**Domain Specific Language for web scraping**

1. *What to build?*

We are building a domain specific language (DSL). A domain specific language is a programming language designed to address a particular domain (Ex:SQL) with a particular programming construct. Our Programming language is aimed at providing a simple, powerful and user-friendly language for performing operations pertaining to web-scraping. This platform is built on python.

1. Why we need to build that?

Web-Scraping is fairly simple considering the modern-day tools like Beautiful-Soup, Scrapy, Requests and Selenium available at our disposal.

But why a DSL?

First, Let’s take a look at what we actually using web-scraping:

1. A web crawler (also known as a web spider or web robot) is a program or automated script which browses the World Wide Web in a methodical, automated manner. This process is called Web crawling or spidering. Many legitimate sites, in particular search engines, use spidering as a means of providing up-to-date data.

Our platform helps in performing simple and complex web-scraping operations just using simple English.

The programming language is divided into 4 functionalities: 1) Providing details/ Metadata if the visiting site. 2) Getting data required by the user from the site. 3) To filter data as per the requirement of the user.

Our language would also provide extensible control over downloadable/viewable content by providing conditional statements and Regex.

Looping constructs could also be implemented.

1. What not to build?

Will Get back to this later.

1. What we provide?

-> We provide an IDE in which we write the code in (We could also provide a CLI).

-> Our objective is to provide 4 basic functionalities to the user:   
 1) “view”, 2) “get”,3) “meta”, 4) “filter”, 5) “select”, 6) “to\_file”, 7) “to\_dir” 8) “help”

1) The “view” command: Similar to the ls command in Linux, this lists the various contents present in the web-page.

2) The “get” command: Basically, fetches the contents of a specific type of data specified.

3) The “meta” command: The meta command is complimented with the “view” command. The “meta” command provides the metadata contents of the contents of the list of items fetched by the “view” command.

4) The “Filter” command: As the name suggests, filters the unwanted items present on the web page.

5) The “select” command is used for the selecting certain data from the lot as per a requirement from the user.

6) The “to\_file” command is used to store the contents from the pull request to a designated file by specifying the file path.

7) The “to\_dir” command is used to a list of files created to a directory.

8) With the usage of regex, minimalization of complexity in finding the desired result is possible.

9) The “help” command would give a briefing of the commands in the DSL.

The finalization if the code is interpreted or compiled is still under speculation.

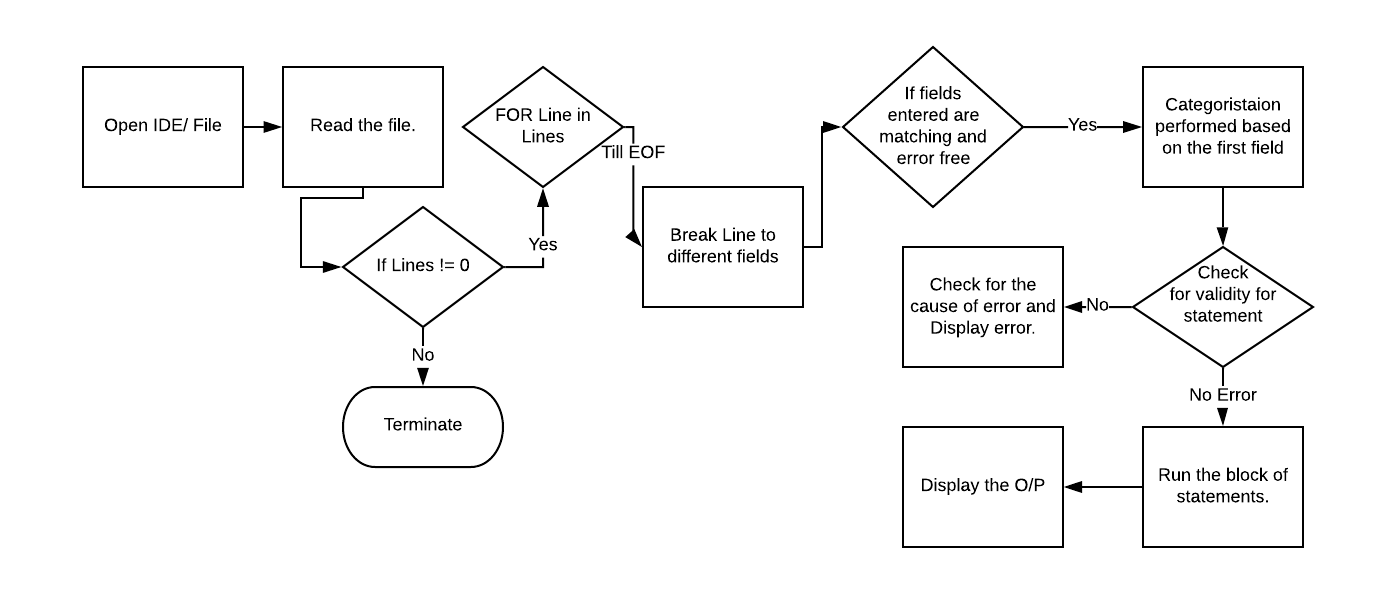
1. Who are our target users?

We look at targeting the naïve users (General Users) and the sophisticated users (Programmers). Since our Language has very little or no syntax whatsoever. Almost anyone with a fair amount of grasp on English and a bit of logic would be able to perform the basic 4 functionalities (Mentioned above).

For performing advanced/ complex operations, we provide the users with conditional statements, looping constructs or Regex.

Analysis and Design:

Low-Level Design:



High-Level Design:

Once Implementation is ongoing.

Review of Literature:

* The Links mentioned below talk about the detailed approach to formulating a DSL with different views/ analysis and inferences.

1. <https://tomassetti.me/how-to-create-programming-language/>: Explains how we should go about creating a programming language.
2. <https://dbader.org/blog/writing-a-dsl-with-python>: Writing a Domain Specific Language in Python.
3. <https://www.researchgate.net/publication/317173940_Usability_Evaluation_of_Domain-Specific_Languages_A_Systematic_Literature_Review>: Usability Evaluation of Domain-Specific Languages: A Systematic Literature Review
4. <https://www.semanticscholar.org/paper/Domain-Specific-Language-for-Web-Scraper-Arifanto-Asnar/42338d59464da70eda493b77e11c4b82c1647a88>: Domain Specific Language for Web Scraper Development.
5. <https://ieeexplore.ieee.org/document/8705842>: Another DSL for Web Scraping.

* The Links attached below show a constructive way of assessing product management. It initiates the learning of an engineer to come to a clear understanding of the product being built.

1. <https://hackernoon.com/product-101-for-engineers-83ef7260cbf1>.

References:

|  |  |
| --- | --- |
| Reference Links | Details |
| <https://tomassetti.me/how-to-create-programming-language/> | [1] Explains how we should go about creating a programming language |
| <https://dbader.org/blog/writing-a-dsl-with-python> | [2] Writing a Domain Specific Language in Python. |
| <https://www.researchgate.net/publication/317173940_Usability_Evaluation_of_Domain-Specific_Languages_A_Systematic_Literature_Review> | [3] Usability Evaluation of Domain-Specific Languages: A Systematic Literature Review |
| <https://www.semanticscholar.org/paper/Domain-Specific-Language-for-Web-Scraper-Arifanto-Asnar/42338d59464da70eda493b77e11c4b82c1647a88> | [4] Domain Specific Language for Web Scraper Development. |
| <https://ieeexplore.ieee.org/document/8705842> | [5] Another DSL for Web Scraping. |
| <https://hackernoon.com/product-101-for-engineers-83ef7260cbf1> | [6] assessing product management |

Gantt Chart:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Week  Work Structure | Week1 | Week2 | Week3 | Week4 | Week5 | Week6 | Week7 | Week8 | Week9 | Week10 |
| Topic Discussion/Finalization. |  |  |  |  |  |  |  |  |  |  |
| Analysis and Design. |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |

Walk-Through:

**Scenario 1:**

The user opens his computer, he surfs online and he comes across a spectacle frame. He really likes it. Hence the user goes to lenskart and views the frame. After some time, the user thinks he wants to use the lenskart feature of recommendation. Lenskart recommends twenty different frames.

The user is mighty thrilled and wants to download the list of all the recommended frames. But the user is super lazy and he does not want to download the images manually as he finds it to be a burden. He wishes if there was an automated way of doing the required work.

Then he remembers there is a DSL that does the required work. He’s excited and he wants to use it.

Sadly, he is hesitant in using the DSL as he isn’t familiar with the programming language. But he gives it a try anyway.

When he opens the application, a pop-up arises and the user is asked if he wants to go through the documentation or if he wishes to go through a tutorial. If the user wishes to go ahead with the documentation, a new file with the documentation opens up and the user goes through. Else if he wishes to proceed with the video, he goes through a demo of what all the user could do and he also goes through a basic run through tutorial.

Now the user slowly starts understanding the syntax of the DSL and of its whereabouts. Once he comes to a fair understanding of how the DSL works, he is now ready to get on with the task of extracting data.

Case 1: After entering the commands, the user wishes to run the commands. But to his dismay, he gets an error. But along with the error he also gets a “help” option to help correct his mistake. The user is delighted and he looks at the “help” option being available. Now the user enters “help” and a new file is opened user is happy with the documentation and he with the help of command, is now able to run the code without errors.

Case 2: The user wishes to proceed with the tutorial. The user now is slowly guided by teaching the basic commands and its usages. The user is able to grasp the basic concepts enough to execute the basic commands. Now the user is able to get the results as required.

He now executes the command to get all the recommended images.

The process of extraction is successful but the user is confused as to where this extracted information is stored. He again reads the documentation and he will now have a clear understanding that he has to transfer all the contents of the file to a different file destination. He writes down the command to transfer the contents to the required destination.

He finds his extracted contents in the file.

He now wants finds the language to be very intriguing. So, he starts going through the language to find out what more it could do.

**Design and UX:**

Step 1: We would be focusing on a command line interface. When the user opens the application, he would find himself in front of a command line.

Step 2: The first time anyone opens a command line they would have to be briefed/ given a run through of the basic functionalities the user would be able to use.

Step 3: Let’s say the user has begun by entering the code and he wishes to run the code. The command line should have the following functions. It should have buttons such as: “file”, “run”, “help”.

Step 4: Since our users are primarily going to consist of naïve users, we have to lay options such as “get”, “view”, “select”, “filter” and “meta” on the menu bar. So, when the user clicks on one of these options, the basic syntax would be laid out for the person. Hence the complexity is dumbed down even further and the prompt statements laid when the user clicks the option, would be set to default operations.

The prompt option basically lays out the default syntax and all that the user has to do is to enter the missing parameters, where these values to be entered would originally be depicted in blank lines.

The **missing parameters** include “**URL**” and “**number\_of\_items**”, etc.

Step 5: Once the user has entered the missing parameters, he would be able to run the command.

Step 6: when the operation “run” is chosen, the output of the command entered previously such as “get” and “view” would be displayed. This displayed output would be stored in a temporary file/ a buffered file.

Step 7: The contents in the buffered file could be transferred to a particular file by simply specifying the command “to\_file” along with the consequent parameter being the file destination. This would transfer all the file contents of the temporary file to the destined file.

Step 8: This would arise another question. What if the user wants to scrape the contents of multiple different URL’s at the same time?

**Solution:** We could use the idea of creating sessions.

Working of a session:

Step 1) Initially when the CLI opens, there has to be a session created. (The creation of the session is displayed on the CLI to the user).

Step 2) The meaning of a session is simply that when the user creates a new session, he’s able to make changes to a file or directory

and while overwriting to a file/ making changes to a directory, the user would get a confirmation message if he wants to go ahead

with the operation or if he’d like to decline.

Step 3) The first time the CLI is opened, there is no necessity for us to create a session as it is already created by default. But we observe.

Step 4) The

**Scenario 1: The code being entered by the user.**

Points to remember: 1) The language is interpreted. 2) The DSL provides both file and IDE as a means to write code in.

We will look at 2 scenarios which talks about how a user will enter their code.

1. User uses the IDE to enter his/ her query.
2. User uses a file to enter his/ her code.

Scenario 1: The user enters the command into the IDE, line by line. After each line the command entered would be executed, like a regular IDE.

Scenario 2: The user writes code in a file. When the code is run since it’s an interpreted language, code is executed line by line. If a mistake arises during the process of execution, there is an error thrown immediately and stops.

**Scenario 2:** **Data fetching and storing of contents to the file.**

Points to remember: 1) A temporary file is used to store the contents being retrieved from the URL. 2) The contents being retrieved from the URL could be: a) Documents, b) Different types of files, c) Text, d) Images etc

Scenario 1: If the user attempts a pull data request from the URL, a temporary file will be created and the extracted data from the data pull will be stored in this file. This data goes and settles into the desired file by specifying the “to\_file” command. Similar to the “to\_file” command, the “to\_dir” is used to store the list of files under a single directory.

Else, even if user has to view data and he decides to transfer the metadata or the viewed data to the file, the temporary file can store the data on request by specifying the command “to\_file”. This will help in a better understanding of the data to be pulled and hence allows us to search what’s required as the contents are being documented.

**Scenario 3:**

Once the data is pulled, and when the user wants to store the data into a file and if the file already exists then, he should be warned before the content of the file being overwritten.

**Scenario 4: The usage of the “help” command.**

If the user is undergoing difficulty with entering the command along with its syntax, we will be able to provide them with a “help” command which would give a briefing of the commands in the DSL. i.e., a mini documentation.

While the user enters the “help” command, a separate file would open showing the list of commands along with their syntax. If the user is looking for a unique command, he could simply find it using “ctrl+f”.

Scenario 5: User is new to the language and he is exploring.

1. We should not complicate the syntax in order to simplify the operations for the user and get him used to the language.
2. We would have to

Executing the given code:

**Step 1**: The given code is entered in an IDE like interface/ a “.dsl ” formatted file.

Sessions can be added a bit later (For multiple activities at the same time).

1. DSL Code is written by the user.
2. He now runs the code.
3. Error detection and correctness of the code is to be computed.

**Step 2**: The given code written by the user is read by the python file that is capturing data.

1. The Python code now reads the file.
2. DSL Code has to be split and stored in a list.
3. We have to make a clear distinction in segregating a keyword and a parameter. Following which,
4. All “key value” pairs are to be identified and stored in a dictionary separately.
5. Consider the command: “add int 1 2”. Here the key words are “add” and “int” and parameters are [1,2], which should yield 3. For this purpose, we write the add() function for the “add” in another python file called module.py that has all modules/operations.
6. Parameters are to be identified: We have to parse the entire code written by the user, break the code into different tokens with the help of lexers by identifying the different keywords provided by the user. Each token is divided by identifying the keyword until parsing to the next keyword.
7. Once we have divided the entire code into different tokens, we now are able to get a comprised format of keywords and parameters.