Вместо введения

Итоговый алгоритм был придуман не с первого раза. Первоначальная идея, сходу возникшая при прочтении ТЗ – это искать элемент(ы) HTML, в котором(ых), в отличии от других элементов, текста больше всего. Но беглый просмотр исходных кодов различных новостных сайтов дал понять, что такой подход работать не будет. Во-первых, потому, что текст статьи сильно фрагментирован, во-вторых, у некоторых рекламных блоков текста больше, чем размер абзаца в статье.

Второй вариант, который пришел на ум – это использовать исключительно теги параграфа <p>. Беглый просмотр кода некоторых новостных сайтах подтвердил мою теорию и было решено начать с этого. Я находил все теги <p> и теги заголовков <h1>-<h6>, собирал их в список, а потом срезал этот список так, чтобы он начинался с тега <h1>, т.е. отсеивая параграфы, которые идут выше заголовка, т.к. они являлись мусорными. При первых испытаниях выяснилось 3 ситуации, которые рушили мои представления об идеально спарсенной статье: во-первых, после статьи тянется много других тегов <p>, никак не относящиеся к ней, т.е. «мусорные». Во-вторых, нашлись новостные сайты, которые совсем не используют тег <p> по назначению, вместо него применяют <div>. Т.е. такие новостные статьи просто не попадали в файл. В-третьих, были и такие информационные сайты, где на использовании тега <p> дело не заканчивается, и к нему подключаются теги <div>. Таким образом частично терялся текст статьи. Такой алгоритм тоже не подходил.

Так появился третий вариант, текущий. Первоначальная идея состояла в том, чтобы найти один единственный тег («главный тег»), в котором заключен весь текст статьи с заголовком. Такой тег был найден у каждого новостного сайта, среди которых я проводил тестирование. С ним было бы гораздо проще работать, т.к. в нем точно не было бы практически ничего лишнего. Но как показали дальнейшие тесты – много мусора может содержаться даже в таком варианте. Однако, если после работы этого алгоритма запросить только теги <p> - работало намного лучше. Только, как говорилось выше, на такой расчет полагаться не стоит, т.к. есть сайты, которые не используют тег <p> совсем или вперемешку с <div>. Тогда решено было доработать алгоритм еще одним вспомогательным алгоритмом. Его цель заключается в том, чтобы из всего дерева «главного тега» найти только те теги, в которых содержится текст. После чего пройтись еще раз по полученным тегам и совершить фильтрацию уже по тексту. Так и было сделано и результаты оказались удовлетворительными.

Структура проекта

Весь проект поделен на 3 части (3 класса):

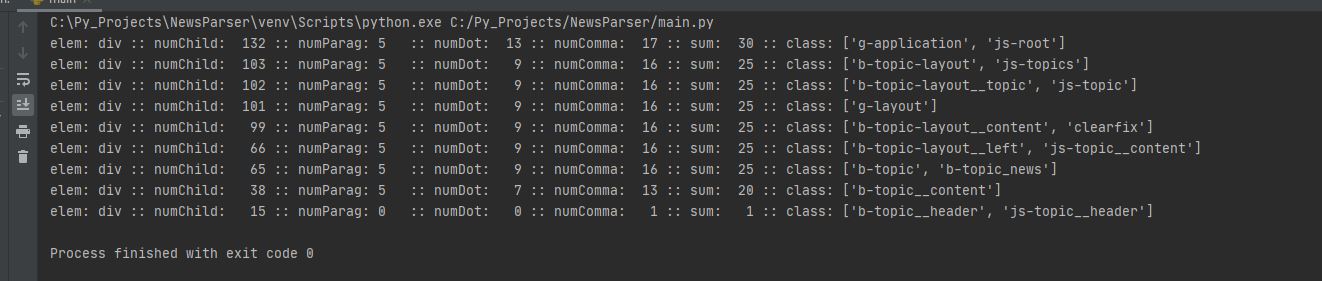
* **DataSource** (класс **FromUrlDataSource**) – класс, отвечающий за получение данных. В данном проекте реализовано получение данных из сети, по URL.
* **Core** (класс **BS4ParseAlgorithm**) – класс, выполняющий основную задачу проекта – парсинг статей.
* **Output** (класс **FileOutput**) – класс, который отвечает за вывод информации. В данном проекте, в соответствии с ТЗ, это текстовый файл.

Алгоритм парсинга

**Первый этап** – найти один «главный» тег, в котором заключена вся статья и заголовок к ней с минимальным количеством мусора. Общий принцип такой: получить список основных тегов, в которых с максимальной вероятностью располагается полный текст статьи и заголовок. У каждого такого тега подсчитать кол-во дочерних элементов, имеющих текст, определить наличие тега <h1> и подсчитать кол-во точек в тексте дочерних тегов. Точки повышают шанс того, что данный текст может быть текстом статьи. На основании этих данных находится самый ближайший к статье тег и выдается за результат работы данного этапа.

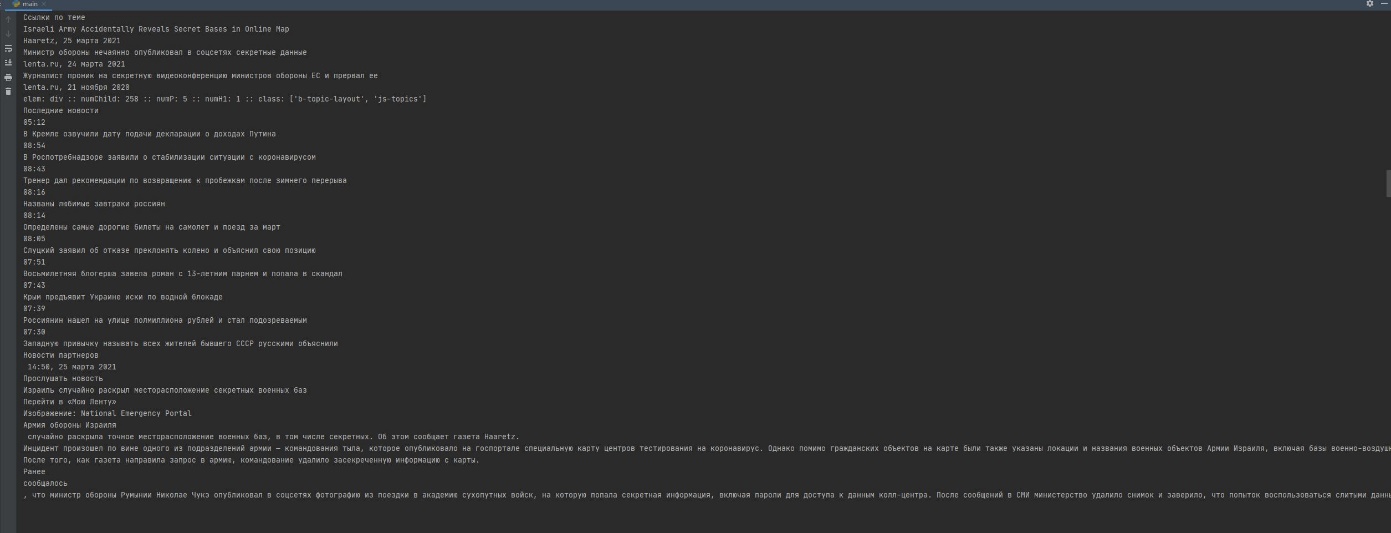
Шаги:

1. Первоначально были определены основные теги – <div> и <article>, в одном из которых 100% можно было получить искомое.
2. Производится поиск и получение всех таких тегов в html-файле. Результаты помещаются в список **list\_all\_text\_tags**.
3. Теперь необходимо рекурсивно пройтись по дочерним элементам каждого такого тега, чтобы подсчитать необходимые параметры: итоговое кол-во дочерних элементов, кол-во точек и наличие заголовка <h1>. В результате получается список **list\_probable\_main\_tag**, который состоит только из тех тегов, у которых в дочерних элементах присутствует тег <h1>, а также прикрепляется информация о кол-ве дочерних элементов и кол-ве точек.



1. Из этого списка необходимо выбрать какой-то один тег, с которым будем работать дальше. Чем меньше кол-во дочерних элементов, тем ближе мы к самой статье. Но выбрать самую минимальную позицию тоже не получится и скриншот это доказывает. Этот div является контейнером для заголовка, сама статья расположена в другом месте. Для того, чтобы правильно определить нужный div из этого списка, используется подсчет среднего кол-ва точек всех элементов и проверка на отклонение не больше 50%. За этот выбор отвечает функция **find\_main\_tag\_with\_article(list\_tag)**. Ее результатом выполнения и станет главный тег, в котором находится весь текст статьи и ее заголовок.

**Второй этап** – получить красиво оформленный текст статьи. Несмотря на то, что в предыдущем этапе был получен тег, максимально близко расположенный в дереве элементов к тексту статьи, в нем все еще много мусора:



Общий принцип таков: из главного тега достать определенные дочерние теги, в которых есть текст, и они не относятся к рекламным блокам. Далее пройтись регулярным выражением по полученному тексту, отформатировать его исходя из задания и выдать за результат.

Шаги:

1. Поиск элементов, которые соответствуют следующему списку условий:
   * Элемент должен быть тегом (у библиотеки bs4 есть 2 основных типа элементов: Tag и NavigableString)
   * Этот тег должен относится к тегам из белого списка **tag\_white\_list**
   * Его дочерним элементом должен быть **NavigableString** и он не должен быть равен ‘\n’ (пустой) ИЛИ дочерний элемент является тегом из списка тегов для форматирования текста **text\_formatter\_tag\_list**
   * И родительскими тегами этого элемента не должны быть теги из черного списка **tag\_black\_list**.

За это составное условие отвечает функция **find\_all\_tag\_with\_text**, которая передается как параметр в метод bs4 **findAll**. На выходе мы получаем список тегов **list\_tag\_with\_text**, в которых содержится текст.

1. Теперь необходимо отфильтровать теги по их вложенному тексту, а заодно и отформатировать этот текст. За это отвечает функция **filtering\_and\_formatting\_text**. Эта функция вызывается в цикле и на вход принимает один элемент. В зависимости от того, что это за элемент происходит форматирование текста.
   * Если текущий элемент относится к тегу <code> - то он форматируется согласно функции **wrapper\_code\_tag**.
   * Если текущий тег – заголовок <h1>-<h4> - то он форматируется согласно заголовкам – переводится в верхний регистр.
   * Все остальные теги проходят через регулярное выражение, которое позволяет иметь любые символы, но обязательным правилом является то, что в строке должна быть как минимум 1 буквы из русского или английского алфавита; строка должна заканчиваться на любой из символов: [.,!?:;]. Если строка прошла валидацию, то ее обрабатывает стандартная библиотека **textwrap**, которая задает ширину текста согласно файлу конфигурации (по умолчанию 80 символов).

Результат функции **filtering\_and\_formatting\_text** конкатенируется в итоговую результирующую строку и возвращается как результат работы всего класса.

Дальнейшее улучшение программы

Несмотря на то, что программа закрывает основные требования ТЗ, есть некоторое количество задач, которые можно было бы доделать, а именно:

1. Некоторые сайты содержат защиту от парсинга. Для обхода можно использовать другие библиотеки вместо request (например, Grab).
2. У некоторых сайтов используется аннотация в скрытом режиме. Эта аннотация содержит в себе половину текста самой статьи (несколько слепленных параграфов), и данная программа этот текст также добавляет в файл. Необходимо ее доработать так, чтобы исключить это дублирование.
3. На некоторых сайтах программа подхватывает вместе со статьей несколько безобидных, но лишних фраз. Например: «Нашли ошибку на сайте?», «Выделите нужный фрагмент текста (не более 3000 символов) и нажмите».
4. Чтобы программа действительно приносила пользу, необходимо переделать вывод из текстового файла в файл-word или в html. Таким образом появилась бы возможность изменять шрифты, размер текста, а в случае с html еще и фон. А также можно было бы лучше разграничивать заголовки и кодовые вставки из тега <code>.
5. Доработать форматирование: определять, являются ли теги <li> нумерованными или нет и реализовать вложенность списков.
6. Повысить отказоустойчивость программы после многочисленных тестов и выявления слабых мест.