Практическое задание по Beautiful Soup

Часть 1:

В этом задании вам необходимо реализовать парсер для сбора статистики со страниц Википедии. Чтобы упростить вашу задачу, необходимые страницы уже скачаны и сохранены на файловой системе в директории **wiki/** (Например, страница https://en.wikipedia.org/wiki/Stone_Age сохранена файле **wiki/Stone_Age**). Архив страниц вы можете скачать ниже: *soup_sample.zip*

Парсер реализован в виде функции **parse**, которая принимает на вход один параметр: **path_to_file** — путь до файла, содержащий html код страницы википедии. Гарантируется, что такой путь существует. Ваша задача — прочитать файл, пройтись **Beautiful Soup** по статье, найти её тело (это *<**div id="bodyContent">**)* и внутри него подсчитать:

1. Количество картинок (img) с шириной (width) не меньше 200. Например: <img width="200">, но не <img> и не <img width="199">
2. Количество заголовков (h1, h2, h3, h4, h5, h6), первая буква текста внутри которых соответствует *заглавной* букве E, T или C. Например: <h1>**E**nd</h1> или <h5>**C**ontents</h5>, но не <h1>About</h1> и не <h2>end</h2> и не <h3>1End</h3>
3. Длину максимальной последовательности ссылок, между которыми нет других тегов, открывающихся или закрывающихся. Например:
   <a>, **<a>**, **<a>** - тут **2** ссылки подряд, т.к. закрывающийся span прерывает последовательность.
   <a>, **<a>**, **<a>** - а тут **3** ссылки подряд, т.к. span находится внутри ссылки, а не между ссылками.
4. Количество списков (ul, ol), не вложенных в другие списки.
   Например: **
**
, **
** - два не вложенных списка (и один вложенный)

Результатом работы функции parse будет **список четырех чисел**, посчитанных по формулам выше. В качестве шаблона для вашего решения **используйте следующий код**:

```
from bs4 import BeautifulSoup
import unittest

def parse(path_to_file):
# Поместите ваш код здесь.
# ВАЖНО!!!
# При открытии файла, добавьте в функцию open необязательный параметр
# encoding='utf-8', его отсутствие в коде будет вызвать падение вашего
# решения на грейдере с ошибкой UnicodeDecodeError
return [imgs, headers, linkslen, lists]

class TestParse(unittest.TestCase):
def test_parse(self):
      test_cases = (
            ('wiki/Stone_Age', [13, 10, 12, 40]),
            ('wiki/Brain', [19, 5, 25, 11]),
            ('wiki/Artificial_intelligence', [8, 19, 13, 198]),
            ('wiki/Python_(programming_language)', [2, 5, 17, 41]),
            ('wiki/Spectrogram', [1, 2, 4, 7]),)

      for path, expected in test_cases:
            with self.subTest(path=path, expected=expected):
            self.assertEqual(parse(path), expected)

if __name__ == '__main__':
unittest.main()
```

В пункте про последовательность ссылок вы можете ошибиться с результатом, если решите использовать метод find_next(). Обратите внимание, что хотя find_next находит тег, идущий сразу за текущим, этот тег может оказаться вложенным в текущий, а не быть его следующим соседом. Возможно, нужно использовать другой метод или алгоритм.

Так же, не упустите момент, что данные во всех пунктах нужно искать внутри * ,* а не по всей странице.

Пример работы функции parse:

```
>>> parse("wiki/Stone_Age")

>>> [13, 10, 12, 40]
```

Несколько других примеров работы вы можете посмотреть в тестах из шаблона кода выше.

Во время проверки на сервере будут доступны только стандартные модули и bs4, сеть не доступна. Ваше решение будет проверяться, как на наборе страниц из приложенного архива, так и на дополнительном наборе страниц википедии.

"Важное замечание! Не используйте для сбора статистики по странице регулярные выражения. Beautiful Soup в своей работе использует специализированные парсеры, которые позволяют корректно обрабатывать невалидные (содержащие ошибки, например: незакрытый тег) html страницы. Статистика, собранная с помощью модуля re, на таких страницах будет возвращать не верное значение."

Часть 2:

В этом задании продолжаем работать со страницами из wikipedia. Необходимо реализовать механизм сбора статистики по нескольким страницам. Сложность задачи состоит в том, что сначала нужно будет определить страницы, с которых необходимо собирать статистику. В качестве входных данных служат названия двух статей(страниц). Гарантируется, что файлы обеих статей есть в папке **wiki** и из первой статьи можно попасть в последнюю, переходя по ссылкам только на те статьи, копии которых есть в папке **wiki**.

Например, на вход подаются страницы: **Stone_Age** и **Python_(programming_language)**. В статье **Stone_Age** есть ссылка на **Brain**, в ней на **Artificial_intelligence**, а в ней на **Python_(programming_language)** и это кратчайший путь от **Stone_Age** до **Python_(programming_language)**. Ваша задача — **найти самый короткий путь** (гарантируется, что существует только один путь минимальной длины), а затем с помощью функции **parse** из предыдущего задания собрать статистику по всем статьям в найденном пути.

Результат нужно вернуть в виде словаря, ключами которого являются имена статей, а значениями списки со статистикой. Для нашего примера правильный результат будет :

```
{ 'Stone_Age': [13, 10, 12, 40],
  'Brain': [19, 5, 25, 11],
  'Artificial_intelligence': [8, 19, 13, 198],
  'Python_(programming_language)': [2, 5, 17, 41]
}
```

Вам необходимо реализовать две функции: **build_bridge** и **get_statistics**.

```
def build_bridge(path, start_page, end_page):
    """возвращает список страниц, по которым можно перейти по ссылкам со start_page на
    end_page, начальная и конечная страницы включаются в результирующий список"""

    # напишите вашу реализацию логики по вычисления кратчайшего пути здесь


def get_statistics(path, start_page, end_page):
    """собирает статистику со страниц, возвращает словарь, где ключ - название страницы,
    значение - список со статистикой страницы"""

    # получаем список страниц, с которых необходимо собрать статистику
    pages = build_bridge(path, start_page, end_page)
    # напишите вашу реализацию логики по сбору статистики здесь

    return statistic
```

Обе функции принимают на вход три параметра:**path** - путь до директории с сохраненными файлами из wikipedia,**start_page** - название начальной страницы,**end_page** - название конечной страницы.

Функция **build_bridge** вычисляет кратчайший путь и возвращает список страниц в том порядке, в котором происходят переходы. Начальная и конечная страницы включаются в результирующий список. В случае, если название стартовой и конечной страницы совпадают, то результирующий список должен содержать только стартовую страницу. Получить все ссылки на странице можно разными способами, в том числе и с помощью регулярных выражений, например так:

```
with open(os.path.join(path, page), encoding="utf-8") as file:
    links = re.findall(r"(?<=/wiki/)[\w()]+", file.read())
```

Обратите внимание, что что на страницах wikipedia могут встречаться ссылки на страницы, которых нет в директории wiki, такие ссылки должны игнорироваться.

Пример работы функции **build_bridge**:

```
>>> result = build_bridge('wiki/', 'The_New_York_Times', 'Stone_Age')

>>> print(result)
['The_New_York_Times', 'London', 'Woolwich', 'Iron_Age', 'Stone_Age']
```

Функция **get_statistics** использует функцию **parse** и собирает статистику по страницам, найденным с помощью функции **build_bridge**. Пример работы функции **get_statistics**:

```
>>> from pprint import pprint
>>> result = get_statistics('wiki/', 'The_New_York_Times', "Binyamina_train_stati
on_suicide_bombing")
>>> pprint(result)
{'Binyamina_train_station_suicide_bombing': [1, 3, 6, 21],
 'Haifa_bus_16_suicide_bombing': [1, 4, 15, 23],
 'Second_Intifada': [9, 13, 14, 84],
 'The_New_York_Times': [5, 9, 8, 42]}
```

Вы можете использовать для проверки вашего решения тесты:

```python
# Набор тестов для проверки студентами решений по заданию "Практическое задание
# по Beautiful Soup - 2". По умолчанию файл с решением называется solution.py,
# измените в импорте название модуля solution, если файл с решением имеет другое
имя.

import unittest

from solution import build_bridge, get_statistics

STATISTICS = {
    'Artificial_intelligence': [8, 19, 13, 198],
    'Binyamina_train_station_suicide_bombing': [1, 3, 6, 21],
    'Brain': [19, 5, 25, 11],
    'Haifa_bus_16_suicide_bombing': [1, 4, 15, 23],
    'Hidamari_no_Ki': [1, 5, 5, 35],
    'IBM': [13, 3, 21, 33],
    'Iron_Age': [4, 8, 15, 22],
    'London': [53, 16, 31, 125],
    'Mei_Kurokawa': [1, 1, 2, 7],
    'PlayStation_3': [13, 5, 14, 148],
    'Python_(programming_language)': [2, 5, 17, 41],
    'Second_Intifada': [9, 13, 14, 84],
    'Stone_Age': [13, 10, 12, 40],
    'The_New_York_Times': [5, 9, 8, 42],
    'Wild_Arms_(video_game)': [3, 3, 10, 27],
    'Woolwich': [15, 9, 19, 38]}

TESTCASES = (
    ('wiki/', 'Stone_Age', 'Python_(programming_language)',
     ['Stone_Age', 'Brain', 'Artificial_intelligence', 'Python_(programming_langu
age)']),

    ('wiki/', 'The_New_York_Times', 'Stone_Age',
     ['The_New_York_Times', 'London', 'Woolwich', 'Iron_Age', 'Stone_Age']),

    ('wiki/', 'Artificial_intelligence', 'Mei_Kurokawa',
     ['Artificial_intelligence', 'IBM', 'PlayStation_3', 'Wild_Arms_(video_game)'
,
      'Hidamari_no_Ki', 'Mei_Kurokawa']),

    ('wiki/', 'The_New_York_Times', "Binyamina_train_station_suicide_bombing",
     ['The_New_York_Times', 'Second_Intifada', 'Haifa_bus_16_suicide_bombing',
      'Binyamina_train_station_suicide_bombing']),

    ('wiki/', 'Stone_Age', 'Stone_Age',
     ['Stone_Age', ]),
)

class TestBuildBrige(unittest.TestCase):
    def test_build_bridge(self):
        for path, start_page, end_page, expected in TESTCASES:
```

```
52              with self.subTest(path=path,
53                                start_page=start_page,
54                                end_page=end_page,
55                                expected=expected):
56                  result = build_bridge(path, start_page, end_page)
57                  self.assertEqual(result, expected)
58
59  class TestGetStatistics(unittest.TestCase):
60      def test_build_bridge(self):
61          for path, start_page, end_page, expected in TESTCASES:
62              with self.subTest(path=path,
63                                start_page=start_page,
64                                end_page=end_page,
65                                expected=expected):
66                  result = get_statistics(path, start_page, end_page)
67                  self.assertEqual(result, {page: STATISTICS[page] for page in expe
68  cted})
69
70  if __name__ == '__main__':
71      unittest.main()
```

Ваше решение должно содержать реализацию функций **get_statistics**, **build_bridge** и **parse**. Вы можете дополнительно объявить в коде другие функции, если этого требует логика вашего решения.

Как и в предыдущем задании, тестовая система будет проверять ваш код как на страницах приложенных к описанию, так и на другом наборе станиц wikipedia. Решение должно выполняться за приемлемое время. Мы понимаем, что рассмотрение алгоритма решения не входит в программу курса, поэтому предлагаем вам ознакомиться с этой темой самостоятельно.

Полезные ссылки:

Реализация графов и деревьев на Python

Реализации алгоритмов/Поиск в ширину

Частые вопросы, возникающие при решении данного задания.

(прикреплено, обновляемая ветка)

Q1. Не понимаю, как найти самый короткий путь. Хоть и знаком с поиском в ширину, в данном случае совсем не представляю как реализовать поиск самого короткого пути.

A1. Возможно вам поможет эта статья.

Q2. В сообщении об ошибке говорится о странице **608_(number)**. В приложенном к описанию архиву такой страницы нет.

A2. В грейдере проверка вашего решения проводится, как на страницах из приложенного к описанию архива, так и на собственном наборе страниц. Сделано это для защиты от решений, которые используют различные «хаки» для того чтобы «обойти» грейдер. Указание страницы в описании приведено скорее для преподавателей, чтобы легче было локализовать вашу ошибку. При корректной работе вашего решения на предоставленных страницах, проблем с прохождением проверок не должно возникать.

Q3. В задании требуется найти количество заголовков (h1, h2, h3, h4, h5, h6), первая буква текста внутри которых соответствует заглавной букве E, T или C. У вас в тесте стоит цифра 10 - ('wiki/Stone_Age', [13, 10, 12, 40]), но заголовков на этой странице 11. Ошибка в тестах?

A3. Нет, все правильно. Из описания задания - ***«Ваша задача — прочитать файл, пройтись Beautiful Soup по статье, найти её тело (это) и внутри него подсчитать: ...»***. Искать нужно не во всём документе, а только внутри контейнера с ** .**

Q4. локальное тестирование и грейдер выдают одну и туже ошибку -

"Total tests: 8. Tests failed: 2, Errors: 0. Total time: 10.811. Failed test - test_2[Spectrogram-expected4]. E AssertionError: Тест 2.5. Вызов функции parse('/grader/gErsy/tests/wiki/Spectrogram') возвращает не верное значение: [1, 2, 5, 7]. Ожидалось: [1, 2, 4, 7]."

Но это какая-то аномалия, мы ищем в тэге ** **, я специально вывел его в лог и убедился что он правильный. Но та последовательность которая даёт у меня 5 она находиться уже ЗА ПРЕДЕЛОМ этого тега. Я ищу с помощью **find_next** и **find_next_siblings**. Как они за пределами тэга ищут? По моей логике такого не должно быть. Или я чего-то не понимаю, или это баг, или у меня в коде ошибка которую я в упор не вижу.

A4. Использование метода **find_next**, ищет ссылки по всему дереву и в этом случае выходит за пределы **bodyContent**, проваливаясь в **footer**. Попробуйте изменить логику нахождения - сначала найти все ссылки с помощью **find_all**, а затем у каждой найденной ссылки посчитать количество соседних ссылок с помощью **find_next_siblings**.

Q5. Я, к сожалению, не могу понять даже первоначальный алгоритм "путешествия" по страницам: 1.Нам нужно перебираться именно по ссылкам? (в таком случае, если на каждой странице есть указатели на следующую, то зачем нам папка wiki? 2. Или же если нам нужно просматривать именно файлы из папки, то непонятно, каким образом это должно реализовываться.

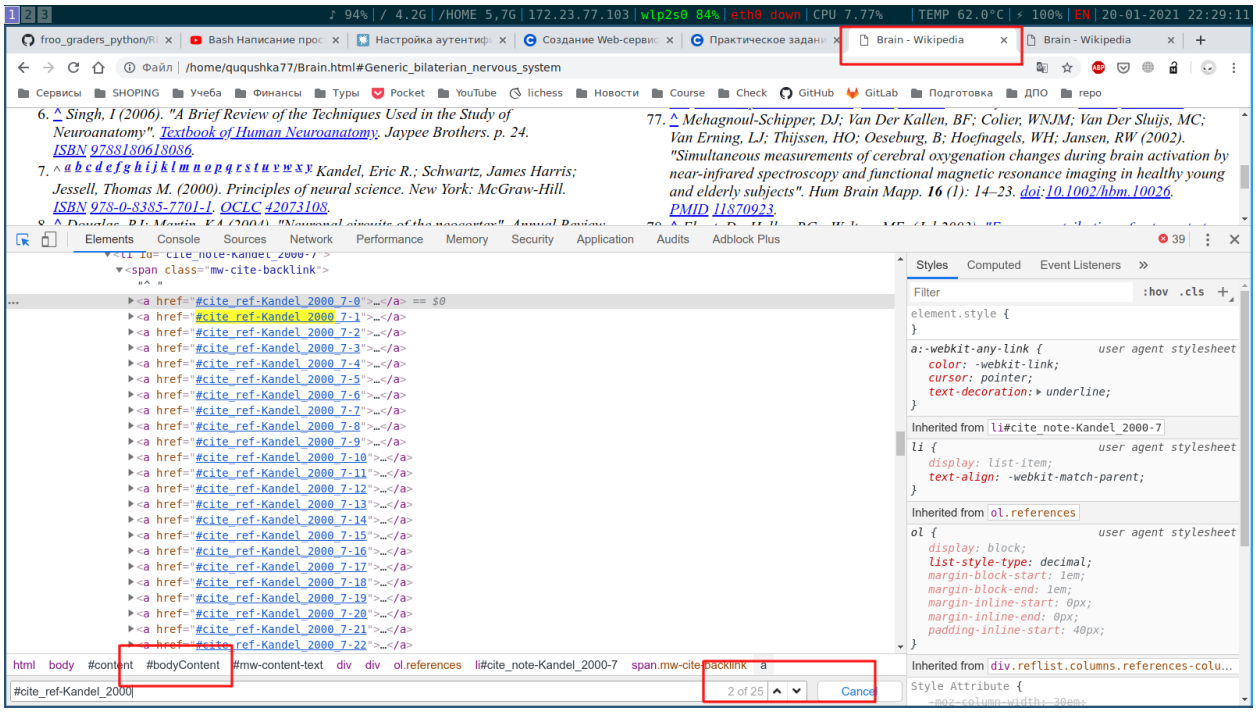
A5. 1. Из описания задания - ***"В примере к этому заданию находится папка wiki, в ней лежит около 500 страниц из английской Википедии. Например, страница https://en.wikipedia.org/wiki/Stone_Age лежит в wiki/Stone_Age."*** - Вы должны работать именно с локальной копией wikipedia, которая сохранена в директории wiki. Если вы откроете любой файл в этой директории, то можете увидеть, что ссылки на другие страницы в ней есть ни что иное, как ссылки на локальные файлы из wiki (по существу пути к файлам).

```
77 </tr>
78 <tr>
79 <th scope="row">Sister channel(s)</th>
80 <td><a href="/wiki/BBC_One" title="BBC One">BBC One</a><br />
81 <a href="/wiki/BBC_Two" title="BBC Two">BBC Two</a><br />
82 <a href="/wiki/BBC_News_(TV_channel)" title="BBC News (TV channel)">BBC News</a><br />
83 <a href="/wiki/BBC_Parliament" title="BBC Parliament">BBC Parliament</a><br />
84 <a href="/wiki/CBBC_(TV_channel)" title="CBBC (TV channel)">CBBC</a><br />
85 <a href="/wiki/CBeebies" title="CBeebies">CBeebies</a></td>
86 </tr>
87 <tr>
```

2. Не совсем ясно, что вас смущает, тот же браузер может легко открывать страницы, как из сети, так и сохраненные на файловой системе. Вы просто читаете данные страницы из файла из директории wiki, вместо получения их из сети, далее используйте Beautiful Soup для парсинга страницы и получения необходимых для решения задания данных.

Q6. Считаю 3-ий пункт задания, а именно: "Длину максимальной последовательности ссылок, между которыми нет других тегов". Для двух приведенных тестов у меня цифра сходится. Для трёх других - нет. Убился. Уже вручную считаю - у меня меньше получается. Например, просчитал вручную по файлу Brain - в тесте должно быть 25, а у меня - 13. Программа тоже 13 выдаёт. Ну не могу я найти в этом файле, даже вручную, 25 подряд ссылок, не прерванных другими тэгами. Как быть то? 3-ий день бьюсь...

A6. В указанном вами файле есть последовательность из 25 ссылок.



Мы настоятельно **НЕ РЕКОМЕНДУЕМ** заниматься поиском и "ручным" подсчетом элементов на страницах. Данное занятие приводит только к "пустой" трате времени. Потратьте его лучше на изучение документации и эксперименты с кодом.