# Парсинг: requests-html, Scrapy | Я.Шпора

## Библиотека requests-html для парсинга динамических страниц

Установить библиотеку:

```
pip install requests-html==0.10.0
```

Для работы с библиотекой используются сессии. Чтобы получить код, который подгружается динамически, следует задать небольшую паузу между запросом к странице и анализом её кода.

Для установки такой паузы применяется метод `response.html.render()`, в него передаётся именованный аргумент `sleep`; значение этого аргумента — длительность задержки в секундах. Необходимую длительность задержки нужно вычислить экспериментально.

```
from bs4 import BeautifulSoup
from requests_html import HTMLSession

# Адрес веб-сайта для парсинга.
PARSING_URL = 'https://httpbin.org/'
# Значение задержки в секундах.
TIMEOUT = 3


if __name__ == '__main__':
        # Создать сессию. Через этот объект выполняются HTTP-запро
    session = HTMLSession()
    response = session.get(PARSING_URL)
        # Выждать три секунды.
        response.html.render(sleep=TIMEOUT)
    soup = BeautifulSoup(response.html.html, 'lxml')
        # Найти объект с id=swagger-ui.
    swagger = soup.find(id='swagger-ui')
```

```
        # Распечатать результат.
    print(swagger.prettify())
```

# Scrapy — фреймворк для асинхронного парсинга

Установить фреймворк:

```
pip install scrapy==2.11.0
```

Создать проект Scrapy:

```
scrapy startproject название_ название проекта [папка_для_проекта]
```

## «Пауки»

Для обработки загруженных из интернета страниц в Scrapy используются специальные классы — *spiders* («пауки»).

Создать «паука»:

```
scrapy genspider название_паука адрес_сайта
```

Результат выполнения команды — новый файл *название_паука.py* в папке *spiders* с подобным содержимым:

```
import scrapy


class ExampleSpider(scrapy.Spider):
    name = 'название_паука'
    allowed_domains = ['адрес_сайта']
    start_urls = ['http://адрес_сайта/']

    def parse(self, response):
        pass
```

- `allowed_domains` — разрешение работать только в пределах определённого домена/доменов. Можно перечислить домены в списке.
- `start_urls` — список адресов, с которых нужно начинать парсинг.

- `parse()` — метод с правилами, по которым паук должен собирать информацию. В параметре `response` содержится объект ответа сервера. Из этого объекта будут получены данные при парсинге.

## Объект `response`

Используется для парсинга и взаимодействия с HTML-элементами. С этим объектом удобно работать через интерактивную оболочку Scrapy Shell. Запускается она так:

```
scrapy shell <адрес_сайта или веб-страницы>
```

Выйти из Scrapy Shell можно по команде `quit()` или нажав **Ctrl + z**, а потом **Enter**.

### CSS-селекторы

Работа встроенного парсера Scrapy базируется на работе с селекторами — выражениями, указывающими на один или несколько элементов HTML-кода.

Пример работы:

```
# Получить список всех блоков div из объекта ответа сервера.
>>> quotes = response.css('div')

# Как и из любого списка, из quotes можно получить элемент по инде
>>> quotes[0]
<Selector xpath="descendant-or-self::div[@class and contains(conca

# И можно поискать в этом элементе вложенный тег small.
>>> quotes[0].css('small')
[<Selector xpath='descendant-or-self::small' data='<small class="a
```

Получить содержимое **одного** найденного объекта Selector — `get()`:

```
# В первом элементе списка quote есть два тега span.
# Метод get() возьмёт только первый из них и напечатает его в виде
```

```
>>> quotes[0].css('span').get()
'<span class="text" itemprop="text">“The world as we have created
```

Извлечь тег из текста — `::text`:

```
>>> quotes[0].css('span::text').get()
'“The world as we have created it is a process of our thinking.
It cannot be changed without changing our thinking.”'
```

Найти все теги `<a>` внутри элементов с классом `quote`:

```
>>> response.css('.quote a')
```

**XPath-селекторы**

Поиск элементов через XPath проводится с помощью метода `response.xpath()`:

```
# CSS-селектор.
>>> quotes.css('.author')

# XPath-селектор для тех же элементов.
# В качестве обозначения любого тега ставится звёздочка.
>>> response.xpath('//*[@class="author"]')
```

Конструкция `[@key="value"]` позволяет найти элемент с атрибутом `key`, значение которого равно `value`.

> Метод `response.xpath()` тоже возвращает список объектов Selector, и к нему точно так же применимы методы `get()` и `getall()`, по объектам в списке точно так же можно проводить дополнительный поиск.

Через XPath-селекторы можно найти элемент, который содержит определённый текст; синтаксис такого селектора — `[contains(., 'E')]`.

- `contains` — «содержит»
- `.` — «в **тексте**, который содержится в теге»

- `'E'` — «заглавную E».

```
# Пример поиска авторов,
# в имени которых есть заглавная буква "E".
>>> response.xpath('//small[contains(., "E")]/text()').getall()
# Выведется:
['Albert Einstein', 'Albert Einstein', 'Albert Einstein', 'Thomas

# Подобный поиск можно провести и с помощью CSS-селектора.
>>> response.css('small:contains("E")::text').getall()
# Выведется:
['Albert Einstein', 'Albert Einstein', 'Albert Einstein', 'Thomas
```

Конструкция `/text()` извлекает текстовое содержимое из найденного тега аналогично `::text` в методе `response.css()`.

# Пример парсера

> **Задача:** написать парсер для веб-сайта *http://quotes.toscrape.com/*, на котором хранятся цитаты известных людей. Нужно спарсить следующие данные: текст цитаты, автор, тег.

## Начало работы

1. Описать класс `QuoteItem`, в экземплярах которого будут хранится цитаты, которые соберёт парсер.

   Объекты класса `Item` должны храниться в файле *items.py*:

   ```
   import scrapy


   class QuoteItem(scrapy.Item):
       text = scrapy.Field()
       author = scrapy.Field()
       tags = scrapy.Field()
   ```

2. Создать паука

```
from quotes.items import QuoteItem


class QuotesSpider(scrapy.Spider):
    # Имя паука должно быть уникальным в рамках одного проекта
    name = 'quotes'
    # Стартовая ссылка для парсинга.
    start_urls = [
        'http://quotes.toscrape.com/',
    ]

    # Загрузить и обработать каждую из стартовых ссылок.
    def parse(self, response):
        for quote in response.css('div.quote'):
            # Создать словарь с данными цитаты.
            data = {
                'text': quote.css('span.text::text').get(),
                'author': quote.css('small.author::text').get(
                'tags': quote.css('a.tag::text').getall(),
            }
            # Передать словарь с данными в конструктор класса.
            yield QuoteItem(data)
```

Каждая цитата находится в теге `<div class="quote">`. Логика работы паука:

- Находит на странице все элементы по селектору `div.quote`.
- Извлекает из каждого найденного элемента текст цитаты, имя автора и теги, помещая их в словарь
- `yield` возвращает специальный объект-генератор и не прерывает выполнение метода `parse()`.

3. Запустить паука и создать файл *quotes.csv* с результатами работы парсера:

```
scrapy crawl quotes -o quotes.csv
```

## Переход по ссылкам

Чтобы получить данные об авторе цитаты, нужно научить парсер переходить по ссылкам.

Переходы по ссылкам в процессе парсинга осуществляются с помощью метода `response.follow()`. В нём описываются задачи для планировщика.

```
import scrapy


class AuthorSpider(scrapy.Spider):
    name = 'author'
    allowed_domains = ['quotes.toscrape.com']
    start_urls = ['http://quotes.toscrape.com/']

    def parse(self, response):
        # Найти все ссылки на авторов.
        all_authors = response.css('a[href^="/author/"]')
        # Перебрать ссылки по одной.
        for author_link in all_authors:
            # Вернуть response.follow() с вызовом метода parse_au
                yield response.follow(author_link, callback=self.p

        # Перейти по страницам пагинации (точно как в пауке quotes
        next_page = response.css('li.next a::attr(href)').get()
        if next_page is not None:
            yield response.follow(next_page, callback=self.parse)

        def parse_author(self, response):
            # Здесь будет код для парсинга страниц авторов
            # и возврат полученных данных.
            ...
```

Строка `all_authors = response.css('a[href^="/author/"]')` найдет все теги `<a>` на странице, у которых значение атрибута `href` будет начинаться с `/author/`.

## Потоки данных (Feeds)

*Feeds* — словарь, в котором указывается, куда и как сохранять данные, полученные пауком. Словарь находится в файле <имя_проекта>*/settings.py*

```
...  # Содержимое файла settings.py

FEEDS = {
    # Имя файла для сохранения данных теперь нужно указывать здесь
    # а не при вызове паука из консоли.
    'quotes_text.csv': {
        # Формат файла.
        'format': 'csv',
        # Поля, данные из которых будут выведены в файл, и их поря
        # Вывести в этот файл только два поля из трёх.
        'fields': ['text', 'tags'],
        # Если файл с заданным именем уже существует, то
        # при значении False данные будут дописываться в существу
        # при значении True существующий файл будет перезаписан.
        'overwrite': True
    },
    # И ещё один файл.
    'quotes_author.csv': {
        'format': 'csv',
        # В этот файл попадёт только список авторов.
        'fields': ['author'],
        'overwrite': True
    },
}
```

Чтобы при каждом запуске парсера данные сохранялись в отдельный файл, к названию файла можно добавить шаблон временной метки: `%(time)s`:

```
FEEDS = {
    'quotes_text_%(time)s.csv': {
        ...
        },
}
```

## Конвейеры (Pipelines)

Конвейеры — цепочка обработчиков, которые можно применить к данным, полученным в результате парсинга, перед тем как они будут сохранены. Конвейеры предоставляют возможность предварительной обработки и фильтрации данных, а также их трансформации.

Классы конвейеров ни от чего не наследуются, но в каждом таком классе обязательно должен быть определён метод `process_item(self, item, spider)` — в нём будет происходить обработка каждого объекта `Items`.

Пример конвейера для сохранения результатов в БД с помощью SQLAlchemy:

```
from quotes.models import Quote


class QuotesToDBPipeline:

    def open_spider(self, spider):
        # Создать движок.
        engine = create_engine('sqlite:///sqlite.db')
        # Создать все таблицы.
        Base.metadata.create_all(engine)
        # Создать сессию как атрибут объекта.
        self.session = Session(engine)

    def process_item(self, item, spider):
        # Создать объект цитаты.
        quote = Quote(
            text=item['text'],
            author=item['author'],
            tags=', '.join(item['tags']),
        )
        # Добавить объект в сессию и выполнить коммит сессии.
        self.session.add(quote)
        self.session.commit()
        # Вернуть Item, чтобы обработка данных не прерывалась.
        return item
```

```
    def close_spider(self, spider):
        self.session.close()
```

Чтобы конвейер заработал, его надо подключить в файле *settings.py*:

```
ITEM_PIPELINES = {
    'training_scrapy.pipelines.QuotesToDBPipeline': 300,
}
```

`300` — число для определения порядка выполнения пайплайнов, когда их много.