tag is determined by the xxxxx_base_link attributes in the instance this method is applied to.

If no padding is needed, (by default), xxxxx_base_link must be an empty string.

```
def get_soup_from_link(self, link_arg: str, use_selenium_arg: bool =
False) -> BeautifulSoup:
```

Gets the response from the link specified by link_arg and converts it into a BeautifulSoup instance.

- Link_arg: is the url we want to extract the html from
- Use_selenium_arg: bool that specifies if we should use selenium (in case the website contains relevant java scripts). Defaults to not using selenium (cause its faster)

```
get_links(self, overwrite_link_arg=None, max_links=100) -> None:
```

Recursive function that crawls through the website extracting links to specific news. It returns None and fills the self.link_pipeline with all the relevant links found, setting their status to “Not_Yet_Executed”.

It first crawls through the first page of news, then goes for the second, third, etc… until the maximum number of links are reached. IT WORKS BUT ITS MISSING A CORE PART; SEE TO DO IN SOURCE CODE FOR MORE INFORMATION (or to do list below).

Arguments:

- overwrite_link_arg: argument used for recursion. By default, the function will crawl through the link to the main page included in website_arg, but a link to the next page is passed recursively
- max_links: maximum of links to be scraped. Note that the implementation is kind of lazy, so it doesn’t stop collecting links until the end of the page. Eg: we set the argument to 100. The first 2 pages contain 90 links in total, so the function will go on a call itself another time to crawl the third page. The third page contains 30 links, so the function will fetch them and stop the recursion, making it so that the total of link obtained is 120, 20 more than the expected 100.

To Do list:

TODO: this code may cause the same link to be analyzed twice if the script is run twice in a short time or if max_links is too high. Implement a "last link" check so that if the collection of links reaches the first link of the last search, it stops

```
def dispatch_links(self, extracting_method_arg: str = "generic",
                   n_of_threads: int = 10,
                   status_filter_arg: str = None) -> None:
```

Processes the pipeline. Generates n consumer threads that processes the links in self.link_pipeline. A filter can be set to select which links to dispatch.

Arguments:

- extracting_method_arg: Method the thread should use. Rn the only one implemented is “generic”, corresponding to the generic_new_scraping method. Some websites, like gdacs has such a well defined structure that particular methods are a good idea. This parameter defaults to “generic”. Note: The method should have (self, link: str) as arguments and return a Dissaster subclass
- n_of_threads: pretty self explanatory. Number of consumer threads to consume links in the pipeline. Note that threads work slower when self.news_need_sel is true, as all threads share a single selenium driver.
- status_filter_arg: The method only processes links whose status is the one specified by status_filter_arg. If None, the method ignores the status and processes the full pipeline. Some values this parameter can take, for example, are: “Not_Yet_Processed”, "Connection_Error", or “Parsing_Error”

If a particular link, for any reason, fails, it will remain in the pipeline with a new status value.

```
def thread_function(self, apply_method_arg,
                    failed_links_queue_arg: Queue,
                    filter_arg) -> None:
```

One consumer thread, started by dispatch_links. Its workflow is the following:

1. Grab a link from the pipeline
2. Check the status of the link to see if it matches the filter
3. Apply the method passed as an argument.
4. Save the data in the instance of class returned by the method to the database
5. Repeat

Arguments:

- apply_method_arg: method to be applied to extract information from the link. Note: The method should have (self, link: str) as arguments and return a Dissaster subclass
- failed_link_queue_arg: queue shared between all threads, in which they put the links that fail with an updated status
- filter_arg: filters status. Check dispatch_links for more information

Note: The thread will never stop until **all** the links in the pipeline have been processed.

```
def generic_new_scraping(self, link_to_new_arg: str) -> Disaster:
```

Designed for unstructured news portals, NLP needed. NOTE: NLP NOT YET IMPLEMENTED, RETURNS ONLY A NoneDisaster INSTANCE WITH THE TITLE, BODY AND LINK TO THE NEW

Takes a link, obtains its html response, and extracts the title and body from it. Passes the link, the title and body to a NLP analyzer function that will, in turn, return an appropriate Disaster subclass instance

EVERYTHING AFTER THIS IS TO BE REMOVED (I think?)

```
```from Disaster_Subclasses import Disaster


class Website:
    """Class To wrap the information needed to crawl through a web"""
    def __init__(self, main_page_link_arg: str, instructions_arg:
[str]):
        self.main_link = main_page_link_arg
        self.next_page_location = instructions_arg


class NewsWrapper:
    """Each instance holds a new, basically just for convenience"""
    def __init__(self, headline_arg: str, body_arg: str, link_arg:
str):
        self.headline = headline_arg
        self.link = link_arg
        self.body = body_arg
        # etc...
```
```

```
    def process_new(self) -> Disaster:
        """Uses nlp to extract relevant data from title and body,
etc"""
        # ideas: location and disaster type can usually be extracted
from headline
        # notes: return none if error, and keep record of "failed"
calls
        raise NotImplemented


class WebCrawler:
    """Class that contains functions to crawl and collect data from
webs"""

    def get_links(self) -> [str]:
        """Crawls through the main page and get links of relevant
instances. Takes as an input an instance of Website"""
        # ideas: recursively go through pages until reaching the last
viewed new
        # notes: set a maximum of links: n. ¿¿¿Perhaps generator
function but then recursion is a no-no???
        raise NotImplemented

    def extract_raw_text(self) -> NewsWrapper:
        """Visits link provided. gets the html, and returns a
string"""
        # ideas: get format of the web in question as argument ig?.
Return newspaper instance
        # notes:
        raise NotImplemented

    def extract_data_gdacs(self) -> Disaster:
        """gdacs has such a well-defined structure that should be
treated as an extra case, in which no NLP is needed"""
        # ideas: extract from the table each disaster has
        # notes:
        raise NotImplemented

    def crawl_website(self) -> None:
        """Methods wrapper. given a website, apply the needed methods
to crawl and scrape information."""
        # ideas: maybe a new class is needed. a class used as argument
in this method, containing:
        #           -> link to the main page
        #           -> ¿is a special case? (eg: gdacs)
        #           -> divs in which the links, and the next page
button are contained
        #        perhaps threading would be interesting. caution with
threads interacting with db in bad ways
        #
        # notes: general flow of the function: get_links-
>extract_raw_text->process_new->store_in_database
        #        be cautious with None value. Maybe monoid?
        raise NotImplemented
```