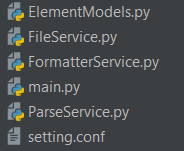
Задача Python – Mini readability

Выполнил: Соболев Александр Андреевич

Решение задачи было принято разбить на несколько файлов.



* ElementModels.py – Модели элементов html
* FileService.py – Чтение и запись текстовых файлов
* FormatterService.py – Форматирование текста по заданным условиям. На данный момент это форматирование длины строки
* main.py – запускаемый файл
* ParseService.py – Парсер html страницы
* setting.conf – конфиг настроек

Рассмотри алгоритм работы программы. Начнем с файла main.py:

import re  
import sys  
from FileService import FileManager  
from ParseService import ParserHTML  
  
  
def is\_valid\_url(url):  
 regex = re.compile(  
 r'^https?://'   
 r'(?:(?:[A-Z0-9](?:[A-Z0-9-]{0,61}[A-Z0-9])?\.)+[A-Z]{2,6}\.?|'   
 r'localhost|'   
 r'\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})'   
 r'(?::\d+)?'   
 r'(?:/?|[/?]\S+)$', re.IGNORECASE)  
 return url is not None and regex.search(url)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 url = sys.argv[1]  
 if is\_valid\_url(url):  
 file\_service = FileManager()  
 parser = ParserHTML(url, file\_service.config["ParserHTML"])  
 parser.parse\_page()  
 file\_service.save\_text\_page(url, parser.get\_text())  
 print("Ok")  
 else:  
 print("Incorrect url")

Функция is\_valid\_url проверяет с помощью регулярного выражение валидность url. Если url не валидная, то выводим сообщение «Incorrect url» и завершаем работу. Иначе создаем FileService и ParserService, настраиваем их, парсим текст и сохраняем в файле, путь к которому задан url.

Давайте посмотрим, как работает FileService:

import os  
import configparser  
  
from FormatterService import TextFormatter  
  
  
class FileManager:  
 def \_\_init\_\_(self):  
 self.config = configparser.ConfigParser()  
 self.config.read(os.path.abspath("setting.conf"))  
 self.main\_dir = self.config["Main"]["save\_folder"]  
 TextFormatter.width = int(self.config["TextFormatter"]["width"])  
  
 def save\_text\_page(self, url, text):  
 basedir = self.main\_dir + os.path.dirname(url[url.find("/") + 1:])  
 if not os.path.exists(basedir):  
 os.makedirs(basedir)  
 file\_path = basedir + "/index.txt"  
 if os.path.exists(file\_path):  
 os.remove(file\_path)  
 with open(file\_path, 'w') as fp:  
 new\_text = TextFormatter.formate\_text(text)  
 fp.write(new\_text)  
 fp.close()  
  
 def get\_conf(self, service\_name):  
 return self.main\_dir[service\_name]

Он представлен классом FileManager. При инициализации мы читаем конфиг и сразу настраиваем корневую папку для сохранений(main\_dir) и ширину для форматирования.

Метод get\_conf необходим для чтения конфига определенного сервиса.

Метод save\_text\_page. Он принимает url и непосредственно сам текст. Мы пытаемся найти путь до места сохранения, базируясь по url. Если такого пути не существует, создаем его. Далее проверяем существует ли файл в папке, если да, то удаляем. Следом создаем новый файл. Форматируем текст с помощью метода TextFormatter.formate\_text(text), записываем его в файл и закрываем его.

Рассмотрим FormatterService.

class TextFormatter:  
 width = 80  
  
 @staticmethod  
 def formate\_text(text):  
 result\_text = ""  
 for line in text.split("\n\n"):  
 if len(line) >= TextFormatter.width:  
 result\_text += TextFormatter.formate\_big\_line(line)  
 else:  
 result\_text += line  
 result\_text += "\n\n"  
 return result\_text  
  
 @staticmethod  
 def formate\_big\_line(big\_line):  
 res\_line = ""  
 len\_line = 0  
 for word in big\_line.split():  
 if len\_line + len(word) > TextFormatter.width:  
 res\_line += "\n"  
 len\_line = 0  
 res\_line += word + " "  
 len\_line += len(word) + 1  
 return res\_line[: len(res\_line) - 1]

он представлен в виде атрибута класса “width”(ширина строки) и двух статичных методов: formate\_text и formate\_big\_line.

Метод formate\_text. Мы разбиваем строку по «Большим отступам» (по условиям ТЗ между заголовками, абзацами пустая строка) и проверяем каждую линию. Если она больше width, то вызываем второй статичный метод formate\_big\_line, а иначе просто записываем в результат. Полученный результат возвращаем.

Метод formate\_big\_line. Принимает линию, длина которой больше width. Мы разбивает её на слова и циклом соединяем, проверяя, будет ли длина больше width при добавлении нового слова, если это так, по добавляем перенос.

Рассмотрим ElementModels

import re  
  
  
class Element:  
 total = 0  
 is\_hard\_struct = False  
  
 def \_\_init\_\_(self, tag, text, href=""):  
 self.tag = tag  
 self.text = self.clear\_text(text)  
 if href != "":  
 self.text += f"[{href}]"  
 Element.total += 1  
  
 @staticmethod  
 def clear\_text(text):  
 result = text.replace("\n", "").replace("\t", "")  
 spec\_symbols = re.findall(r"&\w+;", text)  
 for spec\_symbol in spec\_symbols:  
 result.replace(spec\_symbol,"")  
 return result  
  
  
class IndentElement(Element):  
 def \_\_init\_\_(self, tag):  
 super(IndentElement, self).\_\_init\_\_(tag, "")  
 self.text = "\n\n"

В нем реализовано два класса Element и IndentElement.

При инициализации объекта класса Element нужно указать тег элемента html и его текст. Также если есть ссылка, то передаем и ещё и добавляем в конце текста в квадратных скобках. При добавлении текста мы «чистим» его от табуляций, переносов и спец символов.

IndentElement это пустая строка между абзацами, заголовками.

И наконец рассмотрим ParseService

import re  
import requests  
from ElementModels import Element, IndentElement  
  
  
def set\_conf(config):  
 ParserHTML.tags = [x.strip() for x in eval(config["tags"])]  
 ParserHTML.skip\_tags = [x.strip() for x in eval(config["skip\_tags"])]  
  
  
class ParserHTML:  
 tags = []  
 skip\_tags = []  
  
 def \_\_init\_\_(self, url, config=""):  
 self.page = requests.get(url)  
 self.url\_data = re.search('(?P<scheme>http.\*://)?(?P<host>[^:/ ]+).?(?P<port>[0-9]\*).\*', url)  
 self.cur\_pos = 0  
 self.data = []  
 if config != "":  
 set\_conf(config)  
  
 def parse\_page(self):  
 self.cur\_pos = self.page.text.find("body")  
 main\_tag = ""  
 count\_item\_save = 0  
 href = ""  
 tag = ""  
  
 while self.cur\_pos != len(self.page.text) and self.cur\_pos != -1:  
 end\_block = self.page.text.find(">", self.cur\_pos)  
 element\_config = self.page.text[self.cur\_pos + 1:end\_block]  
 start\_new\_block = self.page.text.find("<", end\_block)  
 if tag in ParserHTML.tags and self.page.text.find(f"</{tag}>",  
 end\_block) - start\_new\_block > 1 and main\_tag == "":  
 Element.is\_hard\_struct = True  
 main\_tag = tag  
 count\_item\_save = Element.total  
 tag = element\_config.split()[0]  
 if "/" + main\_tag == tag:  
 Element.is\_hard\_struct = False  
 main\_tag = ""  
 if Element.total - count\_item\_save != 0:  
 self.data.append(IndentElement(tag))  
 if tag in ParserHTML.skip\_tags or any(x in element\_config[len(tag):] for x in ParserHTML.skip\_tags):  
 self.cur\_pos = self.page.text.find(f"</{tag}>", end\_block)  
 tag = ""  
 continue  
 elif tag == "a":  
 href = self.get\_href()  
 if start\_new\_block - end\_block > 1 and tag != "":  
 self.data.append(Element(tag, self.page.text[end\_block + 1:start\_new\_block], href=href))  
 href = ""  
 self.cur\_pos = start\_new\_block  
  
 def get\_text(self):  
 return "".join([data.text for data in self.data])  
  
 def get\_href(self):  
 href\_start = self.page.text.find("href=", self.cur\_pos) + 6  
 href\_end = self.page.text.find("\"", href\_start)  
 href = self.page.text[href\_start:href\_end]  
 if href[:4] != "http":  
 href = self.url\_data.group('scheme') + self.url\_data.group('host') + self.url\_data.group(  
 'port') + href  
 return href

Тк просили не использовать библиотеки, в которые решают это задачу, то я реализовал свой парсер.

Метод set\_conf настраивает теги для чтения и теги для скипа (У которые мы сразу ищем конец).

Класс ParserHTML при инициализации считывает страницу с помощью request.get(), дальше с помощью регулярного выражения разбиваю url на scheme, host и port. Это понадобится для получения корректных ссылок при переходе внутри сайта.

Метод parse\_page. Парсить мы начинаем с первого появления body. Запускаем циклы, пока текущая позиция меньше длины строки и не равна -1. Каждую итерацию мы ищем конец тега, его тип и начало следующего. Расстояние между концом и началом больше 1 и это тег, в котором может быть текст, мы говорим, что это сложная структура, даже если там будет простой текст. Это поможет нам правильно парсить сложные структуры. Например, состоящие из абзаца и ссылок внутри них, корректно обрабатывать подчеркивания, курсивы и другое. Дальше мы проверяем, а не конец ли это главного тега, если это так отключаем поле главного тега и если внутри было больше 1 элемента, то добавляем пустую строку. Если нам попался тег, который нужно пропустить, ищем его конец и продолжаем оттуда. Если это ссылка запускаем метод get\_href(). И наконец, если между концом и началом есть текст, создаем новый элемент и сохраняем его в массив.

Метод get\_href(). Ищем ссылку и смотрим куда она ведет. Если на этот же сайт, то в начало добавляем путь.

Метод get\_text(). Соединяет элементы в строку и возвращает ещё.

setting.conf

[Main]  
save\_folder = D:/Project  
[ParserHTML]  
tags = ["title", "p", "b", "a", "li", "h1", "h2", "h3", "h4", "h5", "span", "blockquote"]  
skip\_tags = ["head", "nav", "menu", "header", "foot", "script", "button", "popup", "tool", "time", "style"]  
[TextFormatter]  
width = 80

Примеры результатов:

1. <https://lenta.ru/tags/geo/rt/>
2. <https://lenta.ru/news/2023/04/27/hlor/>
3. <https://lenta.ru/>

К сожалению, данная программа не работает на всех сайтах. В целях улучшения это адаптация под разные сайты, лучше бы использовать библиотеки для парсинга (Например BS).