亞洲大學
ASIA UNIVERSITY

Midterm Project Report

Advanced Computer Programming

Student Name	**: Nolando Alvin**
Student ID	**: 113021132**
Teacher	**: DINH-TRUNG VU**

2025-04

Chapter 1 Introduction

1.1 Github

1) **Personal Github Account**: 113021132
2) **Group Project Repository**: https://github.com/113021134/acp-1132

1.2 Overview

To build a program that is both efficient and easy to maintain, this project makes use of several well-established Python libraries along with some of the language's more expressive and powerful capabilities. From handling web requests and parsing data to organizing the code in a modular way, each component plays a key role in supporting the program's overall goals.

The primary library used is Scrapy, an open-source web crawling framework written in Python that is widely used for extracting structured data from websites. It allows developers to write small programs called "spiders" that automatically navigate through web pages, extract specific information, and store it in various formats such as JSON, CSV, or XML. Scrapy is designed for web scraping at scale and offers built-in tools for handling requests, following links, and managing large volumes of data efficiently.

One of Scrapy's core strengths is its ability to automate the data collection process. A Scrapy spider typically begins with a starting URL, from which it downloads the page's HTML content. It then parses the HTML using powerful selectors to locate specific pieces of information, such as text, links, images, or metadata. This makes it ideal for tasks such as price monitoring, job aggregation, research, or collecting datasets from public websites.

In contrast to simpler scraping tools like BeautifulSoup, Scrapy is more robust and scalable. It handles asynchronous requests, allowing it to scrape multiple pages at once, making the process significantly faster. Moreover, Scrapy is capable of following internal links across multiple pages, making it a useful tool for crawling websites in-depth, such as navigating through a user's GitHub repositories and collecting data from each one.

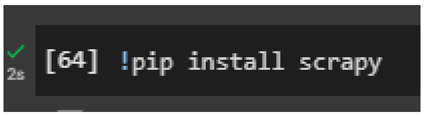
In practical applications, Scrapy can be used to collect job postings from employment websites such as Indeed or LinkedIn. For instance, a spider can be configured to visit a job search results page, extract data such as job titles, company names, locations, salary ranges, and job descriptions, and then store this information in a structured format like a CSV file. This can be useful for individuals analyzing the job market, researchers studying employment trends, or companies conducting competitive analysis.

The requests library was employed to send HTTP requests to GitHub's REST API, crucial for retrieving accurate commit counts, which are not always directly accessible via regular web page scraping. To interpret the API response headers, particularly the pagination links, the built-in re (regular expressions) module was used. This allowed the extraction of the total number of pages, which corresponds to the number of commits from the 'Link' header in the API response.

The code also demonstrates several advanced Python features. It uses CSS selectors provided by Scrapy to accurately target and extract HTML elements, enabling precise data collection from GitHub pages. Conditional expressions, such as about.strip() if about else None, are used to handle optional fields gracefully and reduce the need for verbose if-else statements. Scrapy's meta parameter is another advanced feature utilized to pass data between different stages of the scraping process—particularly between the repository list parser and the individual repository parser. Furthermore, try-except blocks are used for robust error handling, ensuring that the program continues running even if some API requests fail or unexpected data is encountered.

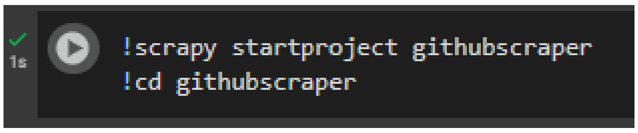
Implementation

My project's code is structured within Google Colab notebook, which allows for the execution of code in individual cells. This cell-based approach enables a step-by-step process of setting up the scraping environment, defining the scraper, and running it. For my project, here are the 5 cells of code and their code also found on my Google Colab https://colab.research.google.com/drive/1tuVA4VCfS95d56J8kcFfC0AXWXu9lo5t?usp=sharing since the screenshots are clearly blurry.



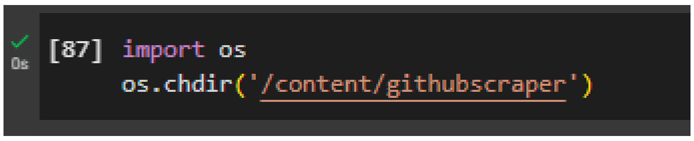
```
[64] !pip install scrapy
```

(1) First Cell



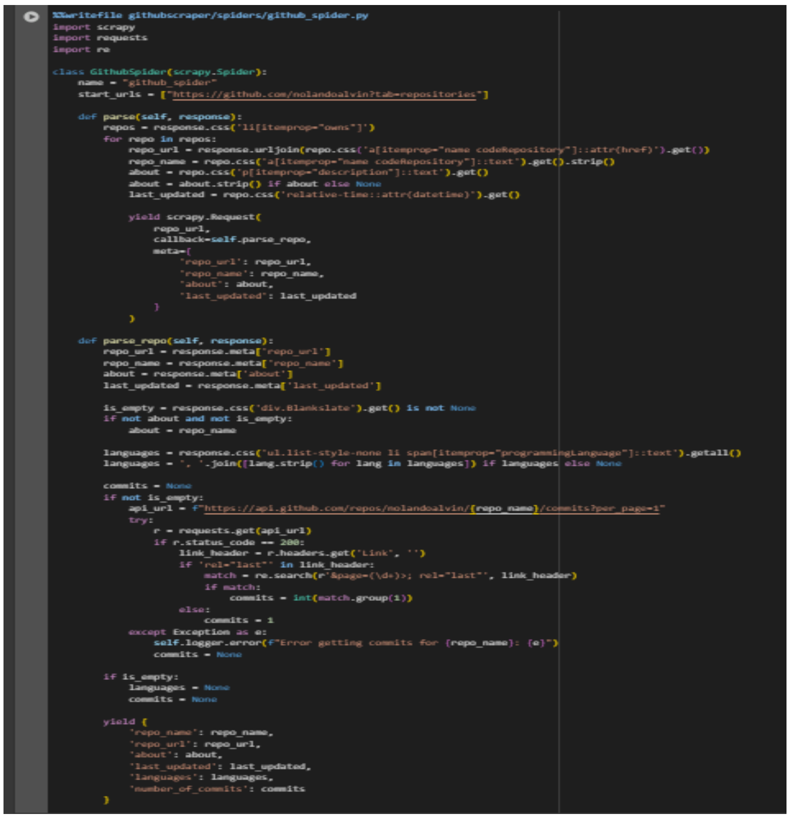
```
!scrapy startproject githubscraper
!cd githubscraper
```

(2) Second Cell



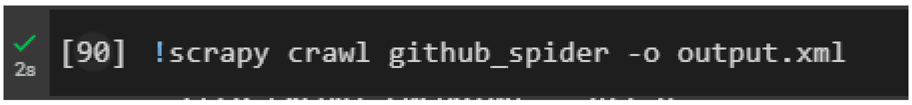
```
[87] import os
     os.chdir('/content/githubscraper')
```

(3) Third Cell



```
%%writefile githubscraper/spiders/github_spider.py
import scrapy
import requests
import re

class GithubSpider(scrapy.Spider):
    name = "github_spider"
    start_urls = ["https://github.com/nolandoalvin?tab=repositories"]

    def parse(self, response):
        repos = response.css('li[itemprop="owns"]')
        for repo in repos:
            repo_url = response.urljoin(repo.css('a[itemprop="name codeRepository"]::attr(href)').get())
            repo_name = repo.css("a[itemprop="name codeRepository"]::text").get().strip()
            about = repo.css('p[itemprop="description"]::text').get()
            about = about.strip() if about else None
            last_updated = repo.css('relative-time::attr(datetime)').get()

            yield scrapy.Request(
                repo_url,
                callback=self.parse_repo,
                meta={
                    'repo_url': repo_url,
                    'repo_name': repo_name,
                    'about': about,
                    'last_updated': last_updated
                }
            )

    def parse_repo(self, response):
        repo_url = response.meta['repo_url']
        repo_name = response.meta['repo_name']
        about = response.meta['about']
        last_updated = response.meta['last_updated']

        is_empty = response.css('div.Blankslate').get() is not None
        if not about and not is_empty:
            about = repo_name

        languages = response.css('ul.list-style-none li span[itemprop="programmingLanguage"]::text').getall()
        languages = ', '.join([lang.strip() for lang in languages]) if languages else None

        commits = None
        if not is_empty:
            api_url = f"https://api.github.com/repos/nolandoalvin/{repo_name}/commits?per_page=1"
            try:
                r = requests.get(api_url)
                if r.status_code == 200:
                    link_header = r.headers.get('Link', '')
                    if 'rel="last"' in link_header:
                        match = re.search(r'&page=(\d+)>; rel="last"', link_header)
                        if match:
                            commits = int(match.group(1))
                    else:
                        commits = 1
            except Exception as e:
                self.logger.error(f"Error getting commits for {repo_name}: {e}")
                commits = None

        if is_empty:
            languages = None
            commits = None

        yield {
            'repo_name': repo_name,
            'repo_url': repo_url,
            'about': about,
            'last_updated': last_updated,
            'languages': languages,
            'number_of_commits': commits
        }
```

(4) Fourth Cell



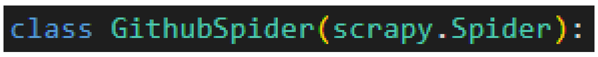
```
[90] !scrapy crawl github_spider -o output.xml
```

(5) Fifth Cell

1.1 Class 1

GithubSpider is the class as the core component responsible for defining the scraping logic. It instructs Scrapy on which websites to crawl, how to navigate those sites, and what data to extract.

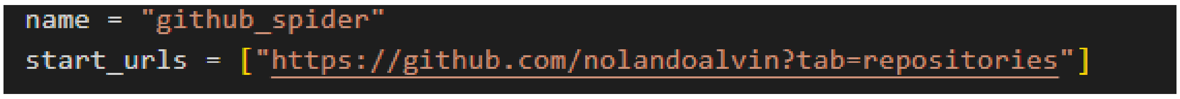
The class parameter is filled with scrapy.Spider which pretty much means the particular GithubSpider class is inheriting the Spider class which is coming from the scrapy module I imported earlier. Inheriting is great as it allows for the reuse of existing code without rewriting it, allowing us to use/build on top of what already exists. Essentially a great save of time.



```python
class GithubSpider(scrapy.Spider):
```

1.1.1 Fields

There are two fields used under the class GithubSpider, the name field isn’t particularly important, it just presents another way of running the scraping where we can call the name instead of the method itself. As for the start_urls, it is a list where the Spider would begin crawling.

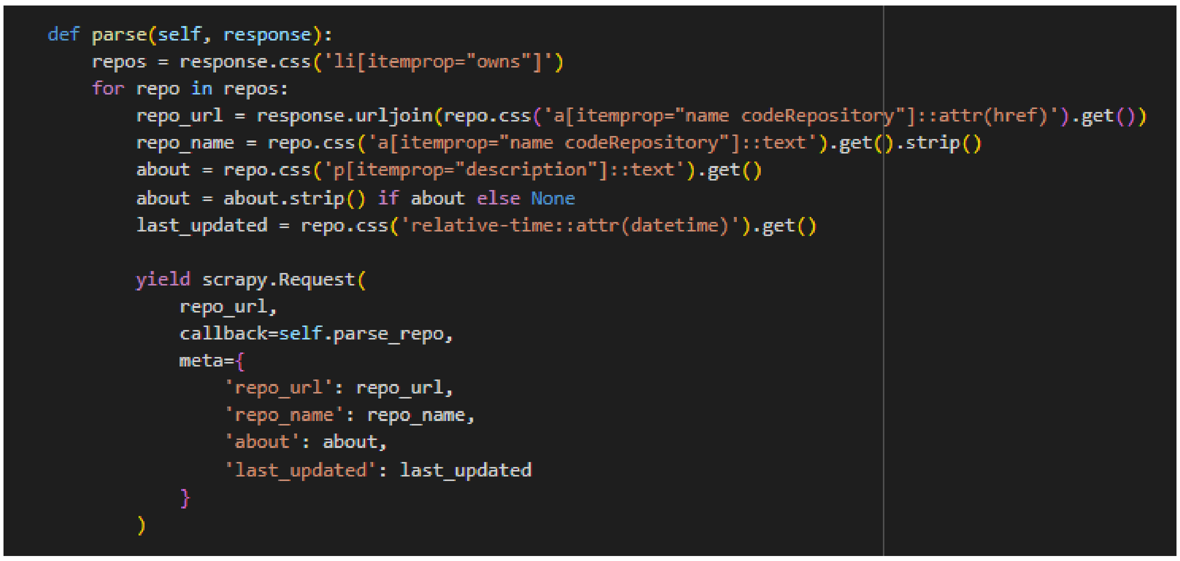


```python
    name = "github_spider"
    start_urls = ["https://github.com/nolandoalvin?tab=repositories"]
```

1.1.2 Methods and/or Functions

Other than class and fields, there are also methods and/or functions depending on where they are positioned. Functions are independent blocks of code that can be called from anywhere, while methods are tied to objects or classes and need an object or class instance to be invoked.

Inside the code, particularly the fourth cell, there are two methods namely def parse(self, response) and def parse_repo(self, response). We will take a look at the first method as seen below.



```python
def parse(self, response):
    repos = response.css('li[itemprop="owns"]')
    for repo in repos:
        repo_url = response.urljoin(repo.css('a[itemprop="name codeRepository"]::attr(href)').get())
        repo_name = repo.css('a[itemprop="name codeRepository"]::text').get().strip()
        about = repo.css('p[itemprop="description"]::text').get()
        about = about.strip() if about else None
        last_updated = repo.css('relative-time::attr(datetime)').get()

        yield scrapy.Request(
            repo_url,
            callback=self.parse_repo,
            meta={
                'repo_url': repo_url,
                'repo_name': repo_name,
                'about': about,
                'last_updated': last_updated
            }
        )
```

This method is mainly to discover all the individual public repositories listed on the initial GitHub user's page and prepare Scrapy to visit each of those repository pages for further data extraction.

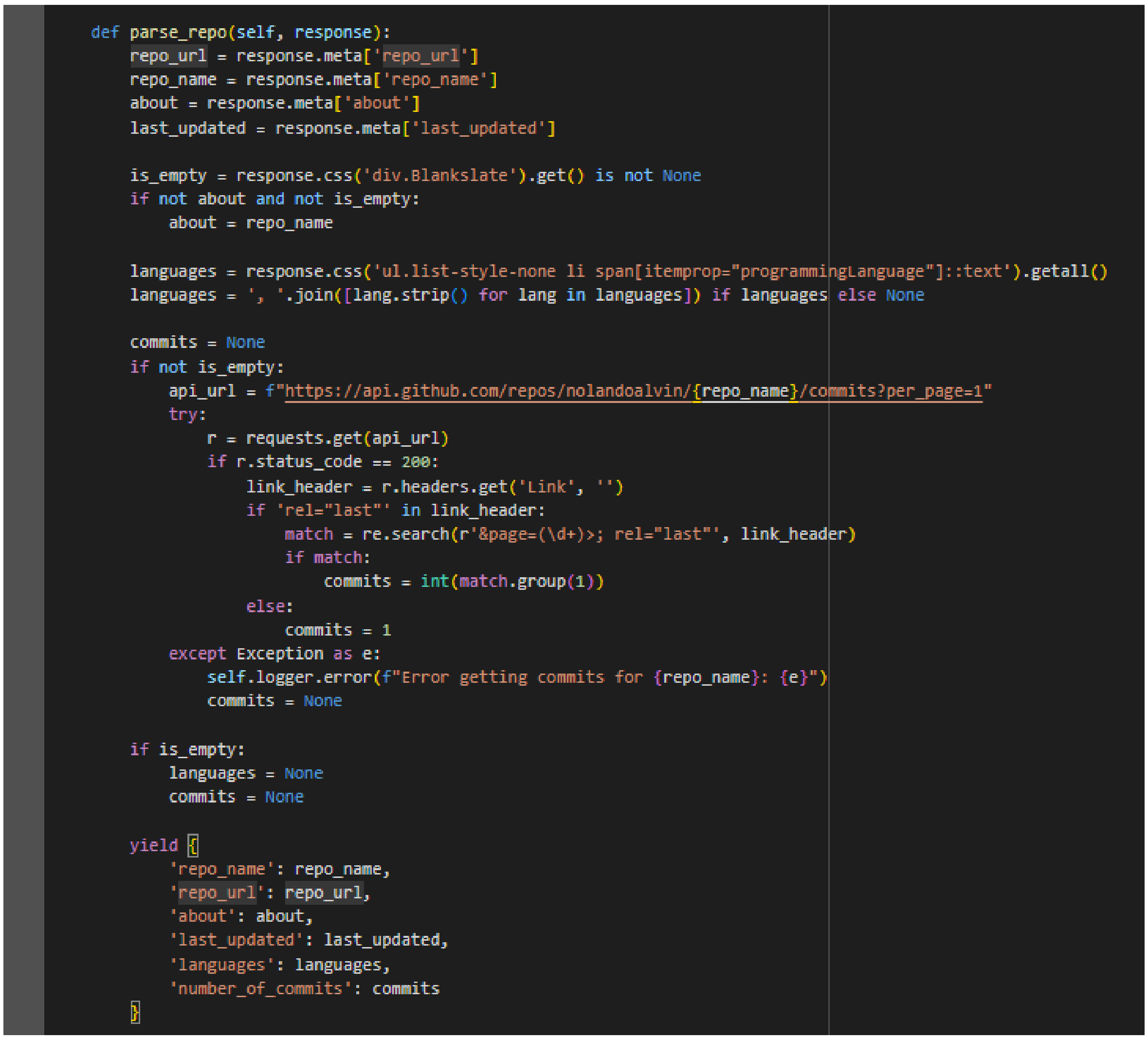
When Scrapy starts crawling the URLs in start_urls, it makes initial request objects. When the responses to these initial requests are received, Scrapy automatically calls this parse method and passes the corresponding response object as an argument. This response contains the HTML of the GitHub user's repositories page.

It uses a CSS selector (response.css('li[itemprop="owns"]')) to find all the HTML elements on the initial page that belong to individual repositories using list items and itemprop=owns whether it belongs to the repository or just an HTML element that doesn't belong to the repositories.

After that, it extracts the repository URL, repository name, repository about, and last updated after then it creates a new request to visit individual repository pages and tells scrapy how to process contents of those pages. It also passes the information we extracted earlier (in the form of meta).

Thanks to the new requests, Scrapy then takes these new requests and start visiting the individual repository pages and the downloader fetches HTML content of each individual repository pages.

Once the content of an individual repository page is downloaded, Scrapy calls the def parse_repo(self, response) method, passing it the response object (containing the HTML of that repository page) and the meta (containing the basic information passed from the parse method).



```python
def parse_repo(self, response):
    repo_url = response.meta['repo_url']
    repo_name = response.meta['repo_name']
    about = response.meta['about']
    last_updated = response.meta['last_updated']

    is_empty = response.css('div.Blankslate').get() is not None
    if not about and not is_empty:
        about = repo_name

    languages = response.css('ul.list-style-none li span[itemprop="programmingLanguage"]::text').getall()
    languages = ', '.join([lang.strip() for lang in languages]) if languages else None

    commits = None
    if not is_empty:
        api_url = f"https://api.github.com/repos/nolandoalvin/{repo_name}/commits?per_page=1"
        try:
            r = requests.get(api_url)
            if r.status_code == 200:
                link_header = r.headers.get('Link', '')
                if 'rel="last"' in link_header:
                    match = re.search(r'&page=(\d+)>; rel="last"', link_header)
                    if match:
                        commits = int(match.group(1))
                else:
                    commits = 1
        except Exception as e:
            self.logger.error(f"Error getting commits for {repo_name}: {e}")
            commits = None

    if is_empty:
        languages = None
        commits = None

    yield {
        'repo_name': repo_name,
        'repo_url': repo_url,
        'about': about,
        'last_updated': last_updated,
        'languages': languages,
        'number_of_commits': commits
    }
```

After we received the meta data from our earlier def parse(self, response) method, we start by retrieving said data to be further used within our methods. Firstly it examines the HTML of the individual repository page to see if it contains a div element with the class Blankslate. If it does, it suggests the repository is empty. It also checks the about, if the about information is still None (meaning it wasn't found on the initial overview page) and the repository isn't empty, the code sets the about to be the repo_name as a fallback.

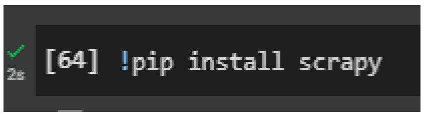
Next, it uses a CSS selector to find all the programming languages listed on the individual repository page. It then formats the list of languages into a comma-separated string, or sets languages to None if no languages were found.

Now, scraping the amount of commits was quite difficult. It required use of GitHub's API. It begins by checking whether the repository is empty, if not then it constructs the URL for the GitHub API to get commit information for the specific repository. It attempts to make an API request to get the commit count by checking the Link header for pagination. If successful, it extracts the total number of commits. If an error occurs during the API request, commits is set to None.

It then checks again whether the repository is empty and sets both the language and commits to None if it is.

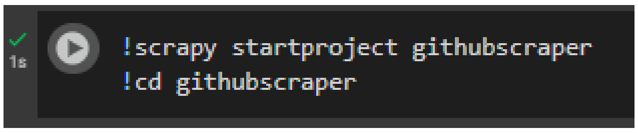
Finally, a Python dictionary is created containing all the extracted information for this repository (repo_name, repo_url, about, last_updated, languages, number_of_commits), and this dictionary is yielded. This represents one complete scraped item for a single repository. So, one execution of the yield statement in parse_repo corresponds to the complete set of information scraped for one individual repository.

Now that the fourth cell is finished, we start first with the first cell. It is preceded with an exclamation mark (!) which means the code is to be run within the Terminal (windows Terminal/Shell) to install the Scrapy library or module.



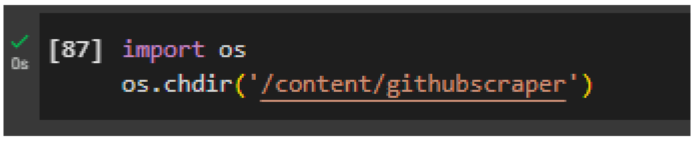
```
[64] !pip install scrapy
```

Then the second cell, Scrapy generates a folder structure setting up the basic framework and changes the terminal/shell's working directory (cd) as well, which is the standard of Linux/Unix command since Google Colab, while running in the cloud, provides a Linux-based environment for its execution. By changing the directory to githubscraper, we ensure that subsequent commands are executed within the context of our Scrapy project.



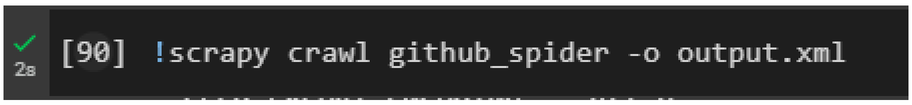
```
!scrapy startproject githubscraper
!cd githubscraper
```

Next off the third cell, we begin with importing the os module to allow access functions for interacting with the operating system. os.chdir('/content/githubscraper') specifically tells the Python interpreter to change its current working directory to /content/githubscraper. It isn't preceded with an exclamation mark, it is different from the previous directory change. This time, it changes the Python Interpreter's working directory



```
[87] import os
     os.chdir('/content/githubscraper')
```

Finally the fifth cell, it is simply a terminal/shell command telling scrapy to start the web scraping. The crawl command tells scrapy to start scraping with a specific spider, in this case the github_spider earlier we assigned the field as. -o output.xml basically outputs the scraped data to a .xml file.



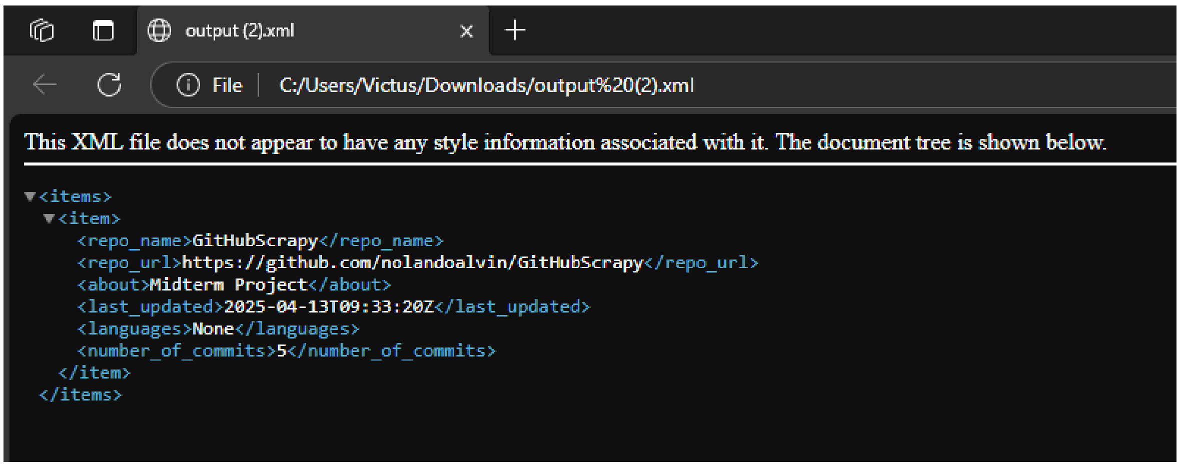
```
[90] !scrapy crawl github_spider -o output.xml
```

Chapter 2 Results

1.1 Result 1

.

There is only one repository available to be scraped on my GitHub repository list as the others are private. Here is the output from my program showing the repository name, repository URL, the about, when it was last updated, the languages used within the repository, and lastly the number of commits.



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<items>
  ▼<item>
     <repo_name>GitHubScrapy</repo_name>
     <repo_url>https://github.com/nolandoalvin/GitHubScrapy</repo_url>
     <about>Midterm Project</about>
     <last_updated>2025-04-13T09:33:20Z</last_updated>
     <languages>None</languages>
     <number_of_commits>5</number_of_commits>
   </item>
 </items>
```

Chapter 3 Conclusions

Within the program, extracting data using Scrapy was successfully done even with unexpected problems where the GitHub API was eventually required to be used within my program. In conclusion, this implementation demonstrates a functional Scrapy spider capable of navigating a GitHub user's repository page, extracting relevant details for each repository, and outputting the structured data into an XML file. The use of Scrapy's request/response cycle, CSS selectors, and the yield keyword enables efficient and organized web scraping. The inclusion of API interaction for commit counts further enriches the collected data. The final output.xml file contains the structured information about the specified GitHub user's public repositories.