

ACIT4420 Problem Solving with Scripting

Final Project Submission – Web Crawler

Christian Ruben Alexander Jahren

OsloMet – Oslo Metropolitan University

December, 2020

Contents

Introduction.....	1
Overview of Available Solutions	9
Web Scrapers.....	9
Components	11
Design.....	14
Implementation	14
Planning.....	14
Development	14
Analysis	20
Input Parameters	20
Features.....	20
Testing.....	21
Test log	21
Debugger	23
Evaluation.....	25
Figure 1 Terminal Interface part one	2
Figure 2 Terminal Interface part two	3
Figure 3 Terminal Interface part three	3
Figure 4 Terminal Interface part four	4
Figure 5 Terminal Interface part five	4
Figure 6 Terminal Interface part six.....	5
Figure 7 Terminal Interface part seven.....	5
Figure 8 Terminal Interface part eight	6
Figure 9 Terminal Interface part nine	7
Figure 10 Terminal Interface part ten.....	7
Figure 11 Terminal Interface part eleven.....	8
Figure 12 A collection of web crawler, spider and resources in the Python language	10
Figure 13 Scrapy's re function matches content with a user-provided regular expression.....	11
Figure 14 Software structure.....	13
Figure 15 PyCharm Debugger part two.....	23
Figure 16 PyCharm Debugger part one.....	23
Figure 17 PyCharm Debugger part three	24
Figure 18 PyCharm Debugger part four	24
Figure 19 PyCharm Debugger part five	25

Introduction

This document is a report of the final project submission in ACIT4420 Problem Solving with Scripting, the autumn semester of 2020. The project task I've chosen to tackle is the web crawler.

In this section you will be briefly introduced to web crawlers be presented with the basics of the software that has been developed in the project.

A web crawler is a type of software that systematically browses the internet, to retrieve information from websites, often for the purpose of indexing websites for search engines, or web scraping.

The functionality of the web crawler I have developed was specified in project description, and thus I've attempted to limit the scope of the software to the specification, to the best of my ability.

The program is capable of:

- taking input from the user in the terminal
- using that input to download a website and its subpages by identifying links in the html
- analysing the content to identify
 - e-mail addresses
 - phone numbers
 - comments inside the source code
 - special data matched by user-defined regular expressions
 - the most frequently used words on the website
- outputting the identified data in the terminal in a way that's readable for the user

```

Would you like to set up your own parameters? y/n
y

How would you like to run the program?:
1. Crawl the web, download pages and analyze.
2. Just crawl the web and download pages.
3. Just analyze pages that have already been downloaded.
1

Please enter a URL on the form 'address.domain',
like 'google.com'.
Do not worry about protocol or 'www':
posten.no

Please enter the desired depth of crawler recursion as an integer,
where a depth of 0 will result in only the main page being downloaded,
and a depth of 1 will result in all the links discovered in the main page being subject to downloads.
A maximum depth of 3 is recommended, but not enforced:
1

Please enter the maximum/total amount of subpages to download,
where the default value is 100, the minimum value is 10, and the maximum value is 1000
20

Please enter a valid regex, or enter c to continue.
For help with regex syntax, visit https://docs.python.org/3/howto/regex.html
{n-x}(20)

Please enter a valid regex, or enter c to continue.
For help with regex syntax, visit https://docs.python.org/3/howto/regex.html
c

Running the webcrawler with your parameters, please wait.
|

```

Figure 1 Terminal Interface part one

When starting the program, the user will be prompted for input in the terminal. First, the user is given the option to define their own parameters or use the defaults provided by the program. Should the user choose to run the program with the default parameters, presumably for the sake of testing, they will be able to view the results of the analysis afterwards.

When the user chooses to provide their own parameters, they must choose whether to run both the crawler and the analysis together, or either of them separately. This allows analysis of websites that have already been downloaded, or to download a website for manual inspection. The user should enter a URL on the simple domain form, without “www”, and without specifying protocol.

Afterwards, the user should specify the depth of recursion, which pertains to how deep the crawler looks for subpages. Furthermore, the user is asked to enter the maximum/total number of subpages to download. This is separate from depth, as it simply means that the crawler will stop when it has downloaded the maximum number of

subpages. If the user wishes, they can provide a series of regular expressions that will be used to identify information on the website. Continuing from this point will start the web crawler.

```
The web has been successfully crawled! *spider noises*

Which results would you like to inspect?:
1. A list of identified e-mail addresses.
2. A list of identified phone numbers.
3. A dictionary containing comments discovered in the source code.
4. A list of data discovered by the user-defined regex.
5. A list of the 100 most frequently used words on the website.
6. A list of the 100 most frequently used words on the website, and their frequency.
7. All of the above in a big, mushy pile.
Any other entry will terminate the program.
|
```

Figure 2 Terminal Interface part two

When the crawling and the analysis have terminated, the user is presented with multiple options to view the data that has been retrieved. The first two options are quite plain lists containing any e-mail addresses and phone numbers identified on the website. When the information has been displayed, the user will be returned to the list of options.

```
1
['frimerketjenesten@posten.no']

Which results would you like to inspect?:
1. A list of identified e-mail addresses.
2. A list of identified phone numbers.
3. A dictionary containing comments discovered in the source code.
4. A list of data discovered by the user-defined regex.
5. A list of the 100 most frequently used words on the website.
6. A list of the 100 most frequently used words on the website, and their frequency.
7. All of the above in a big, mushy pile.
Any other entry will terminate the program.
|
```

Figure 3 Terminal Interface part three

2

```
['22 03 00 00', '23 14 78 70']
```

Which results would you like to inspect?:

1. A list of identified e-mail addresses.
2. A list of identified phone numbers.
3. A dictionary containing comments discovered in the source code.
4. A list of data discovered by the user-defined regex.
5. A list of the 100 most frequently used words on the website.
6. A list of the 100 most frequently used words on the website, and their frequency.
7. All of the above in a big, mushy pile.

Any other entry will terminate the program.

|

Figure 4 Terminal Interface part four

3

Comments were found in 21 files:

```
httpswwww.posten.no.html
httpswwww.posten.noadresstjenesteradresseendring.html
httpswwww.posten.noadresstjenestermidlertidig-ettersending.html
httpswwww.posten.noadresstjenesteroppbevaring-av-post.html
httpswwww.posten.nofortollingmotta-fra-utlandet.html
httpswwww.posten.nofortollingsende-til-utlandet.html
httpswwww.posten.nofrimerker-til-samling.html
httpswwww.posten.nokundeservice.html
httpswwww.posten.nokundeserviceklager-og-reklamasjon.html
httpswwww.posten.nokundeservicepostens-chatbot.html
httpswwww.posten.nomottahente-selv.html
httpswwww.posten.nomottahente-selvpostboks.html
httpswwww.posten.nomottahjemlevering.html
httpswwww.posten.nomottahjemleveringpostkasse.html
httpswwww.posten.nonettbutikker.html
httpswwww.posten.nosende.html
httpswwww.posten.nosendebrevfrimerker-og-porto.html
httpswwww.posten.nosendeklargjoring.html
httpswwww.posten.nosendepakkeutland.html
httpswwww.posten.nosenderetur.html
httpswwww.posten.nosporing-kundeservice.html
```

The html files can be found in the /output/html directory relative to the main file.

Figure 5 Terminal Interface part five

The third option presents the comments that were uncovered in the source code, by first listing the names of the files they were found in.

```
The html files can be found in the /output/html directory relative to the main file.

Let's take a look at httpswww.posten.no.html:

The comment
    <!-- Navbar -->
was found at line number 53,
and ended on line 53.

The comment
    <!-- Mobile menu -->
was found at line number 86,
and ended on line 86.
```

Figure 6 Terminal Interface part six

Then proceeding to print each comment by order of the files, noting the lines they were found at.

```
The comment
/*# sourceMappingURL=/sm/8e603a4d5cb2c02c40859aeb72a98333df41920ebbf854f81308eca7e3454111.map */
was found at line number 13,
and ended on line 14.

The comment
/*<![CDATA[*]
was found at line number 911,
and ended on line 911.

The comment
/*]]>*/
was found at line number 917,
and ended on line 917.

Let's take a look at httpswww.posten.noadressetjenesteradresseendring.html:
```

Figure 7 Terminal Interface part seven

This continues in short intervals until every comment has been visualized.”

The fourth option shows a list of information identified by matching the regular expression(s) provided by the user with the parsed text content of the html. As one can see

in the next figure, the regex `[a-z]{20}` matched any combination of lowercase letters between a and z that were in a sequence of 20.

```
4
['nnleveringspostkasse', 'oppbevaringsperioden', 'fortollingstjenester',
'dokumentasjonskravet', 'fortollingsprosessen', 'kontraktsforpliktels',
'pakkeutleveringssted', 'leilighetskomplekser', 'kspedisjonskostnaden']

Which results would you like to inspect?:
1. A list of identified e-mail addresses.
2. A list of identified phone numbers.
3. A dictionary containing comments discovered in the source code.
4. A list of data discovered by the user-defined regex.
5. A list of the 100 most frequently used words on the website.
6. A list of the 100 most frequently used words on the website, and their
  frequency.
7. All of the above in a big, mushy pile.
Any other entry will terminate the program.
5
['og', 'du', 'i', 'til', 'om', 'posten', 'for', 'er', 'av', 'kan', 'post',
'norge', 'pakker', 'brev', 'frimerker', 'sende', 'med', 'oss', 'på',
'som', 'fortolling', 'sms', 'det', 'eller', 'alt', 'postkasser', 'en',
'utlandet', 'adresser', 'falske', 'postnummer', 'personer', 'fra',
'på', 'når', 'vi', 'etter', 'adresse', 'at', 'sender', 'når',
'sendingen', 'selv', 'ikke', 'sender', 'har', 'kontakt', 'nettbutikken',
'retur', 'e', 'kundeservice', 'toll', 'hente', 'postboks', 'min',
'porto', 'din', 'postkasse', 'midlertidig', 'priser', 'åpningstider',
'mottar', 'pakken', 'leie', 'oppbevaring', 'skal', 'hjemlevering', 'deg',
'vil', 'chat', 'mer', 'flytte', 'endre', 'å', 'bedrifter', 'side',
'skilt', 'adressering', 'sikkerhet', 'ved', 'klager', 'reklamasjon',
'english', 'personvern', 'samling', 'innpakking', 'ettersende',
'sporingshjelp', 'nettsvindel', 'enklere', 'forsiden', 'jul', 'våre',
'innleveringspostkasser', 'jobb', 'nettmagasin', 'hverdag', 'site',
'cookies', 'søk']

Which results would you like to inspect?:
1. A list of identified e-mail addresses.
2. A list of identified phone numbers.
3. A dictionary containing comments discovered in the source code.
4. A list of data discovered by the user-defined regex.
5. A list of the 100 most frequently used words on the website.
6. A list of the 100 most frequently used words on the website, and their
  frequency.
7. All of the above in a big, mushy pile.
Any other entry will terminate the program.
|
```

Figure 8 Terminal Interface part eight

Option number five is a plain list containing the 100 most frequently used words on the website.

The sixth option presents those same words more elegantly along with the frequency itself.


```
6
The word og was found 832 times.

The word du was found 413 times.

The word i was found 339 times.

The word til was found 293 times.

The word om was found 257 times.

The word posten was found 227 times.

The word for was found 191 times.

The word er was found 184 times.

The word av was found 169 times.

The word kan was found 164 times.
```

Figure 9 Terminal Interface part nine

All 100 of them are printed in short intervals as seen these two figures.

```
The word forsiden was found 42 times.

The word jul was found 42 times.

The word våre was found 42 times.

The word innleveringspostkasser was found 42 times.

The word jobb was found 42 times.

The word nettmagasin was found 42 times.

The word hverdag was found 42 times.

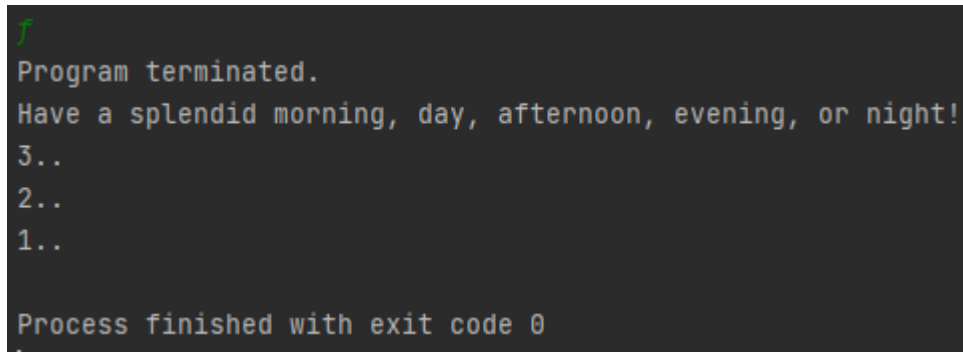
The word site was found 42 times.

The word cookies was found 42 times.

The word søk was found 40 times.
```

Figure 10 Terminal Interface part ten

Option number seven prints all the information in one big pile, for quick reference. That will not be showcased in this report. By entering any other command than the numbers 1 through 7, the program terminates.

A terminal window with a dark background and light-colored text. The text displayed is: 'Program terminated.' followed by 'Have a splendid morning, day, afternoon, evening, or night!' on the next line. Below this are three lines of numbers: '3..', '2..', and '1..'. At the bottom, it says 'Process finished with exit code 0'.

```
Program terminated.  
Have a splendid morning, day, afternoon, evening, or night!  
3..  
2..  
1..  
  
Process finished with exit code 0
```

Figure 11 Terminal Interface part eleven

In the next section, we'll have a look at some of the available solutions.

Overview of Available Solutions

Web Scrapers

There are many tools available for the purpose of web scraping today. Some standalone solutions exist, but most of the tools require some programming skills. A compilation of the best standalone web scrapers can be found on Octoparse's blog (<https://www.octoparse.com/blog/top-20-web-crawling-tools-for-extracting-web-data>), which naturally features Octoparse's own web application for web scraping. Scrapinghub is another big actor among the standalone solutions, and a credited contributor to Scrapy, the Python framework. <https://github.com/BruceDone/awesome-crawler> is a collection of web crawler, spider and resources in different programming languages. For this project, we are

particularly interested in existing Python tools.

Python

- [Scrapy](#) - A fast high-level screen scraping and web crawling framework.
 - [django-dynamic-scraper](#) - Creating Scrapy scrapers via the Django admin interface.
 - [Scrapy-Redis](#) - Redis-based components for Scrapy.
 - [scrapy-cluster](#) - Uses Redis and Kafka to create a distributed on demand scraping cluster.
 - [distribute_crawler](#) - Uses scrapy,redis, mongodb,graphite to create a distributed spider.
- [pyspider](#) - A powerful spider system.
- [CoCrawler](#) - A versatile web crawler built using modern tools and concurrency.
- [cola](#) - A distributed crawling framework.
- [Demiurge](#) - PyQuery-based scraping micro-framework.
- [Scrapely](#) - A pure-python HTML screen-scraping library.
- [feedparser](#) - Universal feed parser.
- [you-get](#) - Dumb downloader that scrapes the web.
- [Grab](#) - Site scraping framework.
- [MechanicalSoup](#) - A Python library for automating interaction with websites.
- [portia](#) - Visual scraping for Scrapy.
- [crawley](#) - Pythonic Crawling / Scraping Framework based on Non Blocking I/O operations.
- [RoboBrowser](#) - A simple, Pythonic library for browsing the web without a standalone web browser.
- [MSpider](#) - A simple ,easy spider using gevent and js render.
- [brownant](#) - A lightweight web data extracting framework.
- [PSpider](#) - A simple spider frame in Python3.
- [Gain](#) - Web crawling framework based on asyncio for everyone.
- [sukhoi](#) - Minimalist and powerful Web Crawler.
- [spidy](#) - The simple, easy to use command line web crawler.
- [newspaper](#) - News, full-text, and article metadata extraction in Python 3
- [aspider](#) - An async web scraping micro-framework based on asyncio.

Figure 12 A collection of web crawler, spider and resources in the Python language

As one can see, there are already so many great tools to choose from when it comes to web scraping. Chances are that when attempting to implement a web scraper from scratch, it won't be as powerful as most of these tools. Scrapy does for instance allow users to provide their own regular expressions to return matches from html content, which is one

the requirements of the project.

```
re(regex, replace_entities=True) [source]
```

Apply the given regex and return a list of unicode strings with the matches.

`regex` can be either a compiled regular expression or a string which will be compiled to a regular expression using `re.compile(regex)`.

By default, character entity references are replaced by their corresponding character (except for `&` and `<`). Passing `replace_entities` as `False` switches off these replacements.

Figure 13 Scrapy's `re` function matches content with a user-provided regular expression

However, the diversity of the existing tools may tell us that there is a demand for specialized tools like the one specified in the project description. Regardless, the task at hand is more about showing programming and problem solving capabilities of the candidate, than it is about creating a new tool that contributes to the web scraping paradigm.

Components

So, what are the components needed to create a useful web scraping tool? The most important key component is downloading the HTML (html) content itself. Furthermore, one or several tools are needed to parse the content in some way, in order to extract the information specified in the project description. Lastly, some form of user interface is needed, where the obvious alternatives are Graphical User Interface and Terminal Interface.

For the acquisition of website content, there are quite a few solutions available. I looked at the HTTP libraries Requests and urllib.request, and the pure download utility wget. Wget would perhaps suffice early in the development for testing purposes, but lacks important HTTP request functionality that web crawling demands. Requests is generally perceived to be the superior choice, for its elegant and simplistic syntax, and is what will be used in this project for HTTP requests.

When it comes to the parsing of html, the obvious options are Python's built-in html.parser, BeautifulSoup and lxml.html. This project implements the latter, which is based on lxml's HTML parser combined with its ElementTree API. lxml.html provides special Element API for HTML elements, as well as a number of utilities from common HTML processing tasks (<https://lxml.de/lxmlhtml.html>). It promises to be several times faster than BeautifulSoup, while implementing features from BeautifulSoup, especially it's encoding

detection. Python's `re` module is used for all regular expression matching operations, because it's fast and easy to use.

If the web crawler was to have a GUI, the package `tkinter` would be the obvious choice for me personally, as I've worked with it before and find it both simple and powerful. Creating a decent GUI from scratch takes a lot of time, however, and is not a part of the project description. The time afforded to this project entails that the user interface will be implemented in the terminal, prompting user input with the `input()` function.

Having explained some of the background of the project, the next section will provide an overview of the design of the software.

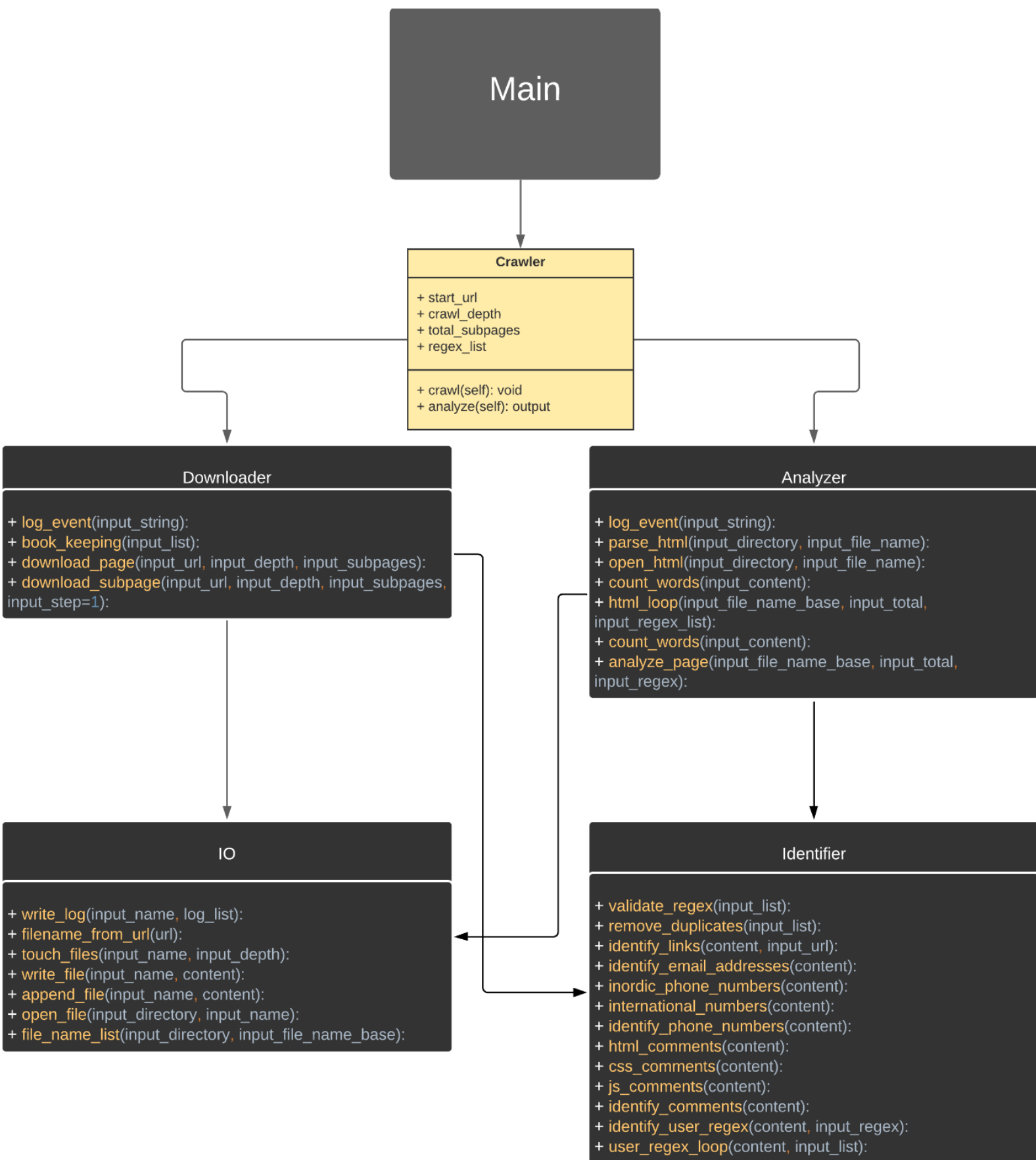


Figure 14 Software structure

Design

The design of the software is a single class `Crawler` that is initialized with the input parameters provided by the user. The package is divided into several Python scripts that contain static functions that the `Crawler` object can use. Refer to Figure 14 Software structure for an overview of the software structure. The main python file outside the package is simply an entry point with an interface that runs the `Crawler` object, and visualizes the output data to the user.

Implementation

Planning

The planning phase was relatively short, because the program specification in the task description was very concise. For the sake of order, I created a package called `crawler` where I put python files named `Crawler`, `Downloader`, `Analyzer` and `Identifier`, each catering to a specific portion of the functionality.

- `Crawler` would be the only class needed, initialized with the parameters defined by the user, serving as a controller in the package. The other files were to contain functions for specific tasks.
- `Downloader` would deal with downloading the website.
- `Identifier` would identify data from the website using predefined patterns.
- `Analyzer` would serve as a mini-controller for `Identifier` while looping through the html, and deal with the word count.
- I later added `IO.py` to provide functions for dealing with, you guessed it, Input / Output – a file handler.

Development

What I knew thus far was that the program would require the aid of a library or package to download the html files themselves. Python's built-in *urllib* was a good option, but I was looking for something a bit purer, and simplistic like ubuntu's *wget*. I discovered that Python has an equivalent download utility, however I chose not to implement it because it's not maintained. The consensus among developers seemed to be that the

requests library was the superior tool for downloading html via *HTTP*. It also allowed solid configuration and error handling through its *Session()* method, like preventing too many redirects and too long load times when fetching html. I also had some previous experience with a similar HTTP package in JavaScript, so that's what I went with.

I proceeded to implement a download function for the web crawler, and tested it on the front page of <https://www.crawler-test.com/>. After inspecting the layout of the html, especially the links to subpages, I designed a regular expression that would look for the `<a>`-tags and return the group inside the *href* attribute that followed the original link. That way I would retrieve a list of strings that make up the path component (starts with a slash after the hostname) of the URLs to the subpages, that I could put together with the base URL and iterate over later.

Through testing on that website I realized that some of the subpages had purposefully long load times, or redirected the HTTP request an certain number of times, even infinitely. That's why I chose to set a *timeout* of 1.3 seconds, and the maximum number of redirects to 3 in the *requests.Session()*-object. When the requests would timeout or be redirected too many times, they would raise exceptions which I in turn had to deal with by wrapping the request in a try/except block. Malformed links also lead to a connection error, which I dealt with in the same way. The program simply logs the event and moves on silently.

Next, I had to make the *download_subpage* function recursive, to fulfil the requirement specified in the project description that the user can decide the depth of the crawling. I interpreted this in such a way that links are only looked for in *subpages* if the depth is *two* or higher. A depth of *one* would entail that links are looked for in the front page and those subpages are downloaded, while a depth of *zero* would mean that only the front page is downloaded.

This posed a problem where a deep crawl would result in a great number of subpages being iterated through for every level of recursion. I solved the issue by creating txt-files corresponding with the depth of the crawling, that the subpages were appended to depending on which level of recursion they were downloaded in. If the depth of the crawling was set to two, three txt files would be created. The current step in the recursion had to be

kept track of in order to search for new links in the correct file. The links that had already been identified and fetched were appended to a list for what I called bookkeeping, to prevent the crawler from downloading the same subpage twice.

After playing around with what I had so far (some call it testing), I discovered that there was a broad variance in the structure of html elements from website to website. The *href* attribute I was looking for wasn't always placed first inside the <a>-tag, in fact there could be multiple attributes before and after it. Back to drawing board I went, several times, and came up with a more robust regex that worked with multiple html styles. Through this iterative testing, I also came upon html files that were not encoded in utf-8, that I had been using in my file handlers up until this point. The `open_file` function had to be wrapped in a try/except block, and if *UnicodeDecodeError* was excepted, the default encoding of `open()` was used instead.

It was at this time I uncovered that the output files mentioned earlier, were overwritten in the recursive function. I solved the issue (which I had created in order to solve another issue), by moving the creation of the files to its own function and calling it at the top level of the `download_page` function. That meant the files were in place before the subpages were downloaded.

The next requirement I decided to tackle was the identification of e-mail addresses and phone numbers. I started looking around for guidance on [stackoverflow.com](https://stackoverflow.com/questions/201323/how-to-validate-an-email-address-using-a-regular-expression), where I found a general question on how to validate e-mail addresses using a regular expression (<https://stackoverflow.com/questions/201323/how-to-validate-an-email-address-using-a-regular-expression>). It became apparent that I needed to implement some of the hard work that others had gone through before me in order to attain a regex for *any* e-mail address that complies with the current specification for e-mail addresses (RFC 5322: <https://www.ietf.org/rfc/rfc5322.txt>).

I used the one I found at <http://emailregex.com/>, which almost sufficed “out of the box”. I ended up removing the slashes found in the name address, the portion before the @, because it could easily be mixed up with other elements in the html. Even though I discovered that it occasionally matched with some resource fetching in JavaScript (using @), I reasoned that it was coincidental and not significant enough to warrant making the regex

less compatible with the current standards for e-mail addresses. I also added support for capital letters in the e-mail regex, in case of the address fields being capitalized by mistake.

I repeated the process for phone numbers, and I found many perfectly good regular expressions on stackoverflow.com for standard 10-digit numbers. US numbers. To start off I particularly wanted to match Norwegian, and perhaps other Nordic numbers, seeing as Swedish numbers are in the same format.

I adapted my findings to design a regex that could match Norwegian phone numbers in a few different formats. Even unformatted, which would simply be 8 digits with no spacing. The unformatted numbers could be mistaken with any 8-digit integer, naturally, so I used negative lookahead to exclude matches that were preceded or followed by certain characters. I deemed that the risk of misinformation was not great enough to warrant making the reach of the regex even narrower. In a worst-case scenario, one would simply have to manually inspect the phone numbers retrieved by the crawler.

The Downloader file was getting a bit cluttered with file-handlers and similar functions, so I moved all such functions to a new file called IO. Simultaneously implementing a `write_log` function for handling logging of timers and errors in a more controlled manner. Testing and debugging by printing to the console can only work for so long.

The program still needed to retrieve a list of the most common words, as specified in the project description. I implemented a function to count all the words that were found in html files, in the Analyzer file. Utilizing string replacement on a wide variety of symbols, I sought to strip away everything by text in the downloaded content. It returned a dictionary with tuples, which elements were the word and the number of times it was counted. Testing this function showed that many of the html tags were carried over even when I replaced the symbols they consisted of. Weird. I also struggled a lot with whitespace like tabs and newlines.

I came to realize that would have to look for advanced regular expressions, or a library or package that could assist my program in parsing html for text content. It turned out that regex would be a very bad idea, because it's quite frankly bad at parsing. *BeautifulSoup* looked like it provided the functionality, but seemed big and clunky, and the consensus was that it was slow for this task. I discovered that the library *lxml* came with a

dedicated Python package for dealing with html: *lxml.html*, able to use *cssselect* to get text from an html element and all its child elements.

With the help of *lxml.html*, the downloader would now create individual html files for all the downloaded subpages in addition to the big txt-files used for recursive link-parsing, in order to parse every subpage's text content effectively. As I perceived I, storing the same data in both the txt-files and html made the program more categorical and effective in its approach, at the expense of disk space. A decent trade-off, especially since the files can be deleted manually as needed.

The filenames were created by stripping away illegal filename characters from the bare URL to each downloaded subpage. I was able to retrieve a list of filenames in the output/html folder that started with the base filename. Allowing html from several websites to remain in the folder, while only getting the subpages from the current website.

I implemented an outer loop in the Analyzer file to parse the text content of the html files and count the words iteratively. *Collections.Counter* was used to mathematically add the values of identical keys in two dictionaries to a new dictionary, thus storing the count globally for each iteration.

Testing at this stage uncovered that on some websites, some of the links to subpages contained *www* between the protocol and the rest of the domain, whereas the URL to the front page did not. The regex couldn't match with these links, since it formatted the base URL into itself. I decided to split the URL parameter I had been passing to the Crawler object, adding <https://www> to the front of it when the object is initialized. Making sure only to provide simplified URL like google.com to the crawler afterwards. At the same time, I changed the regex to accommodate the new input, and added *(?:www)?* to make *www* optional in any links to be matched.

These changes resulted in a lot more links being discovered on certain websites. It took so long I started debugging the download and analyze functions, to see if they were stuck on something. Things were fine. Scraping a random e-commerce site took 39 minutes. Discovered around 600 links, 508 of them were downloaded (no duplicates) which is 13 links per minute. This prompted me to add a limit to the maximum/total number of subpages to download, in addition to the depth of crawling.

One of the two remaining functionalities specified in the project description was identification of special data by matching with user-provided regular expressions. Expressions, plural, so it had to be a list. Made a `user_regex_loop` function that iterated over the list of expressions and return the lists of matches in the text content as one list. Looking over the project description at the time, I noted that I had to define how many of the most frequent words to include in the output myself, so I sliced the results of the word count to the 100 most frequent.

Testing these changes made me realize that I had been overwriting the results of the identifiers for each html file by doing assignment in the loop. This was easily fixed by changing the operator to `+=` for lists and using `update()` for dictionaries.

The only functionality yet to be implemented at this point was identifying comments inside the source code and making a list of them indicating the file name and line number of the comment. It sounded simple at first, but I quickly understood it was a daunting task. Retrieving the comments might be tricky enough, but I would get there with some regex tinkering, I thought. However, up until this point I had been using Python's `re.findall` to match regex in the content, which had no concept of its position in the text, let alone line numbers. And the project description specifically mentions "source code", implying the comments within CSS and JS are also of interest. The other identifiers utilize the parsed text content where typical JS-syntax has been removed as per my intentions.

The first thing that came to mind was that I had to use the un-parsed html to discover as many comments as possible, luckily easily available through the already implemented file-handlers. I constructed the basic regular expressions needed to identify HTML, CSS and JS multiline and inline comment, however not perfect it would suffice for testing. The easiest way to indicate the name of the file in which the comment was found, was to add the filename as a key to a dictionary, with the value of that key being a list containing the comments found within that file. All I needed now was to find a regex method that could traverse line by line. I recalled `finditer` from the back of my head, which is what was also suggested on stackoverflow.com for similar problems. The implementation wasn't straight forward, though, but it boiled down to this:

- Use `finditer` with multiline flag to match content with regular expression

- Loop through the matches
- Slice from the start of the content to the *start* of the match
 - Count the number of newlines on the way
- Slice from the start of the content to the *end* of the match
 - Count the number of newlines on the way
- Return the match, the number of newlines to the start of the match and the number of newlines to the end of the match

With the final requirement implemented, the entry point needed an interface for users. Given the timeframe, a terminal interface based on input queries would suffice.

The entry point isn't so much a part of the solution as it is a script that uses and showcases the program, so I won't go into too much detail on the implementation of `main.py`. However, the user is presented with different options of running the program and given clear explanations of which parameters to define. Comprehensive printing iterators are implemented in the "post-crawler"-loop, where the uses can view the collected data.

Analysis

This section will briefly cover how well the implementation meets the requirements specified in the project description. It has been covered in portions throughout the document, and the reader is advised to refer to the Implementation section for explanations of the thought process behind implementing the requirements.

Input Parameters

As seen in Figure 1 Terminal Interface part one, the user is prompted for the start URL of the web crawling, the depth of the crawling and an optional series of regular expressions. In addition to the three parameters required by the project description, the user can provide a limit to the number of subpages the crawler will download. The user is given thorough explanation of what the parameters do in the interface.

Features

The website is clearly downloaded as specified in the project description. Links are identified and the subpages they direct to are downloaded recursively. "The maximum number of jumps" is a bit vague in the description, but a limit is implemented by taking both

the depth of recursion and the maximum/total number of subpages into account when downloading.

As seen in Figure 2 Terminal Interface part two and onwards, the user is presented with a terminal loop when the web crawler has terminated. E-mail addresses and phone numbers, at least Norwegian numbers, are clearly being identified and presented to the user.

Comments are identified and grouped together by the name of the file they were discovered in. Both the line numbers of the start and the end of the comment are included in the representation, which exceeds the requirements specified in the project description.

The program is clearly able to use the regular expression(s) provided by the user to match strings in the text content of the html. It's worth noting that the html is parsed beforehand, so content that is associated with the source code only will not be traversed by the user-defined regex. The only identifier's that use the raw source code are the link and comment identifiers.

Both a list of the most common words only, and a list containing those words paired with their word count is retrieved and presented to the user. The latter is printed in a tailored and elegant way.

All in all, it's safe to say that the requirements of the project have been met with flying colours.

Testing

The testing was executed iteratively through the process and is mostly documented by short log entries by yours truly. I will also include a description of the debug tool in my IDE of choice, PyCharm.

Test log

First, please find enclosed a list of log entries from the implementation process:

- Used <https://www.crawler-test.com/> to run initial tests on identifying links and downloading subpages
- Used <https://regex101.com/> to test regex on downloaded html

- Tested with different urls, international and Norwegian
- Testing uncovered need to set Timeout to prevent long load times
- Testing uncovered need to set max redirects to prevent infinite loops
- Testing uncovered malformed links lead to connection error
- Testing uncovered html sites can have different encodings, which can break the identification of links after saving the webpage
- Testing uncovered output files were overwritten in recursion
- Testing showed that a lot of html tags are included even when converting special symbols to spaces.
- Testing uncovered a problem with links that refer to urls with www in between https:// and the address. One can use the url without www to get there, but the link regex won't be valid.
- Testing uncovered that routes can contain illegal file characters which haven't been removed up until this point
- Testing uncovered fatal regex weakness that matches link with poorly formatted html, resulting in match with a huge chain of links.
- Testing uncovered that I've been overwriting the results of the identifiers in the analyzer for each html file in the html loop.
- Testing uncovered the comment regex could find ridiculously long "comments" when slashes were escaped and followed by an asterisk in javascript. Made it so that all comments that are taken into consideration must be preceded by whitespace.

Debugger

The debug tool in PyCharm provides the developer with full view of breakpoints, that can either pause the execution or simply log that they've been reached. See Figure 1 Terminal

```

Connected to pydev debugger (build 203.5981.165)
Would you like to set up your own parameters? y/n
~? n
Running the webcrawler with default parameters, please wait.
Breakpoint reached: Identifier.py:7
Breakpoint reached: Identifier.py:17
Breakpoint reached: Identifier.py:18
Breakpoint reached: Identifier.py:17
Breakpoint reached: Identifier.py:17
Breakpoint reached: Identifier.py:17
Breakpoint reached: Identifier.py:18
Breakpoint reached: Downloader.py:71
Breakpoint reached: Downloader.py:83
Breakpoint reached: Downloader.py:92
Breakpoint reached: Downloader.py:83
Breakpoint reached: Downloader.py:92
Breakpoint reached: Downloader.py:83
Breakpoint reached: Downloader.py:92
Breakpoint reached: Downloader.py:83
Breakpoint reached: Downloader.py:92
Breakpoint reached: Downloader.py:83
Breakpoint reached: Downloader.py:92
Breakpoint reached: Downloader.py:83
Breakpoint reached: Downloader.py:92
Breakpoint reached: Downloader.py:83
Breakpoint reached: Downloader.py:92
Breakpoint reached: Downloader.py:83
Breakpoint reached: Downloader.py:92
Breakpoint reached: Downloader.py:83
Breakpoint reached: Downloader.py:92
Breakpoint reached: Analyzer.py:102
Breakpoint reached: Analyzer.py:117

```

Figure 16 PyCharm Debugger part one

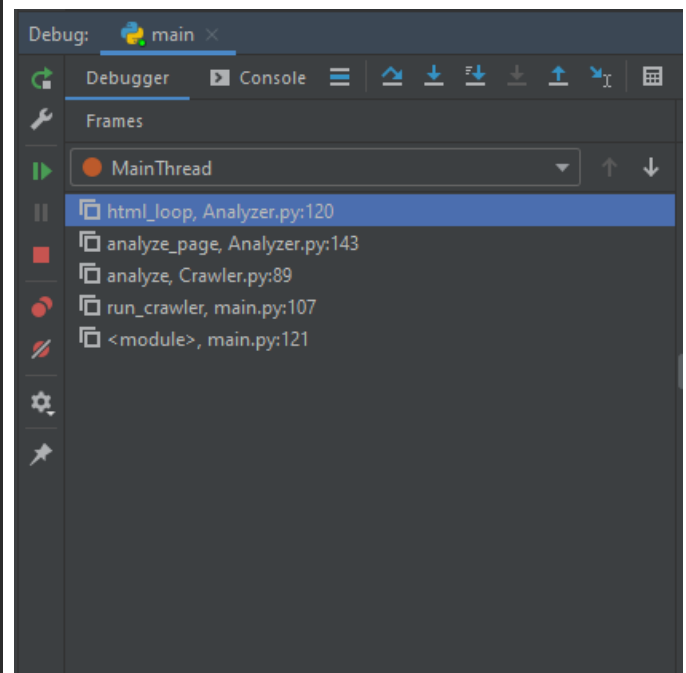


Figure 15 PyCharm Debugger part two

Interface part one for the basic output of the debugger console.

If the execution is paused, as seen in Figure 2 Terminal Interface part two, the halting breakpoint is highlighted in the debugger. To the right of the breakpoint overview, the current variables are presented to the developer. Figure 3 Terminal Interface part three shows the states of the variables when the program executes line 120 in Analyzer.py for the first time, while the states of the variables seen in Figure 4 Terminal Interface part four are from the second time it's executed. The information identified in the first html file has been appended to some of the data structures.

```

comments = {dict: 0} {}
  01 __len__ = (int) 0
emails = {list: 0} []
  01 __len__ = (int) 0
file_name = {str} 'httpswww.posten.no.html'
file_name_list = {list: 21} ['httpswww.posten.no.html', 'httpswww.posten.noadresse
  01 00 = {str} 'httpswww.posten.no.html'
  01 01 = {str} 'httpswww.posten.noadresetjenesteradresseendring.html'
  01 02 = {str} 'httpswww.posten.noadresetjenestermidlertidig-ettersending.html'
  01 03 = {str} 'httpswww.posten.noadresetjenesteroppbevaring-av-post.html'
  01 04 = {str} 'httpswww.posten.nofortollingmotta-fra-utlandet.html'
  01 05 = {str} 'httpswww.posten.nofortollingsende-til-utlandet.html'
  01 06 = {str} 'httpswww.posten.nofrimerker-til-samling.html'
  01 07 = {str} 'httpswww.posten.nokundeservice.html'
  01 08 = {str} 'httpswww.posten.nokundeserviceklager-og-reklamasjon.html'
  01 09 = {str} 'httpswww.posten.nokundeservicepostens-chatbot.html'
  01 10 = {str} 'httpswww.posten.nomottahente-selv.html'
  01 11 = {str} 'httpswww.posten.nomottahente-selvpostboks.html'
  01 12 = {str} 'httpswww.posten.nomottahjemlevering.html'
  01 13 = {str} 'httpswww.posten.nomottahjemleveringpostkasse.html'
  01 14 = {str} 'httpswww.posten.nonettbutikk.html'
  01 15 = {str} 'httpswww.posten.nosende.html'
  01 16 = {str} 'httpswww.posten.nosendebrevfrimerker-og-porto.html'
  01 17 = {str} 'httpswww.posten.nosendeklargjoring.html'
  01 18 = {str} 'httpswww.posten.nosendepakkeutland.html'
  01 19 = {str} 'httpswww.posten.nosenderetur.html'
  01 20 = {str} 'httpswww.posten.nosporing-kundeservice.html'
  01 __len__ = (int) 21
input_file_name_base = {str} 'httpswww.posten.no'
input_regex_list = {list: 2} [['a-z]{20}', '[0-9]{20}']
  01 0 = {str} '[a-z]{20}'
  01 1 = {str} '[0-9]{20}'
  01 __len__ = (int) 2
input_total = (int) 11
new_comments = {list: 0} []
  01 __len__ = (int) 0
parsed_html = [_ElementUnicodeResult] \n \n \n\n \n \n\n \n \n \n
phone_numbers = {list: 0} []
  01 __len__ = (int) 0
t_start = {float} 1608211739.4715135
user_specified_patterns = {list: 0} []
  01 __len__ = (int) 0
word_count = {dict: 0} {}

```

Figure 17 PyCharm Debugger part three

```

comments = {dict: 1} {'httpswww.posten.no.html': [('<!-- Navbar -->', 53, 53), ('<!--
  01 'httpswww.posten.no.html' = {list: 19} [('<!-- Navbar -->', 53, 53), ('<!-- Mobile
  01 __len__ = (int) 1
emails = {list: 0} []
  01 __len__ = (int) 0
file_name = {str} 'httpswww.posten.noadresetjenesteradresseendring.html'
file_name_list = {list: 21} ['httpswww.posten.no.html', 'httpswww.posten.noadreset
  01 00 = {str} 'httpswww.posten.no.html'
  01 01 = {str} 'httpswww.posten.noadresetjenesteradresseendring.html'
  01 02 = {str} 'httpswww.posten.noadresetjenestermidlertidig-ettersending.html'
  01 03 = {str} 'httpswww.posten.noadresetjenesteroppbevaring-av-post.html'
  01 04 = {str} 'httpswww.posten.nofortollingmotta-fra-utlandet.html'
  01 05 = {str} 'httpswww.posten.nofortollingsende-til-utlandet.html'
  01 06 = {str} 'httpswww.posten.nofrimerker-til-samling.html'
  01 07 = {str} 'httpswww.posten.nokundeservice.html'
  01 08 = {str} 'httpswww.posten.nokundeserviceklager-og-reklamasjon.html'
  01 09 = {str} 'httpswww.posten.nokundeservicepostens-chatbot.html'
  01 10 = {str} 'httpswww.posten.nomottahente-selv.html'
  01 11 = {str} 'httpswww.posten.nomottahente-selvpostboks.html'
  01 12 = {str} 'httpswww.posten.nomottahjemlevering.html'
  01 13 = {str} 'httpswww.posten.nomottahjemleveringpostkasse.html'
  01 14 = {str} 'httpswww.posten.nonettbutikk.html'
  01 15 = {str} 'httpswww.posten.nosende.html'
  01 16 = {str} 'httpswww.posten.nosendebrevfrimerker-og-porto.html'
  01 17 = {str} 'httpswww.posten.nosendeklargjoring.html'
  01 18 = {str} 'httpswww.posten.nosendepakkeutland.html'
  01 19 = {str} 'httpswww.posten.nosenderetur.html'
  01 20 = {str} 'httpswww.posten.nosporing-kundeservice.html'
  01 __len__ = (int) 21
input_file_name_base = {str} 'httpswww.posten.no'
input_regex_list = {list: 2} [['a-z]{20}', '[0-9]{20}']
  01 0 = {str} '[a-z]{20}'
  01 1 = {str} '[0-9]{20}'
  01 __len__ = (int) 2
input_total = (int) 11
new_comments = {dict: 1} {'httpswww.posten.no.html': [('<!-- Navbar -->', 53, 53),
  01 'httpswww.posten.no.html' = {list: 19} [('<!-- Navbar -->', 53, 53), ('<!-- Mobile
  01 __len__ = (int) 1
parsed_html = [_ElementUnicodeResult] \n \n \n\n \n \n\n \n \n \n \n \n \n \n \n \n
phone_numbers = {list: 1} ['22 03 00 00']
  01 0 = {str} '22 03 00 00'
  01 __len__ = (int) 1
t_end = {float} 1608211892.7748945
t_start = {float} 1608211892.7748945
user_specified_patterns = {list: 1} ['nnleveringspostkasse']
  01 0 = {str} 'nnleveringspostkasse'
  01 __len__ = (int) 1
word_count = {dict: 203} {'til': 9, 'forsiden': 2, 'min': 3, 'side': 3, 'søk': 3, 'etter': 4, 'adr

```

Figure 18 PyCharm Debugger part four

```

def html_loop(input_file_name_base, input_total, input_regex_list):
    """
    Iterates over the html files corresponding with the base URL.
    Identifies specific information present on the website.
    :param input_file_name_base: String
    :param input_total: Integer
    :param input_regex_list: List of regular expressions (String).
    :return: A list of e-mail addresses, a list of phone numbers,
    a dictionary of comments and their position in the file,
    a list of the 100 most frequently used words on the website,
    and lastly a list of those same words paired with the frequency itself.
    """
    global number_of_analyzed_pages
    file_name_list = IO.file_name_list(html_directory, input_file_name_base)
    file_name_list: ['httpswww.posten.no.html', 'httpswww.posten.noaddresssetjenesteradresseendring.html']
    emails, phone_numbers, comments, new_comments, user_specified_patterns, word_count = [], [], {}, [], [], {}
    for file_name in file_name_list:
        file_name: 'httpswww.posten.noaddresssetjenesteradresseendring.html'
        # Check if maximum amount of pages has been reached.
        if number_of_analyzed_pages < input_total:
            t_start = time.time()
            t_start: 1608211892.7748945
            parsed_html = parse_html(html_directory, file_name)
            parsed_html: \n \n \n\n \n \n\n
            emails += Identifier.identify_email_addresses(parsed_html)
            phone_numbers += Identifier.identify_phone_numbers(parsed_html)
            new_comments = {file_name: Identifier.identify_comments(open_html(html_directory, file_name))}
            comments.update(new_comments)
            user_specified_patterns += Identifier.user_regex_loop(parsed_html, input_regex_list)
            word_count.update(dict(Counter(word_count) + Counter(count_words(parsed_html))))
            number_of_analyzed_pages += 1
            t_end = time.time()
            t_end: 1608211892.7748945
            log_event("Analyzing {} took {} seconds.".format(file_name, t_end - t_start))
    sorted_word_count = sorted(word_count.items(), key=lambda item: item[1], reverse=True)[:100]
    hundred_most_frequent_words = [x[0] for x in sorted_word_count]
    return Identifier.remove_duplicates(emails), Identifier.remove_duplicates(phone_numbers), \
    comments, Identifier.remove_duplicates(user_specified_patterns), \
    hundred_most_frequent_words, sorted_word_count

```

Figure 19 PyCharm Debugger part five

The debugger also provided in-line inspection of data structures that were currently being operated on, as visualized in Figure 19 PyCharm Debugger part five. Use of the debugger helped uncover fatal flaws like assignment errors and files being overwritten.

Evaluation

The project has been a success. If it had been afforded more time, the result might have been prettier and easier to use. However, the functionality required by the project task has been met in an eloquent and thought out way. The intentions of the user have been considered, and the code has been well documented. The process has also been heavily documented, almost like a journey, providing useful information to anyone who would undergo similar tasks in the future.