Web Crawler

I built a simple asynchronous web crawler using Python 3.8.

I tried to write everything the way I would write production code, however of course I had to make some assumptions on coding guidelines and whatnot: for example, different companies could have different conventions on how docstrings should be written, whether and when variables should be protected, (via double leading underscores or a single leading underscore), what packaging tool to use, where to put licenses, if and when type hints are mandatory and so on.

Project Structure and packaging tool

The project contains one root folder that I called challenge and at the same level there's a pipfile which contains the 3 external libraries that I used: aiohttp (for making asynchronous HTTP requests), beautifulsoup4 (for HTML parsing) and mockito. The latter is only used for tests, and I chose to use it over unittest.mock simply because it's very similar to Java's mockito library, which I use daily in my current full time job and for me it feels easier to use than unittest.mock.

Pipenv can be used to create a virtual environment based on the pipfile.lock. Python 3.8 or higher is required.

Inside the challenge directory there are 3 subdirectories and one py file:

- crawler: where crawler.py is located, which contains the most important class of this software: the Crawler class
- utils: where url_utils.py is contained, which contains a few utility functions and two classes used by Crawler. A helper printer function, results_printer.py, is also included
- tests: where tests are located
- uml: where a simple uml class diagram is located
- example_main.py, which can be run to check run the software

I haven't created any License document or any license folder for third party libraries, as I supposed the software would be closed-source, so all licenses would actually be documented in the software's documentation.

Initial Development Approach

I studied HTML pages of a few websites and figured out what the software should support.
First of all, I found that a lot of sites do not have a sitemap.xml and when a website has one, it can be incomplete. This went against my will to support as many websites as possible, so crawling just sitemap.xml pages was not an option.
I noticed that most links to other HTML and user-visitable pages in a HTML page are inside "href" attributes of "a" tags, so I figured that I had to concentrate my efforts on parsing HTML pages. However, some useful resources could also be found outside such elements, so I decided to support that as well via regex matching for URLs.
Very few links seemed to be javascript generated content, so I decided not to support that at the moment.

Moreover, I found out that there could be different kinds of links in a page: absolute links, relative links, relative from root links and "malformed" links. I decided to support and not to support the

following kinds:

To support:

- Normal links: e.g. https://www.mysite.com
- Simple Relative links: e.g. ./relative-link *or* relative-link
- Relative from root links: e.g. /relative-from-root-link
- Slightly malformed links: e.g. https://mysite.com//simple-https-link/

Not to support:

- Links to page fragments: e.g. #fragment, which directs to a section of the page that is being analyzed, which is useless
- Links to contents: e.g. .pdf *or* .jpg files, which are not "crawlable"
- Complex relative links: e.g. ../path *or* ../../path as they are very rare, but complex to handle. However, I wanted to make sure that the software would require few modifications to handle them, in case their support should be added.

Hence, I started writing some simple HTML files to be used as examples of what the software should be able to read from. I also wrote some "dummy" tests to be completed when the crawler would be implemented. The test are set up in a way that they mock the HTTP Response from HTTP Requests to specific URLs in order to return my previously created HTML example pages, in which I know exactly how many links there are and which pages the crawler should crawl in.

As for the crawler itself, I had no doubt about which library to use. I've been using asyncio and aiohttp for a while and I thought they were perfect for the job, as making HTTP requests is a IO-bound task, for which cooperative multitasking is perfect.

As previously mentioned, I decided to ignore javascript generated content, with the possibility to implement it and let it be configurable to be used in the future, possibly by using some other library (e.g. selenium) to actually use a browser in headless mode, however with a significant (in my opinion) performance loss.

A problem that I encountered is the fact that the same URL could be found in different "forms" in different HTML pages (or even in the same page): e.g. an URL found in https://mysite.com could be https://mysite.com/path or simply /path or path or ./path, as they point to the same URL, but their string representation is different. I had to find a way to "normalize" all URLs, so that https://mysite.com/path, path and ./path found in https://mysite.com or /path found in https://mysite.com/another-path would be seen as formally the same URL by the software.

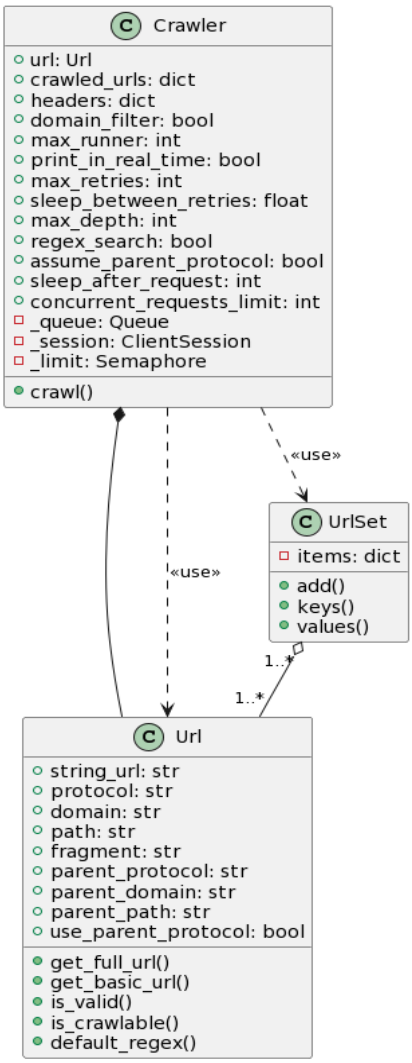
Hence I decided to create a simple class called "Url" which would accept a string representation of the URL and a parent "Url" object, so that it could understand the URL and return a "normalized" URL when its get_basic_url() method is called. I thought this "normalization" would be very useful to avoid making HTTP requests to the same URL more than once and to display the end result of the crawling in a cleaner way. I also created a "UrlSet" class to be able to group Urls in a Set without duplications: basically, their hash would be the return value of url.get_basic_url().

Then I started by creating a dummy crawler that I thought had to be very easy to use: a simple instantiation of a Crawler object (given a root URL and optionally some options) and a simple "crawl()" method to be called to start crawling.

So I went on with this approach, evolving tests and implementation at the same time. In the next

section a summary of the Software Architecture can be seen.

Software Architecture



Crawler:
It is instantiated by at least providing a valid URL as a string. If the other public properties are not provided, then default values are used

Url:
Used to “normalize” URLs. It is instantiated by providing a url as a string, and optionally also the parent Url object (None by default) and the use_parent_protocol.

UrlSet:
Used for creating sets of normalized Urls without using any extra logic in other classes. UrlSet considers two URLs equals when get_basic_url() returns the same string for both of them.

Crawler:

url: the root url for crawling
headers: the headers to use for the HTTP requests. A default one will be used if None
regex_search: if True, the urls found in a page will be found using regex matching on the whole html page. Otherwise, only links present in href attributes of a tags will be considered
domain_filter: if True, only urls with the same domain of the root url will be added to the list of urls present in a page. If False, also urls with different domains will be considered. In any case, no urls with a different domain will be crawled into
max_runners: the max numbers of runners to use for crawling. It must be a positive number, otherwise it is set to 1
print_in_real_time: if True, it will print all urls found in a page in real time.
max_retries: the maximum number of retries to execute when an HTTP request fails. It must be 0

or a positive number, otherwise it is set to 0
sleep_between_retries: the float number of seconds to wait before retrying an HTTP request
max_depth: the max depth allowed for crawling. It can be None or 0 or a positive number, otherwise it is set to 0
sleep_after_request: number of seconds to sleep after each request. It must be 0 or a positive integer number, otherwise it is set to 0
concurrent_requests_limit: the limit of concurrent HTTP requests that the crawler can perform. It's None by default
_queue: the internal asyncio queue
_session: the ClientSession
_limit: the Semaphore

Url:

url: the url
root_url: the root url. It's the URL from which the crawling process started
use_parent_protocol: if True, the protocol from the parent will always be used if the parent domain is the same as the url domain

UrlSet:

_items: the dict of URLs, with the results url.get_basic_url() as keys and the Url objects as values.

Crawler Configuration and Details

I wanted to have the Crawler as configurable as possible, so that a user could decide:

- Which headers to use for the HTTP requests, as some sites could block requests with some headers (e.g. with User-Agent that includes “robot” or “python”, even though the website’s robots.txt does not disallow crawling per-se)
- Whether to show or not to show URLs with different domains in the results, as they could be interesting to see
- Whether to print in real time the crawling results or just get them as a return of the crawl() method, as the former is slower but could be useful
- Whether to retry a failed HTTP request and how many times, as some recoverable errors could happen, and how much time should pass between one try and the next
- How “deep” should the crawler go, as some websites are huge and it could take a very long time to crawl through all the links, but it might be needed to just get the crawling result until a certain depth.
- Whether to also get elements outside href attributes of “a” tags, by using a regex expression through the entire html page
- The maximum number of runners that wait and then process elements in the queue as they come
- Whether to see http and https links as different, or to always see them as the same, as most websites now have http to https redirects
- Whether to sleep or not after each request. Some websites have a limit on how many requests per minute can be done, and sleeping would help in that case.
- Whether to have a limit to concurrent executions, as that would help in case the server limits connections from a single IP

For all of these options, a default options is provided so that the instantiation of a Crawler could be as simple as Crawler(url), with url as a string.

The crawling process is started by calling crawl(): the root Url is then put into the queue and N queue_processor tasks are created. The Url is then retrieved by a queue processor and its HTML page is retrieved by making an HTTP Request and getting the response. The HTML page is then parsed and all URLs inside it are obtained, “normalized” and then put into a queue to be processed, and so on. The cycle ends when all found URLs have been scraped or when the max_depth has been reached.

The code is fully documented.

Test Coverage

As previously mentioned, I wrote dummy tests before actually implementing the crawler. So I simply expanded them as I went on with the actual implementation. I wrote tests for every single “public” function and method of both url_utils.py and crawler.py, which also test multiple edge cases. I usually aim at 80%+ test coverage and cover all edge cases that I can find.
The next step I would take to improve tests is running a test coverage analysis and then write more tests to cover all uncovered lines.

Example Run

You can check out the Example_run.txt file to see the scraping result with root URL Anonymyzed, domain filtering ON, regex_search OFF, concurrent_requests_limit set to 7 and print_in_real_time ON.

A printed line looks like this:

The urls present in anonymized are: https://anoymized1 (1),
anonymized2 (1), anonymized (2),

The number between parenthesis is the number of time that that URL appears in the page. The results also include contents such as PDF or emails, but they could be easily left out by modifying the custom print function code that prints the crawling results.

example_main.py can simply be run to check the same results. Scraping multiple domains at the same time can also be done by instantiating multiple Crawler object with different URLs and using “await asyncio.gather(crawler1.crawl(), crawler2.crawl(), crawlerN.crawl())” instead of “await crawler.crawl()”

Limitations and improvements

- Slightly more complex relative urls such as ../../path are considered invalid. With a bit more work they should work as well
- The sitemap is completely ignored, but in theory it can be “crawled” by enabling regex_search, but it’s inefficient. It could be supported without too much hassle by using a xml parser.
- crawler_test.py could be improved by using a different mock library. It could also be extended and properly optimized.
- Currently, almost all exceptions that are encountered when making an HTTP request are considered as recoverable, which means the HTTP request is retried until the maximum number of retries is reached. One could study which exceptions can actually be recoverable and which cannot, so the software could be sped up.
- Some more edge cases should be covered. For example, xml files could be recognized and parsed differently than html pages, which would also mean directly supporting sitemap scraping.
- Some more testing on real web sites would be useful to find out what the software currently doesn’t support and what can be done to improve it.
- javascript generated content is ignored. A javascript mode that would use a headless browser or some other library that supports javascript could be implemented and configured to be used instead of simple HTTP requests
- Multiple domain scraping could in theory be enabled without having to manually instantiate multiple objects. A helper function for that purpose could be implemented.
- Adding the possibility to use proxies to make HTTP requests, so that the remote server would not disconnect or ban because of too many requests from the same IP
- The print function could be modified to be
- General optimization and clean up