Методы сбора и обработки данных при помощи Python

Scrapy



# На этом уроке

1. Узнаем, что такое открытые данные, для чего они нужны и как используются.
2. Научимся работать с CSV в Python и самостоятельно их создавать.

# Оглавление

Введение

Scrapy

Установка

Создание поискового робота

Сбор данных с hh.ru

Сбор дафотографийнных и фотографий с Avito.ru

Глоссарий

Домашнее задание

Используемая литература

# Введение

Web scraping — это техника сбора данных о страницах и занесения их в базу поисковой системы.

Поисковые роботы позволяют извлечь текстовые и количественные данные, фотографии с сайта без официального API и многое другое.

Scraping состоит из двух этапов:

* систематический поиск и загрузка веб-страниц;
* извлечение данных с веб-страниц.

Создать поискового робота с нуля можно с помощью различных модулей и библиотек, которые предоставляет язык программирования, однако по мере роста программы это может вызвать ряд проблем. Например, вам понадобится переформатировать извлечённые данные в CSV, XML или JSON. Существуют сайты, для работы с которыми необходимы специальные настройки и модели доступа. Наилучший вариант — изначально создать поискового робота на основе библиотеки, которая устраняет эти потенциальные проблемы. Для этого будем использовать Python и Scrapy.

# Scrapy

Scrapy — одна из наиболее популярных и производительных библиотек Python для получения данных с веб-страниц. Она включает в себя большинство общих функциональных возможностей. Вам не придётся самостоятельно прописывать многие функции. Scrapy позволяет быстро и без труда создать «веб-паука».

## Установка

Пакет Scrapy можно найти в PyPI (Python Package Index, также известен как pip) — поддерживаемом сообществом репозитории для всех вышедших пакетов Python.

| pip install scrapy |
| --- |

После установки создадим проект scrapy внутри нашего проекта

| scrapy startproject brickset-scraper . |
| --- |

Создастся новый каталог с шаблонной структурой внутри.

## Создание поискового робота

Создадим файл для поискового робота по имени scraper.py. В нём будет храниться весь код «паука». Введём в терминал:

| Scrapy genspider BrickSet brickset.com |
| --- |

Также возможно создать файл с помощью текстового редактора внутри директории spiders.

Внутри будет базовый код робота, который будет основан на библиотеке Scrapy. Класс BrickSetSpider является наследником scrapy.Spider — базового класса для поисковых роботов, предоставленный Scrapy. Он имеет два обязательных атрибута:

* name — название «паука»;
* start\_urls — список ссылок на страницы, которые нужно проанализировать.

| import scrapy  class BrickSetSpider(scrapy.Spider):  name = "brickset\_spider"  start\_urls = ['http://brickset.com/sets/year-2016'] |
| --- |

Рассмотрим этот код подробнее:

Первая строка импортирует Scrapy, что позволяет использовать доступные классы библиотеки.

| import scrapy |
| --- |

Следующая строка добавляет класс Spider из библиотеки Scrapy и создаёт подкласс BrickSetSpider. Подкласс — это более узкий, специализированный вариант родительского класса. Класс Spider предоставляет методы для отслеживания URL и извлечения данных с веб-страниц, но он не знает, где искать страницы и какие именно данные нужно извлечь. Для передачи классу недостающих данных мы создали подкласс.

| class BrickSetSpider(scrapy.Spider): |
| --- |

Зададим имя поискового робота:

| name = "brickset\_spider" |
| --- |

Следующая строка — это ссылка на страницу, которую нужно просканировать.

| start\_urls = ['http://brickset.com/sets/year-2016'] |
| --- |

Обычно файлы Python запускаются с помощью команды path/to/file.py. Однако Scrapy предоставляет собственный интерфейс командной строки, чтобы оптимизировать процесс запуска «паука». Для этого используем следующую команду:

| scrapy runspider scraper.py |
| --- |

Команда вернёт:

| 2016-09-22 23:37:45 [scrapy] INFO: Scrapy 1.1.2 started (bot: scrapybot) 2016-09-22 23:37:45 [scrapy] INFO: Overridden settings: {} 2016-09-22 23:37:45 [scrapy] INFO: Enabled extensions: ['scrapy.extensions.logstats.LogStats', 'scrapy.extensions.telnet.TelnetConsole', 'scrapy.extensions.corestats.CoreStats'] 2016-09-22 23:37:45 [scrapy] INFO: Enabled downloader middlewares: ['scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware', ... 'scrapy.downloadermiddlewares.stats.DownloaderStats'] 2016-09-22 23:37:45 [scrapy] INFO: Enabled spider middlewares: ['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware', ... 'scrapy.spidermiddlewares.depth.DepthMiddleware'] 2016-09-22 23:37:45 [scrapy] INFO: Enabled item pipelines: [] 2016-09-22 23:37:45 [scrapy] INFO: Spider opened 2016-09-22 23:37:45 [scrapy] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0 items/min) 2016-09-22 23:37:45 [scrapy] DEBUG: Telnet console listening on 127.0.0.1:6023 2016-09-22 23:37:47 [scrapy] DEBUG: Crawled (200) <GET http://brickset.com/sets/year-2016> (referer: None) 2016-09-22 23:37:47 [scrapy] INFO: Closing spider (finished) 2016-09-22 23:37:47 [scrapy] INFO: Dumping Scrapy stats: {'downloader/request\_bytes': 224, 'downloader/request\_count': 1, ... 'scheduler/enqueued/memory': 1, 'start\_time': datetime.datetime(2016, 9, 23, 6, 37, 45, 995167)} 2016-09-22 23:37:47 [scrapy] INFO: Spider closed (finished) |
| --- |

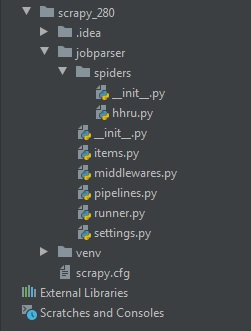
Сначала робот инициализирует и загружает дополнительные компоненты и расширения, необходимые ему для обработки считываемых данных. Затем он использует URL, указанный в start\_urls, и загружает HTML (как это делает браузер). После этого робот передаёт HTML методу parse.

# Cбор данных с hh.ru

Сделаем проект по сбору данных на hh.ru. Наша задача:

* перейти по всем страницам, выданным по запросу;
* внутри каждой страницы перейти по всем вакансиям;
* внутри каждой вакансии собрать данные по наименованию вакансии и зарплате.

Структура программы:



Паук создаётся через командную строку: scrapy genspider hhru hh.ru, где hhru — название паука (оно может быть любым), а hh.ru — домен, в рамках которого он будет работать. Программа создаст файл hhru.py. В нём изначально будет класс HhruSpider с определёнными атрибутами.

Настройки нашего паука находятся в файле settings.py.

Модули, указывающие на место хранения пауков:

| SPIDER\_MODULES = ['jobparser.spiders'] NEWSPIDER\_MODULE = 'jobparser.spiders' |
| --- |

Информация о user\_agent, предоставляемая браузером:

| USER\_AGENT = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130 Safari/537.36' |
| --- |

В ROBOTSTXT указана структура данных, лежащих на сайте. В большинстве случаев этот файл закрыт для доступа, поэтому ставим false.

| ROBOTSTXT\_OBEY = False |
| --- |

Добавим параметр LOG\_ENABLED = True. Теперь мы увидим всё, что происходит внутри scrapy. Так как он будет выдавать очень много информации, поставим ограничение LOG\_LEVEL = DEBUG. Возможны варианты INFO и ERROR. Для записи логов можно добавить LOG\_FILE = 'log.txt'

| LOG\_ENABLED = True LOG\_LEVEL = 'DEBUG' *#INFO ERROR* |
| --- |

Общий вид hhru.py:

| *# -\*- coding: utf-8 -\*-* import scrapy from scrapy.http import HtmlResponse from jobparser.items import JobparserItem   class HhruSpider(scrapy.Spider):  name = 'hhru'  allowed\_domains = ['hh.ru']  start\_urls = ['https://izhevsk.hh.ru/search/vacancy?area=&st=searchVacancy&text=python']   def parse(self, response: HtmlResponse):  next\_page = response.css('a.HH-Pager-Controls-Next::attr(href)').extract\_first()  yield response.follow(next\_page, callback=self.parse)   vacansy = response.css(  'div.vacancy-serp div.vacancy-serp-item div.vacancy-serp-item\_\_row\_header a.bloko-link::attr(href)'  ).extract()   for link in vacansy:  yield response.follow(link, callback=self.vacansy\_parse)   def vacansy\_parse(self, response: HtmlResponse):  name = response.css('div.vacancy-title h1.header::text').extract\_first()  salary = response.css('div.vacancy-title p.vacancy-salary::text').extract()  *# print(name, salary)*  yield JobparserItem(name=name,salary=salary) |
| --- |

Сначала укажем более детальный start\_urls для get-запроса:

| start\_urls = ['https://izhevsk.hh.ru/search/vacancy?area=&st=searchVacancy&text=python'] |
| --- |

Чтобы были доступны различные методы для response, сделаем импорт:

| from scrapy.http import HtmlResponse |
| --- |

Для передачи в дальнейшем информации в items.py сделаем импорт:

| from jobparser.items import JobparserItem |
| --- |

Так как мы хотим перейти по всем страницам поисковой выдачи, нужно найти кнопку, которая будет отвечать за переход. В next\_page сохраняем результат выполнения метода от response, вызываем метод css, где указываем один из классов кнопки «дальше» (HH-Pager-Controls-Next) и атрибут href. Мы получим указатель на элемент, после чего нужно его извлечь, для чего дописываем extract\_first().

| next\_page = response.css('a.HH-Pager-Controls-Next::attr(href)').extract\_first() |
| --- |

Теперь нужно разделить работу нашего паука на две составляющие: один поток будет работать на сбор всех страниц с ссылками на вакансии, а второй передаст управление дальше по программе, чтобы собрать информацию с полученной страницы. Для этого используем yield. Он сохраняет состояние на том месте, где остановился. Будет вызван метод follow относительно response и произойдет разветвление работы программы. Одна часть возвращается обратно внутрь метода parse, но в response попадает результат get запроса по next\_page.

| yield response.follow(next\_page, callback=self.parse) |
| --- |

Далее получаем информацию о вакансиях, которые присутствуют на страницах. В response разбираем первую страницу. Нам необходимо получить ссылки на все вакансии на ней. Для этого указываем в css селекторе контейнеры div до тега «а» с классом bloko-link, атрибут href и извлекаем ссылку extract().

| vacansy = response.css(  'div.vacancy-serp div.vacancy-serp-item div.vacancy-serp-item\_\_row\_header a.bloko-link::attr(href)'  ).extract() |
| --- |

Нам остается перейти по этим ссылкам. Для этого указываем:

| for link in vacansy:  yield response.follow(link, callback=self.vacansy\_parse) |
| --- |

Мы делаем get-запрос на ссылку и передаём управление в метод, который будет заниматься сбором информации внутри страницы с вакансией. Для этого создадим метод:

| def vacansy\_parse(self, response: HtmlResponse): |
| --- |

В нём укажем наименование вакансии name и зарплату salary:

| name = response.css('div.vacancy-title h1.header::text').extract\_first()  salary = response.css('div.vacancy-title p.vacancy-salary::text').extract() |
| --- |

Теперь собранную информацию передадим в items.py:

| yield JobparserItem(name=name,salary=salary) |
| --- |

Общий вид items.py:

| import scrapy   class JobparserItem(scrapy.Item):  *# define the fields for your item here like:*  \_id = scrapy.Field()  name = scrapy.Field()  salary = scrapy.Field()   pass |
| --- |

В items.py прописывается структура под собранные пауком данные. Есть свойство, для которого мы указываем тип, это будет scrapy.Field(). Нам не нужно задумываться о том, какого типа данные придут. Необходимо только указать, какие поля будут поступать. В нашем случае это будут:

| name = scrapy.Field() salary = scrapy.Field() |
| --- |

Для работы базы данных пропишем:

| \_id = scrapy.Field() |
| --- |

Из items.py оформленная структура попадает в pipeline.py, где происходит финальная обработка данных. Здесь для неё указываются все методы, классы и функции, формируется конечный документ с нужными нам полями. Мы его либо возвращаем, либо складываем в базу даннего либо возвращаем, либо складываем в базу даннего либо возвращаем, либо складываем в базу даннего либо возвращаем, либо складываем в базу даннего либо возвращаем, либо складываем в базу данных.

Общий вид pipeline.py:

| from pymongo import MongoClient  class JobparserPipeline(object):  def \_\_init\_\_(self):  client = MongoClient('localhost',27017)  self.mongobase = client.vacansy\_280   def process\_item(self, item, spider):  collection = self.mongobase[spider.name]  collection.insert\_one(item)  print(item['salary'])   return item |
| --- |

Для начала добавим базу данных:

| from pymongo import MongoClient |
| --- |

Настроим подключение к ней:

| def \_\_init\_\_(self):  client = MongoClient('localhost',27017)  self.mongobase = client.vacansy\_280 |
| --- |

В pipeline.py указывается не только объект, но и паук, от которого он поступил. Это полезная функция, так как на практике обычно используется несколько пауков одновременно:

| def process\_item(self, item, spider): |
| --- |

Благодаря тому, что известно, от какого паука приходят данные, мы можем раскладывать их по коллекциям:

| collection = self.mongobase[spider.name] |
| --- |

Укажем, что именно будем складывать в коллекцию:

| collection.insert\_one(item) |
| --- |

Добавим метод для корректного отображения salary в базе данных:

| print(item['salary']) |
| --- |

Для запуска всего проекта необходим файл runner.py.

Общий вид runner.py:

| from scrapy.crawler import CrawlerProcess from scrapy.settings import Settings  **from jobparser import settings** from jobparser.spiders.hhru import HhruSpider *# from jobparser.spiders.hhru import SjruSpider*   if \_\_name\_\_ == '\_\_main\_\_':  crawler\_settings = Settings()  crawler\_settings.setmodule(settings)  process = CrawlerProcess(settings=crawler\_settings)  process.crawl(HhruSpider)  *# process.crawl(SjruSpider)*  process.start() |
| --- |

Для начала импортируем те модули, которые помогут связать между собой составляющие проекта.

Scrapy.crawler отвечает за сами процессы:

| from scrapy.crawler import CrawlerProcess |
| --- |

Scrapy.settings подключает общие настройки:

| from scrapy.settings import Settings |
| --- |

Jobparser подключает именно наши настройки:

| from jobparser import settings |
| --- |

Jobparser.spiders.hhru подключает нашего паука:

| from jobparser.spiders.hhru import HhruSpider |
| --- |

Теперь создадим экземпляр настроек:

| crawler\_settings = Settings() |
| --- |

Привяжем к нему наши текущие настройки паука:

| process = CrawlerProcess(settings=crawler\_settings) |
| --- |

Создадим процесс, содержащий в себе привязку наших настроек:

| process = CrawlerProcess(settings=crawler\_settings) |
| --- |

Сделаем вызов метода crawl, в качестве значения указав нашего паука HhruSpider.

| process.crawl(HhruSpider) |
| --- |

В конце прописываем запуск процесса:

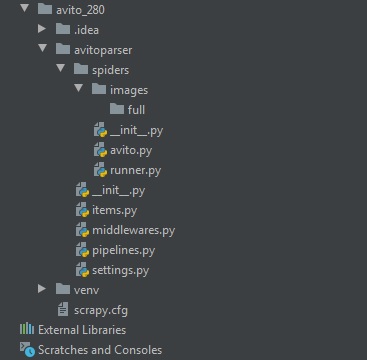
| process.start() |
| --- |

# Сбор данных и фотографий с Avito.ru

Для примера осуществим поиск на avito.ru в разделе «бытовая электроника» по ключевому слову asus. Будем извлекать название лота, цену и фотографии продаваемой вещи. Перед началом работы нужно установить pillow для работы с изображениями:

| scrapy genspider example example.com |
| --- |

Структура программы:



Создадим паука аналогично предыдущему проекту: scrapy genspider avito avito.ru.

Настроим нашего паука в файле settings.py:

| ROBOTSTXT\_OBEY = False LOG\_ENABLED = True LOG\_LEVEL = 'DEBUG' USER\_AGENT = 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130 Safari/537.36' |
| --- |

В связи с особенностями работы Avito.ru, уменьшим частоту и количество запросов:

| CONCURRENT\_REQUESTS = 16 DOWNLOAD\_DELAY = 3  CONCURRENT\_REQUESTS\_PER\_DOMAIN = 8 CONCURRENT\_REQUESTS\_PER\_IP = 8 |
| --- |

Для работы с фотографиями укажем директорию, куда будут попадать фото:

| IMAGES\_STORE = 'images' |
| --- |

Расставим приоритет для классов в pipelines.py (чем меньше цифра, тем выше приоритет):

| ITEM\_PIPELINES = {  'avitoparser.pipelines.DataBasePipeline': 300,  'avitoparser.pipelines.AvitoPhotosPipeline': 200, } |
| --- |

Общий вид avito.py:

| import scrapy from scrapy.http import HtmlResponse from avitoparser.items import AvitoparserItem from scrapy.loader import ItemLoader   class AvitoSpider(scrapy.Spider):  name = 'avito'  allowed\_domains = ['avito.ru']   def \_\_init\_\_(self, mark):  self.start\_urls = [f'https://www.avito.ru/rossiya/bytovaya\_elektronika?q={mark}']   def parse(self, response: HtmlResponse):  ads\_links = response.xpath('//a[@class="snippet-link"]/@href').extract()  for link in ads\_links:  yield response.follow(link, callback=self.parse\_ads)   def parse\_ads(self, response: HtmlResponse):  name = response.css ('h1.title-info-title span.title-info-title-text::text').extract.first()  photos = response.xpath(//div[contains(@class, "gallery-img-wrapper")]//div[contains(@class, "gallery-img-frame")]/@data-url').extract()  price = response.xpath(//span[@class="js-item-price"][1]  /text()').extract()   yield AvitoparserItem(name=name, photos=photos,price=price)  print(name, photos, price) |
| --- |

Создадим метод \_init\_, перенесём в него параметр start\_urls и добавим ключевое слово для поиска:

| def \_\_init\_\_(self, mark):  self.start\_urls = [f'https://www.avito.ru/rossiya/bytovaya\_elektronika?q={mark}'] |
| --- |

В методе parse найдём ссылки на все наши объявления, находящиеся на странице:

| def parse(self, response: HtmlResponse):  ads\_links = response.xpath('//a[@class="snippet-link"]/@href').extract() |
| --- |

Разделим работу нашего паука на две части, как в предыдущем примере:

| for link in ads\_links:  yield response.follow(link, callback=self.parse\_ads) |
| --- |

Общий вид items.py

| import scrapy  class AvitoparserItem(scrapy.Item):  *# define the fields for your item here like:*  \_id = scrapy.Field()  name = scrapy.Field()  photos = scrapy.Field()  price = scrapy.Field()  pass |
| --- |

Файл будет выглядеть как в предыдущем прим ере, только добавим строку:

| photos = scrapy.Field() |
| --- |

Общий вид pipelines.py:

| import scrapy from scrapy.pipelines.images import ImagesPipeline from pymongo import MongoClient  class DataBasePipeline(object):  def \_\_init\_\_(self):  client = MongoClient('localhost',27017)  self.mongo\_base = client.avito\_photo  def process\_item(self, item, spider):  collection = self.mongo\_base[spider.name]  collection.insert\_one(item)  return item  class AvitoPhotosPipeline(ImagesPipeline):  def get\_media\_requests(self, item, info):  if item['photos']:  for img in item['photos']:  try:  yield scrapy.Request(f'http:{img}')  except Exception as e:  print(e)   def item\_completed(self, results, item, info):  if results:  item['photos'] = [itm[1] for itm in results if itm[0]]  return item |
| --- |

Для работы с фотографиями импортируем:

| from scrapy.pipelines.images import ImagesPipeline |
| --- |

Подключим базу данных, как в предыдущем примере:

| class DataBasePipeline(object):  def \_\_init\_\_(self):  client = MongoClient('localhost',27017)  self.mongo\_base = client.avito\_photo  def process\_item(self, item, spider):  collection = self.mongo\_base[spider.name]  collection.insert\_one(item)  return item |
| --- |

Создадим ещё один класс, в который фотографии будут попадать раньше, чем в DataBasePipeline:

| class AvitoPhotosPipeline(ImagesPipeline): |
| --- |

Внутри него определим метод:

| def get\_media\_requests(self, item, info): |
| --- |

Бывают объявления, у которых нет фото. Чтобы программа не прекращала работу с ошибкой, пропишем (если фото нет, программа будет выводить «е»):

| if item['photos']:  for img in item['photos']:  try:  yield scrapy.Request(img)  except Exception as e:  print(e) |
| --- |

Нам недостаточно просто скачать фотографию, нужно информацию о ней записать в наш текущий item, чтобы потом не запутаться. Добавим ещё один метод, который позволит переопределить значение photos. Изначально у нас это был просто список из ссылок на фотографии. Теперь, когда мы берём первый элемент из каждого result, если нулевой элемент равен true, у нас будет не только ссылка, но и путь внутри директории проекта, и checksum:

| def item\_completed(self, results, item, info):  if results:  item['photos'] = [itm[1] for itm in results if itm[0]]  return item |
| --- |

Наш изменённый item передается дальше в класс DataBasePipeline.

Общий вид runner.py:

| from scrapy.crawler import CrawlerProcess from scrapy.settings import Settings  from avitoparser.spiders.avito import AvitoSpider from avitoparser import settings  if \_\_name\_\_ == '\_\_main\_\_':  crawler\_settings = Settings()  crawler\_settings.setmodule(settings)  process = CrawlerProcess(settings=crawler\_settings)  process.crawl(AvitoSpider, mark='asus')  process.start() |
| --- |

Он выглядит так же, как в предыдущем примере, но мы добавляем строку, в которой указываем два параметра — сам паук и ключевое слово для поиска:

| process.crawl(AvitoSpider, mark='asus') |
| --- |

# Глоссарий

**Web scraping** — это техника сбора данных о страницах и занесения их в базу поисковой системы.

**Scrapy** — одна из наиболее популярных и производительных библиотек Python для получения данных с веб-страниц, которая включает в себя большинство общих функциональных возможностей.

**Селектор** — это шаблон, который позволяет найти элементы страницы, содержащие необходимые данные.

# Дополнительные материалы

1. [Собираем данные с помощью Scrapy](https://habr.com/ru/post/115710/).
2. [Scrapy — простой скрапинг сайтов](https://python-scripts.com/scrapy-example).
3. [Статья по CSS селекторам](https://msiter.ru/tutorials/css-nachalnogo-urovnya/selektory-svoistva-znacheniya)
4. [Дополнительные возможности CSS селекторов](https://naikom.ru/blog/archives/2306)
5. [Видео по функциям-генераторам и yield](https://yadi.sk/i/N_PYcvcdyWc9PQ)

# Домашнее задание

1. Доработать паука в имеющемся проекте, чтобы он складывал все записи в БД (любую) и формировал item по структуре:

* наименование вакансии;
* зарплата от;
* зарплата до;
* ссылка на саму вакансию;
* сайт, откуда собрана вакансия.

1. Создать в имеющемся проекте второго паука по сбору вакансий с сайта superjob. Паук должен формировать item по аналогичной структуре и складывать данные в БД.
2. Взять любую категорию товаров на сайте [Леруа Мерлен](https://leroymerlin.ru/catalogue/). Собрать с использованием ItemLoader следующие данные:

* название;
* все фото;
* параметры товара в объявлении.

1. С использованием output\_processor и input\_processor реализовать очистку и преобразование данных. Цены должны быть в виде числового значения.
2. \*Написать универсальный обработчик параметров объявлений, который будет формировать данные вне зависимости от их типа и количества.
3. \*Реализовать более удобную структуру для хранения скачиваемых фотографий.

Дополнительно:

Перевести всех пауков сбора данных о вакансиях на ItemLoader и привести к единой структуре. Сайт можно взять и любой другой (интернет-магазин или [сайт с объявлениями](https://apipost.ru/rating-dosok-obyavleniy.php) отсюда). Главное, чтобы по get-запросу вам возвращалась нужная информация.

# Используемая литература

1. [Документация Scrapy](https://docs.scrapy.org/en/xpath-tutorial/index.html).

2. [Работа с файлами и фото на Scrapy](https://docs.scrapy.org/en/latest/topics/media-pipeline.html).

3. [Web scraping с помощью Scrapy и Python](https://www.8host.com/blog/web-scraping-s-pomoshhyu-scrapy-i-python-3/).