Documentation for *Spider Class*:

Command to execute this file: scrapy crawl debate_crawler -o data.json

In this spider, named debate_crawler, we crawl, fetch the title, category, debate Id, and list of pro & con arguments from first five topics of debate.org's popular page. It's necessary to name the spider so that crawling can be done through this particular spider. The urls, common class names etc. are declared as constant variables for easier access. *start_urls* have the url of main page where we want to begin. And *TOPIC_LIMIT* consists of only 5 as we need to crawl through top 5 debates.

```
class DebateSpider(scrapy.Spider):
    name = "debate_crawler"
    start_urls = ["https://www.debate.org/opinions/?sort=popular"]

    # Constants
    TOPIC_LIMIT = 5
    BASE_URL = "https://www.debate.org"
    LOAD_MORE_ARGUMENTS = "https://www.debate.org/opinions/~services/opinions.asmx/GetDebateArgumentPage"
    LIST_DATA_HEADING_CLASS = "li.hasData h2"
    LIST_DATA_PARAGRAPH_CLASS = "li.hasData p"
```

It's necessary to remove the data.json file if it exists already so as it doesn't append the new data with the old data.

```
file_name = "data.json"
if os.path.exists(file_name):
    os.remove(file_name)
```

function parse:

The parse method processes the response of the first request, in this case the response of popular debates listing page of debate.org. From this page, we scrape of the links (<a href> tag) of top 5 popular debate topics using their css selectors. After each topic is scraped the *function crawl_each_debate_topic* is called where scraping of information of each topic is carried out.

```
def parse(self, response):
    debate_topics = response.css("a.a-image-contain::attr(href)").getall()[:self.TOPIC_LIMIT]
    debate_topics = [self.BASE_URL+ext for ext in debate_topics]
    return (scrapy.Request(url, callback=self.crawl_each_debate_topic) for url in debate_topics)
```

function crawl_each_debate_topic:

After scraping a popular topic, now crawling into that topic to get information like its title, category, id and pro-con arguments is accomplished. The *function extract_title_category_debateId* is called to extract the title, category, debate Id and is stored in *crawled_data* dictionary object. Also, to get list of pro and con arguments, each topic is crawled using *function crawl_pro_con_arguements_from_response*. The pro and con arguments are appended in the *crawled_data*. And to yield more data from next page the method *function load_more_arguments* is used with arguments *crawled_data* which contains all existing crawled data and the next page number, which is set to 2 by default (as first page arguments have already been crawled).

```
def crawl_each_debate_topic(self, response):
    crawled_data = self.extract_title_category_debateId(response)
    arguments_object = self.crawl_pro_con_arguements_from_response(response)
    crawled_data['pro_arguments'] = arguments_object['pro_arguments']
    crawled_data['con_arguments'] = arguments_object['con_arguments']
    yield from self.load_more_arguments(crawled_data, 2)
```

function extract_title_category_debateId:

This function crawls the title, category and debateId of a particular debate from the response. debateId is used later to fetch more arguments from the POST request.

```
def extract_title_category_debateId(self, response):
    debate_category = response.css("#breadcrumb a:nth-child(3)::text").get()
    debate_title = response.css("#col-wi span.q-title::text").get()
    debate_guid = response.css(".hasData::attr(did)").get()
    return {
        'topic': debate_title,
        'category': debate_category,
        'debateId': debate_guid,
    }
```

function crawl_pro_con_arguments_from_response:

Here in this method *function crawl_pro_con_arguments_from_response*, extraction of the title and contents of the argument is realised. It's checked if the argument is *yes* or *no* from the html file and then it gathers all the actual arguments using the *function collect_all_arguments.* The reason here we did not use *::text* attribute to read the argument text is because titles may be in simple form as *<h2>Title…</h2>* or as a part of *<h2><a href>Title…</h2>.* Hence, we read the entire html structure of <h2> and later remove the html tags.

```
def crawl_pro_con_arguements_from_response(self, response):
    id_list = ["#yes-arguments","#no-arguments"]
    for i in id_list:
        titles = response.css(i+" "+self.LIST_DATA_HEADING_CLASS).getall()
        contents = response.css(i+" "+self.LIST_DATA_PARAGRAPH_CLASS).getall()
        if '#yes' in i:
            pro_arguments = self.collect_all_arguments(titles, contents)
        else:
            con_arguments = self.collect_all_arguments(titles, contents)
    return {
        'pro_arguments': pro_arguments,
        'con_arguments': con_arguments
    }
```

function collect_all_arguments:

The *function collect_all_arguments*, returns a list of dictionary objects. Each object is an argument which was passed to the function from *function crawl_pro_con_arguments_from_response*. Before appending an argument, it is cleaned of html tags in *function remove_html_tags*.

```
def collect_all_arguments(self, titles_array, contents_array):
    return_array = []
    for i in range(len(titles_array)):
        return_array.append({
            'title': self.remove_html_tags(titles_array[i]),
            'body': self.remove_html_tags(contents_array[i])
        })
    return return_array
```

function remove_html_tags:

In this method removal of html tags is fulfilled and a clean string is returned. To do so we make use of regex. It helps to remove html tags and return a Unicode string. This is especially necessary for the titles of arguments in the debates.

```
def remove_html_tags(self, html_text):
    remover_regex = re.compile('<.*?>')
    html_free_text = re.sub(remover_regex, '', html_text)
    return html_free_text
```

function load_more_arguments:

After fetching first page, it's necessary to crawl through other pages. To do so this function is used to make a POST request to fetch more arguments from the server. We require to fetch header for POST request. It's done by using *function get_request_header.* For crawling through the new page, we defined the callback *function crawl_more_agruments*. The arguments *crawled_data* and *page_number* are passed to the callback function.

```
def load_more_arguments(self, crawled_data, page_number):
    headers = self.get_request_header()
    data = {
        "debateId": crawled_data["debateId"],
        "pageNumber": page_number,
        "itemsPerPage": 10,
        "ysort": 5,
        "nsort": 5
    }
    yield scrapy.http.Request(
        self.LOAD_MORE_ARGUMENTS,
        method="POST",
        body=json.dumps(data),
        headers= headers,
        callback=self.crawl_more_arguments,
        cb_kwargs=dict(crawled_data = crawled_data, page = page_number)
        )
```

function get_request_header:

Here, headers for *load_more_arguments* POST request is returned.

```
def get_request_header(self):
    return {
        "Content-Type": "application/json;charset=UTF-8",
        "Accept":"application/json, text/plain, */*"
        }
```

function crawl_more_arguments:

In this method, crawling of the response of more arguments is done and is appended to the existing arguments in *crawled_data*. The response of the POST request contains a "keyword" which separates the pro arguments, con arguments and also the status of whether current page is the last page. This keyword is *{ddo.split}*. If all data is been loaded then it returns *{finished}* and if data loading is pending it returns *{needmore}*. In *continue_flag* checks if we've reached the last page of the debate or not, According to the *continue_flag* we either yield the next page from *function load_more_arguments* (*continue_flag* is true) or yield *crawled_data* back as output (*continue_flag* is false).

```
def crawl_more_arguments(self, response, crawled_data, page):
    output = json.loads(response.text)
    output = output["d"]
    pro_section = Selector(text=output.split("{ddo.split}")[0])
    con_section = Selector(text=output.split("{ddo.split}")[1])
    continue_flag = False if 'finished' in output.split("{ddo.split}")[2] else True

    # Extract pro and con arguments from new response
    pro_arguments = self.collect_all_arguments(
        pro_section.css(self.LIST_DATA_HEADING_CLASS).getall(),
        pro_section.css(self.LIST_DATA_PARAGRAPH_CLASS).getall()
    )
    con_arguments = self.collect_all_arguments(
        con_section.css(self.LIST_DATA_HEADING_CLASS).getall(),
        con_section.css(self.LIST_DATA_PARAGRAPH_CLASS).getall()
    )

    # Append new arguments and existing arguments
    crawled_data["pro_arguments"] = crawled_data["pro_arguments"] + pro_arguments
    crawled_data["con_arguments"] = crawled_data["con_arguments"] + con_arguments

    # Finished not found in response, get more arguments from POST request
    if continue_flag:
        yield from self.load_more_arguments(crawled_data, page+1)
    # Finished found in response, return entire crawled data
    else:
        yield crawled_data
```

Documentation for *visualisation of crawled data* file:

Command to execute this file: jupyter notebook debate_data_plotting.ipynb

The code starts with loading the “data.json” file using the “json” Python module. The loaded data is then saved in the variable “debate_data”. The type of this variable is “dict”.

```
#load the json file from given path
with open("data.json", "r") as f:
    debate_data = json.load(f)
```

Then we have initialised some empty arrays to store the number of argu–ments, number of words, etc. found in the data.

The data crawled by the crawler is store in a list in the json file. The number of topics in the “data.json” is extracted using the len() method of python on the “debate_data” variable and stored in “debate_data_len”.

Now, we extract the data from each topic by putting it inside a loop.

- First the lists of pro and con arguments are fetched and store in “pro_arguments” and “con_arguments” variables. Then the category, to which this topic belongs to, and the title of the topic are appended to “category_list” and “topic_list” lists respectively.

```
#loop for processing each topic in debate_data
for x in range(debate_data_len):
    # fetch the list of pro and con arguments and store it
    pro_arguments = debate_data[x]['pro_arguments']
    con_arguments = debate_data[x]['con_arguments']

    # store the category and topic name of each topic in lists
    category_list.append(debate_data[x]['category'])
    topic_list.append(debate_data[x]['topic'])
```

- Then, we have calculated the length of “pro_arguments” and “con_arguments” lists and stored it in “no_of_pro_args” and “no_of_con_args” variables.
- Now, we processed each pro argument in the “pro_arguments” list by extracting the body of the argument and calculating the number of words in it. This number is then stored in “pro_words” variable and appended to the “words_arr” list.

```
# loop for processing each pro argument
for y in range(no_of_pro_args):
    pro_words = 0                                   # initialize pro_words variable to zero
    pro_arg_body = pro_arguments[y]['body']         # fetch the argument body and store it in "pro_arg_body"
    pro_words = len(pro_arg_body.split())           # calculate the no. of words in the fetched body.
    words_arr.append(pro_words)                     # append the no. of pro words found to words_arr array
```

- Similarly, each con argument is fetched and processed, and the number of words of the body is stored in “con_words” variable and appended to the “words_arr” list.

```
# loop for processing each con argument
for z in range(no_of_con_args):
    con_words = 0                          # initialize con_words variable to zero
    con_arg_body = con_arguments[z]['body']  # fetch the argument body and store it in "con_arg_body"
    con_words = len(con_arg_body.split())  # calculate the no. of words in the fetched body.
    words_arr.append(pro_words)            # append the no. of con words found to words_arr array
```

- Finally, the total number of arguments is calculated by adding the number of pro_arguments and con_arguments and is then appended to “total_args_arr” list.

After exiting the loop, we processed the “category_list” list. First, the duplicate elements are removed using the unique() of pandas library. Then the length of the newly generated unique list is calculated using the len() method.

Now, we process each category to find the number of arguments for each category by using loop. The number of arguments found for each category is appended to “total_args_by_category_arr” list.

```
# loop for processing each category
for p in range(category_list_len):
    no_of_args = 0                         # initialize the number of arguments to zero
    for x in range(debate_data_len):       # looping each topic in the debate_data
        if category_list[p] == debate_data[x]['category']: # check if the category is equal to the category of the topic in the loop
            no_of_args = no_of_args + total_args_arr[x] # add the number of args for that category and stored it in "no_of_args"
    total_args_by_category_arr.append(no_of_args)          # store the no_of_args by appending it to "total_args_by_category_arr" list
```

Plotting graphs:

- Use the plot from mathplotlib and divide the plot into subplots of 1 column and 3 rows and assign it to 3 axes.
- In the first subplot, we have plotted the histogram of *number of words per argument*. The x-axis gives the length of arguments (number of words, without tokenization) and the y-axis denotes the number of arguments.
- For the second subplot, we have plotted a bar graph of number of arguments per category. Here the x-axis represents the category and the y-axis denotes the number of arguments.
- Finally for the third subplot, we have plotted a bar graph of number of arguments per topic. Here the x-axis represents each topic and the y-axis denotes the number of arguments.

```
#Plotting Graphs
# Divide the plot into 3 subplots
f, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(8,23))

# Set the labels of x-axis to rotate to 90 degrees
ax2.tick_params(labelrotation=90)
ax3.tick_params(labelrotation=90)

# Plot the histogram for number of words
ax1.hist(words_arr)

#Plot the bar graphs for number of arguments for category and topic respectively
ax2.bar(category_list,total_args_by_category_arr)
ax3.bar(topic_list,total_args_arr)

# Setting the title of each subplots
ax1.set_title('No. of Words')
ax2.set_title('No. of Arguments per category')
ax3.set_title('No. of Arguments per topic')

# Setting axis label for each subplot
ax1.set_xlabel('Length of Arguments (as number of words)')
ax1.set_ylabel('No. of Arguments')

ax2.set_xlabel('Category')
ax2.set_ylabel('No. of Arguments')

ax3.set_xlabel('Topics')
ax3.set_ylabel('No. of Arguments')

plt.show()
```