

Парсинг HTML. XPath

Сбор и разметка данных



Оглавление

Словарь терминов	3
Введение	3
Основы lxml	3
XPath	10
XPath в lxml	12
CSS-селекторы	15
Скрейпинг веб-сайта с помощью XPath	16
Заключение	25
Что можно почитать еще?	25

Словарь терминов

XPath (XML Path Language) — язык запросов к элементам XML-документа.

Узел XPath (node) — вложенные теги, атрибуты и тексты, составляющие содержимое корневого элемента.

Ось XPath определяет отношение узлового набора к текущему узлу.

Выражение XPath (Expression) — путь к нужному элементу в дереве документа, где каждый уровень отделяется от другого косой чертой (/), а результатом его обработки может быть node-set или комплект узлов.

Предикаты XPath — дополнительные условия отбора. Их может быть несколько. Каждый предикат заключается в квадратные скобки и подразумевает логическое выражение для проверки отбираемых элементов. Если предиката нет, отбираются все подходящие элементы.

Введение

Сегодня мы продолжим изучать веб-скрейпинг и перейдем к разбору Python-модуля lxml и языка XPath.

На одном из предыдущих уроков мы познакомились с BeautifulSoup и использовали его для скрейпинга HTML-страниц, на этом будем работать с lxml. BeautifulSoup богаче с точки зрения функционала, но для простых задач подойдет и lxml.

Сегодня наша задача — научиться писать выражения XPath для скрейпинга нужных частей информации. В первой половине урока мы займемся теоретическим разбором инструментов, а во второй — применим знания на практике и выполним скрейпинг сайта IMDb. Советуем открыть VS Code и повторять тот код, который будет на лекции.

Основы lxml

Lxml — это библиотека Python для обработки документов XML и HTML. Она предоставляет быстрый и эффективный API для парсинга, манипулирования и сериализации данных XML и HTML. Основные возможности lxml — поддержка XPath и работа с деревом элементов.

Перейдем в среду разработки Visual Studio Code. В первую очередь установим виртуальную среду. Создать ее можно, например, с помощью Anaconda.

Установка lxml с помощью pip:

```
pip install lxml
```

Чтобы начать работу с lxml, возьмем элементарную HTML-страницу (файл web_page.html).

```
<html lang="en">

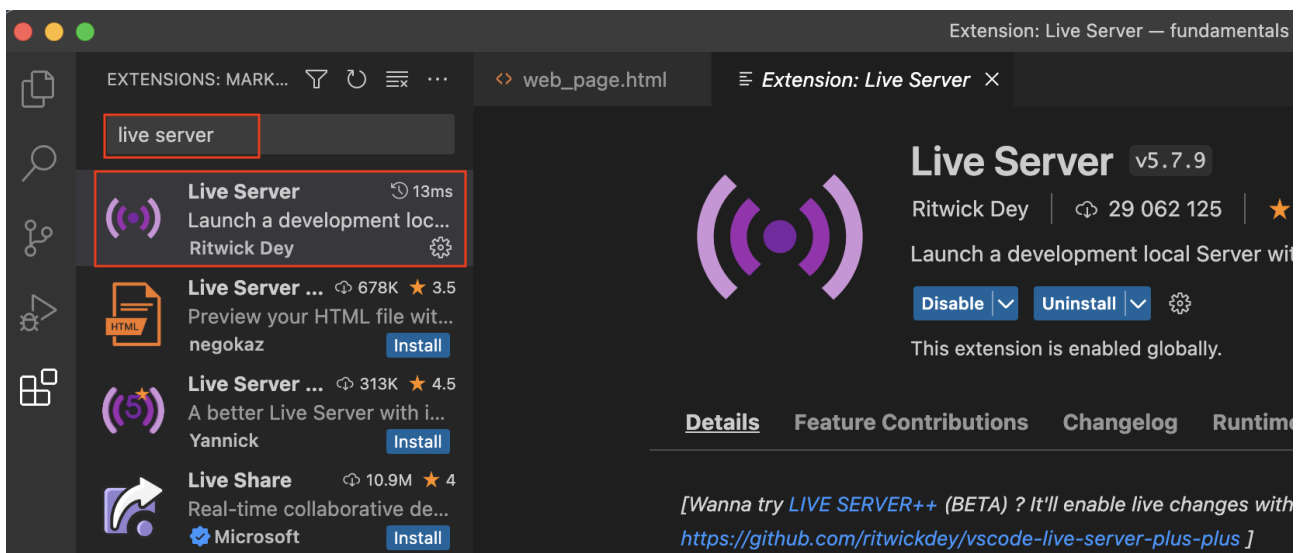
<head>
  <title>This is title of the page</title>
</head>

<body>
  <p>Hello GeekBrains</p>
  <ul>
    <li id="myID">Data scraping course.</li>
    <li class='myClass'>GeekBrain's link:
      <a
href='https://gb.ru/geek_university/developer'>Developer</a>
    </li>
  </ul>
</body>

</html>
```

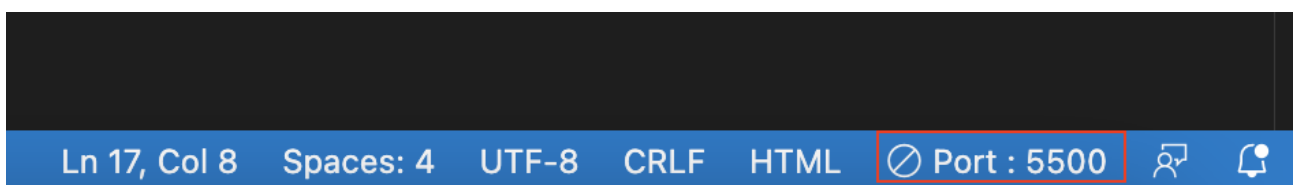
Как видим, в теле страницы один параграф и один список с элементами списка внутри. Второй элемент списка содержит тег гиперссылки.

Можно запустить этот файл прямо из среды разработки. Для этого нужно установить расширение Live Server на вкладке Extensions. После установки расширения переходим на вкладку с HTML-кодом и во всплывающем меню (жмем правую кнопку мышки) выбираем Open with Live Server. Это действие поднимет локальный сервер, и страница откроется в браузере.



Пока мы работаем с игрушечным примером — это нужно для учебных целей. Позже поработаем с настоящим сайтом.

Чтобы остановить сервер, нужно кликнуть в правом нижнем углу, как показано на рисунке:



Теперь создадим новый Python-файл и назовем его app.py.

С помощью lxml откроем нашу учебную веб-страницу. В файле app.py напомним код, чтобы импортировать модуль etree из библиотеки lxml:

```
from lxml import etree
```

В модуле etree есть функция parse() — она принимает в качестве аргумента источник, который может быть файлом или частью файла. Так что в качестве аргумента мы укажем относительный путь к нашему HTML-файлу, сохраним все в переменной tree и выведем ее:

```
tree = etree.parse("src/web_page.html")
print(tree)
```

Функция parse возвращает объект ElementTree:

```
<lxml.etree._ElementTree object at 0x7fab3427d80>
```

Что произошло? Функция `parse` берет HTML-файл и преобразует его в дерево. О дереве HTML мы говорили во второй лекции, сейчас возвращаемся к этой теме. С нашей точки зрения, HTML-файл — это текст с определенным синтаксисом внутри. В то же время XML или HTML можно рассмотреть как дерево элементов.

Посмотрим на нашем примере. Выведем элементы дерева:

```
from lxml import etree

def print_tree(element, depth=0):
    """Рекурсивная печать древовидной структуры элемента HTML"""
    # Вывод текущего элемента с соответствующим отступом
    print("-" * depth + element.tag)

    # Рекурсивная печать дочерних элементов с увеличенным отступом
    for child in element.iterchildren():
        print_tree(child, depth + 1)

# Парсинг HTML-документа
tree = etree.parse("src/web_page.html")

# Получение корневого элемента дерева
root = tree.getroot()

# Вывод структуры дерева
print_tree(root)
```

В этом коде мы определяем рекурсивную функцию `print_tree()`, которая принимает на вход объект `lxml.etree._Element`, а также необязательный аргумент `depth`, задающий текущий уровень отступа (по умолчанию 0).

Сначала функция выводит имя тега текущего элемента с соответствующим отступом, основанным на текущей глубине. Затем она перебирает дочерние элементы текущего элемента, используя метод `iterchildren()`, и рекурсивно вызывает `print_tree()` для каждого дочернего элемента с увеличенной глубиной отступа. В результате древовидная структура HTML печатается в нужном формате.

Обратите внимание: эта функция печатает всю древовидную структуру HTML-документа, которая может быть довольно большой для сложных документов, поэтому служит здесь только для демонстрации.

```
html
--head
----title
--body
----p
----ul
-----li
-----li
-----a
```

Мы получили древовидную структуру, в которой все теги преобразованы в объекты `element`.

Элемент `html` — корневой, у него есть два дочерних элемента — `head` и `body`. У `head` один дочерний элемент — `title`. У `body` два — `p`, `ul`.

У элемента `ul` два дочерних элемента `li`. У первого `li` есть атрибут `id`, установленный на «`myID`». У второго — атрибут `class`, установленный в «`myClass`», и один дочерний элемент — `a`.

В древовидной структуре HTML элемент считается родительским, если у него есть один или несколько дочерних элементов. Дочерний элемент — это элемент, вложенный в другой элемент.

Представим, что мы хотим извлечь или сделать скрейпинг заголовка нашей HTML-страницы. Мы можем это сделать, потому что теперь у нас есть дерево элементов и существует метод, который позволяет извлечь заголовок из нашей веб-страницы. У объекта `ElementTree` есть метод **`find()`**, который принимает в качестве аргумента путь к элементу или тегу, который мы хотим извлечь.

В нашей HTML-разметке мы знаем, что заголовок находится внутри тега `<head>`, поэтому в методе `find` вводим строковое значение — путь к заголовку `head/title`:

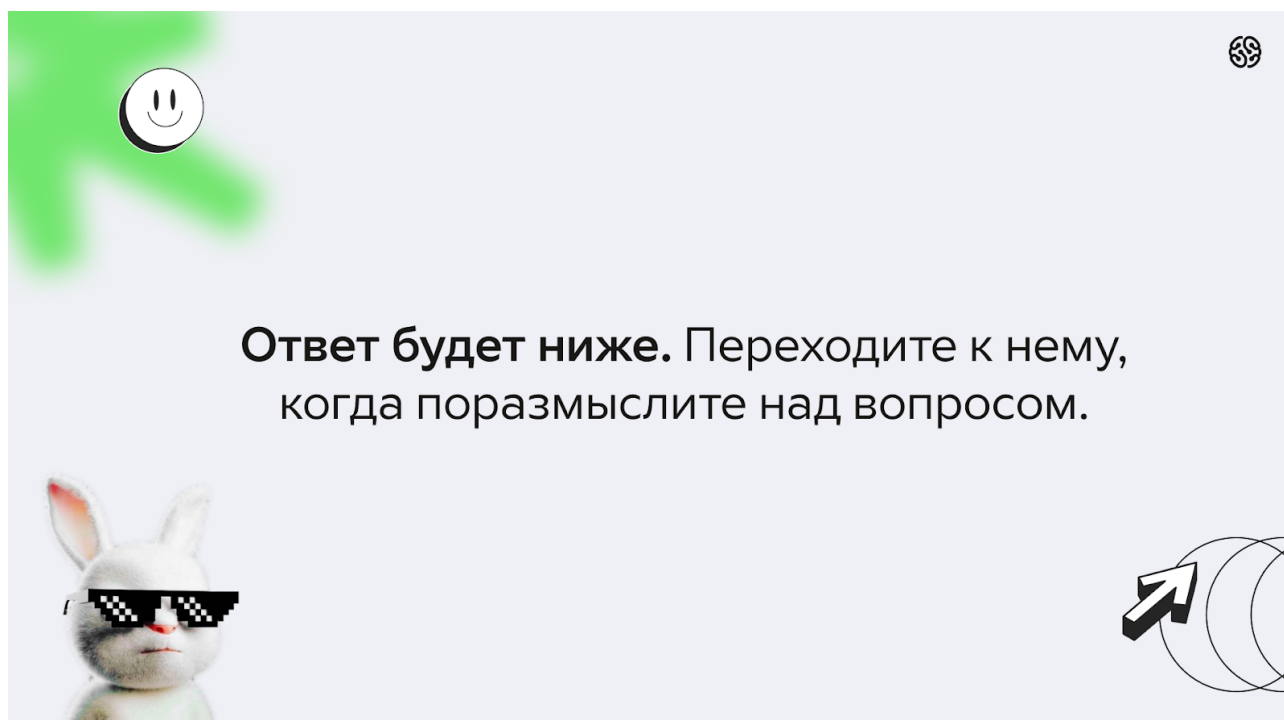
```
title_element = tree.find("head/title")
print(title_element)
```

Это выражение вернет `<Element title at 0x7fb8467cd5c0>` — объект `Element` типа `title`.

Если теперь мы хотим вывести фактическое значение заголовка, можем вызвать свойство `text` элемента `title`:

```
print(title_element.text)
```

Вопрос. Почему в пути, который мы указали в методе `find()`, мы начали не с тега `<html>`, а с тега `<head>`?



Тег `<title>` находится внутри тега `<head>`, а тег `<head>` — внутри тега `<html>`. Мы всегда должны начинать с одного из прямых дочерних элементов. Например, если мы хотим выбрать тег абзаца — `<p>Hello GeekBrains</p>` — мы должны начать с тега `<body>`, потому что `<body>` — прямой дочерний элемент тега `<html>`.

Задание. Выведите текст абзаца `<p>Hello GeekBrains</p>`, используя метод `find`.

Чтобы вывести содержимое тега `<p>`, мы вызываем `tree.find()` и указываем путь к тегу абзаца — `body/p`:

```
p_element = tree.find("body/p")
print(p_element.text)
```

В качестве альтернативы методу `find` у нас есть другой метод — **`findall()`**, аналогично тому, как было в `BeautifulSoup`. По названию понятно, что он будет находить все совпадающие теги и возвращать их в виде списка. Например, чтобы выбрать все элементы списка внутри тега ``, мы должны использовать метод `findall()`. Если используем `find`, получим только первый элемент списка.

Создадим переменную `list_items`, равную `tree.findall()`. Внутри нее укажем путь к элементам списка — `body/ul/li`. Затем выведем `list_items`:


```
list_items = tree.findall("body/ul/li")
print(list_items)
```

Мы получили список из двух элементов типа li:

```
[<Element li at 0x7fc6747cc880>, <Element li at 0x7fc6747cc8c0>]
```

Чтобы извлечь текстовое содержимое, нужно обратиться к каждому элементу списка, например, в цикле:

```
for li in list_items:
    print(li.text)
```

```
>>>
Data scraping course.
GeekBrain's link:
```

Обратите внимание, что для первого элемента списка мы получили полное текстовое значение, однако для второго элемента списка мы не видим содержимого тега <a>. Причина в том, что в методе findall мы указали, что хотим получить все элементы списка, и нас не интересует, есть ли внутри него тег <a>.

Чтобы решить эту проблему, нужно внутри цикла создать переменную или что-то еще. Поэтому одно быстрое решение, которое мы можем применить, — выполнить поиск внутри цикла for. Создадим переменную a = li.find("a"). Так как у нас только один элемент содержит тег <a>, используем оператор if, и, если a не равно none, значит, внутри элемента списка есть тег <a>, содержимое которого мы выводим.

```
for li in list_items:
    a = li.find("a")
    if a is not None:
        print(f"{li.text} {a.text}")
    else:
        print(li.text)
```

Обратите внимание, что перед словом «Developer» есть большой пробел. Это связано с нашей HTML-разметкой, где есть пробелы перед словом «Developer». Чтобы очистить вывод, можем вызвать метод strip():

```
for li in list_items:
    a = li.find("a")
```

```
if a is not None:
    print(f"{li.text.strip()} {a.text}")
else:
    print(li.text)
```

Один из недостатков lxml — он небогат в плане методов, которые открываются нам через объект ElementTree. Более того, каждый раз, когда мы хотим выбрать тег из разметки HTML, мы должны указать абсолютный путь до тега. Если мы имеем дело с большой HTML-страницей, это может быть проблемой.

Решить эту проблему можно с помощью более эффективного метода выбора тегов в HTML — XPath.

XPath

Xpath (XML Path Language) — язык запросов к элементам XML-документа. Используется для уникальной идентификации или адресации частей XML-документа.

Выражение XPath можно использовать для поиска в XML-документе и извлечения информации из любой части документа, например, элемента или атрибута (в XML он называется узлом — node). Но это, конечно, не значит, что мы не можем использовать его для запроса или выбора элементов / тегов из HTML-страницы.

Начнем изучать XPath с выражений (XPath Expression).

XPath Expression — выражение, определяющее шаблон для выбора узлов. Как мы уже обсуждали, HTML-документ рассматривается как дерево узлов.

В XPath мы можем выбрать элемент, используя двойной слэш, а затем имя элемента. Например, если хотим выбрать все div на HTML-странице, используем //div.

Также мы можем выбирать элементы по их атрибуту, id или классу. Для этого добавляем две квадратные скобки, следующие за именем элемента, а затем значение атрибута.

```
//elementName [@attribute = 'value']
//elementName [@id = 'value']
//elementName [@class = 'value']
```

Мы можем выбрать элемент на основе его позиции. Например, если хотим выбрать первый элемент списка ``:

```
//li[1]
```

Если хотим выбрать первый и второй элемент списка, мы должны использовать функцию `position()` плюс логический оператор, например:

```
//li [position() = 1 or position = 2]
```

Если хотим выбрать только первый элемент списка, который должен содержать текст "hello", используем функцию `contains()`:

```
//li [position() = 1 and contains(@text, "hello")]
```

В XPath все, что мы пишем в квадратных скобках, называется **предикатом** (условием).

У XPath есть еще одно полезное свойство — возможность перемещаться в дереве HTML вверх или вниз, используя оси XPath.

Оси XPath — это способ выбора элементов, которые находятся относительно текущего элемента в иерархии документа.

Ось — это именованная связь между элементами, которая определяет направление и набор узлов для выбора. Например, ось предков (`ancestor`) выбирает все элементы-предки текущего элемента — то есть элементы, расположенные выше в иерархии документа. А ось потомков (`following-sibling`) выбирает все элементы-потомки, которые идут после текущего элемента.

Синтаксис такой: указываем имя оси, двойное двоеточие и затем целевой элемент, который ищем:

```
axisName::elementName
```

Для перехода вверх по дереву HTML есть четыре оси:

- **parent** — возвращает родителя определенного узла.
- **ancestor** — позволяет получить всех предков определенного узла.
- **preceding** — выбирает все узлы, которые появляются перед текущим узлом, за исключением предков, узлов атрибутов и пространства имен.

- **preceding-sibling** — возвращает «братьев» определенного элемента, то есть все элементы одного уровня до текущего узла.

Для перехода вниз по дереву HTML тоже есть четыре оси:

- **child** — получает дочерние элементы (потомков) определенного узла.
- **following** — возвращает все элементы, находящиеся после закрывающего тега определенного узла.
- **following-sibling** — возвращает все элементы одного уровня после текущего узла.
- **descendant** — возвращает всех потомков текущего узла.

XPath в lxml

Для выбора тегов с помощью XPath в lxml мы можем заменить метод `find` методом **XPath**. Метод XPath принимает в качестве аргумента путь к целевому тегу точно так же, как и метод `find`, но особенность в том, что нам не нужно указывать полный путь до тега.

Так, для выбора заголовка нам нужно сделать запрос следующим образом:

```
title_element = tree.xpath("//title")
print(title_element[0].text)
```

Метод XPath возвращает список, поэтому, чтобы получить доступ к свойству `text`, нужно получить доступ к первому элементу списка. Обращаемся в квадратных скобках к первому элементу — `[0]`.

Далее XPath позволяет получить доступ к тексту без использования свойства `text`, связанного с lxml. Мы можем вызвать функцию внутри метода XPath под названием `text` и получить тот же результат:

```
title_element = tree.xpath("//title/text()")[0]
print(title_element)
```

Попробуйте самостоятельно получить доступ к тексту тега `<p>` и вывести текст «Hello GeekBrains».

Теперь давайте получим данные из списка в HTML-документе с помощью XPath. Только на этот раз будем использовать не функцию `text`, а метод `etree.tostring`:

```
list_items = tree.xpath("//li")
for li in list_items:
    print(etree.tostring(li))
```

После выполнения кода мы увидим, что получили два элемента списка, содержащих полную HTML-разметку.

```
b'<li id="myID">Data scraping course.</li>\n          <li
class="myClass">GeekBrain\'s link:\n          <a
href="https://gb.ru/geek_university/developer">Developer</a>\n
</li>\n      </ul>\n</body>\n\n</html>\n          '
b'<li class="myClass">GeekBrain\'s link:\n          <a
href="https://gb.ru/geek_university/developer">Developer</a>\n
</li>\n      </ul>\n</body>\n\n</html>\n          '
```

Чтобы получить текст, который находится внутри элементов списка, создадим переменную в цикле, которая будет применять функцию Xpath text() к каждому элементу списка:

```
list_items = tree.xpath("//li")
for li in list_items:
    text = li.xpath("//text()")
    print(text)
```

После выполнения кода мы снова получаем два списка, однако все еще получаем текст со всеми специальными символами в HTML-коде. Чтобы это исправить, нужно добавить точку перед двойным слешем функции text — “//text()”.

Если мы еще раз выполним код, на этот раз получим только текст двух тегов .

```
['Data scraping course.']
["GeekBrain's link:\n          ", 'Developer', '\n          ']
```

Пока что мы получили два списка, первый из которых содержит только один элемент с текстом, а второй — три элемента, но все еще содержит пробелы. Избавимся от этих лишних символов. Для этого можем использовать метод map, который принимает в качестве первого аргумента функцию, которая будет применена к каждому элементу списка. Используем str.strip, чтобы удалить все пробелы и \n символы. В качестве второго аргумента будет список, возвращаемый li.xpath. Поскольку функция map возвращает объект map, чтобы увидеть список преобразуем объект map в список:

```
list_items = tree.xpath("//li")
for li in list_items:
    text = map(str.strip, li.xpath("./text()"))
    print(list(text))
```

Выполняем код:

```
['Data scraping course.']
["GeekBrain's link:", 'Developer', '']
```

Наконец, чтобы код возвращал не списки, а только текст, мы можем использовать другую функцию — join:

```
list_items = tree.xpath("//li")
for li in list_items:
    text = ''.join(map(str.strip, li.xpath("./text()")))
    print(text)
```

Выполняем код и получаем чистый текст.

Наконец, давайте посмотрим пример работы с осями XPath. Выберем все элементы li, которые являются потомками элемента ul:

```
list_items = tree.xpath("//ul/descendant::li")
for li in list_items:
    text = ''.join(map(str.strip, li.xpath("./text()")))
    print(text)
```

Выберем все элементы, которые являются предками тега

```
list_items = tree.xpath("///li/parent::*")
for li in list_items:
    text = ''.join(map(str.strip, li.xpath("./text()")))
    print(text)
```

Таким образом, XPath — это мощный инструмент для выбора и навигации по элементам в документах XML и HTML. Он особенно полезен для задач веб-скрейпинга и извлечения данных.

В этой части лекции мы рассмотрели основы использования XPath в lxml для выбора и извлечения данных из HTML-документов. А сейчас познакомимся с еще одним инструментом для извлечения данных из HTML-кода — CSS-селекторами.

CSS-селекторы

CSS служит для стилизации HTML-страниц.

Мы можем использовать lxml вместе с CSS-селекторами для скрейпинга тега или элемента стилизации веб-страницы.

Чтобы использовать CSS-селекторы, нам нужно установить модуль под названием cssselect:

```
pip install cssselect
```

Теперь вместо метода XPath мы можем использовать метод cssselect(), который принимает в качестве аргумента CSS-селектор. Например, чтобы выбрать заголовок из HTML-страницы, все, что нам нужно сделать, — вызвать тег <title>:

```
title_element = tree.cssselect("title")
print(title_element[0].text)
```

Аналогично методу XPath метод cssselect() вернет список, поэтому нам нужно сохранить квадратные скобки с индексом 0. И, поскольку у нас нет функции в CSS, которая выбирает текстовое содержимое, нужно вызвать свойство text так же, как в методе find().

Если выполним файл в таком виде, получим ошибку — AttributeError: 'lxml.etree._ElementTree' object has no attribute 'cssselect'. Это связано с тем, что cssselect работает непосредственно с HTML-элементами, а не с объектом ElementTree. Поэтому предварительно нужно конвертировать ElementTree. Создадим для этого новую переменную html = tree.getroot(). Эта функция конвертирует объект дерева в HTML-элемент.

```
html = tree.getroot()
title_element = html.cssselect("title")
print(title_element[0].text)
```

Задание. Используйте метод cssselector вместо XPath и выберите тег абзаца.

Теперь сделаем то же самое с элементами списка.

```
list_items = html.cssselect("li")
for li in list_items:
    a = li.cssselect("a")
```

```
if len(a) == 0:
    print(li.text)
else:
    print(f"{li.text.strip()} {a[0].text}")
```

CSS-селекторы и XPath — два разных языка запросов для извлечения данных из документов HTML и XML. Хотя цели языков общие, между ними есть несколько ключевых различий.

XPath — более мощный и гибкий язык запросов. Он позволяет использовать более сложные выражения и поддерживает более широкий спектр типов данных и функций. XPath также поддерживает оси, которые позволяют перемещаться по иерархии документа и выбирать элементы на основе их связи с другими элементами.

CSS-селекторы — более простой и лаконичный язык запросов. Может быть полезен для задач веб-скрейпинга, связанных с выбором элементов на основе их класса, ID или других CSS-селекторов. Для таких задач его часто проще использовать и понимать, чем XPath.

Одно из главных преимуществ CSS-селекторов перед XPath в том, что они быстрее и эффективнее для определенных типов выбора, особенно при выборе на основе атрибутов класса или ID.












Но все же XPath остается более мощным и гибким языком запросов. Он лучше подходит для сложных задач извлечения данных, которые включают выбор элементов на основе их текстового содержимого, структуры или других атрибутов.

Итог — выбор языка зависит от требований задачи извлечения данных.

Скрейпинг веб-сайта с помощью XPath

В этой части лекции займемся скрейпингом сайта IMDb, а именно страницы с самыми популярными фильмами: [Most Popular Movies](#). Познакомимся с некоторыми хитростями, которые помогут скрейпировать веб-страницы, избавимся от ненужных классов, внедренных в JavaScript, а также сохраним скрейпированные данные в MongoDB.





Переходим на целевую страницу:

IMDb Charts			
Most Popular Movies			
As determined by IMDb Users			
Showing 100 Titles		Sort by: Ranking	
Rank & Title	IMDb Rating	Your Rating	
 Аватар: Путь воды (2022) 1 ( 4)	 8,0		
 Пинокио Гильермо дель Торо (2022) 2 ( 13)	 7,8		
 Аватар (2009) 3 ( 25)	 7,8		
 Быстрее пули (2022) 4 ( 3)	 7,3		

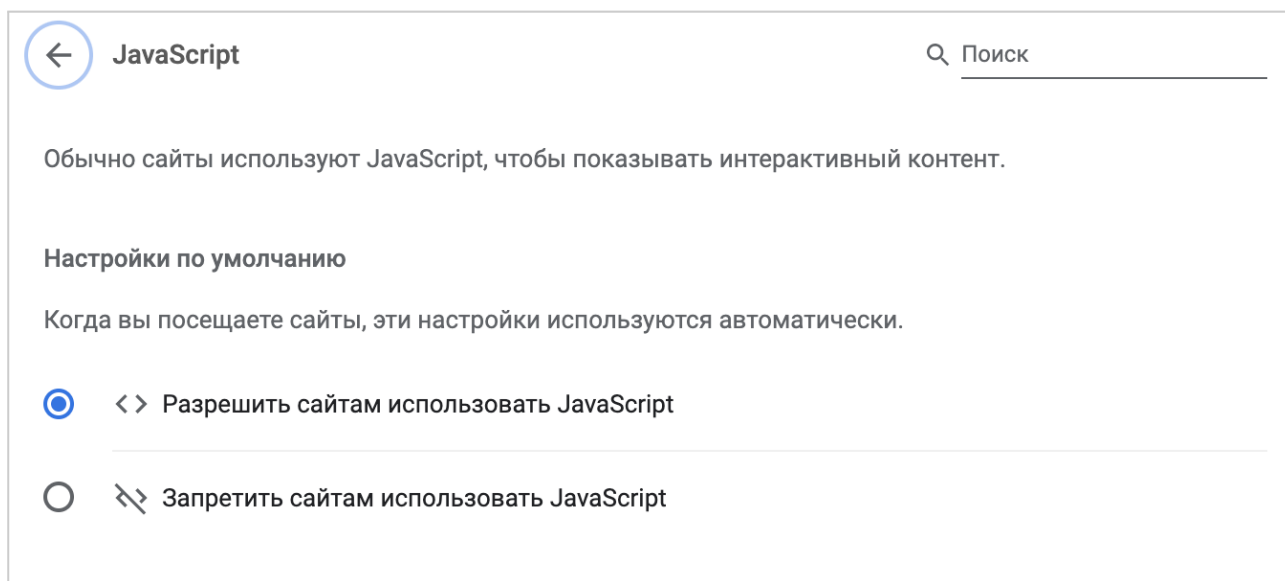
Напишем код, который будет извлекать название, место в рейтинге, изменение места в рейтинге, IMDb Rating.

Как вы знаете, большинство сайтов используют JavaScript. Проблема в том, что библиотека requests в Python не понимает JavaScript, поэтому высоки шансы, что мы не получим ту же информацию, которую видим при отображении страницы в браузере.

Чтобы исключить JavaScript из разметки, можно использовать трюк: нажимаем три вертикальных точки в правом верхнем углу браузера Chrome, идем в настройки, в поисковой строке вводим «javascript», находим раздел «Контент».

Контент	
	Файлы cookie и данные сайтов Сторонние файлы cookie заблокированы в режиме инкогнито.
	JavaScript Разрешить сайтам использовать JavaScript
	Картинки Разрешить сайтам показывать изображения
	Всплывающие окна и переадресация Запретить сайтам показывать всплывающие окна и использовать переадресацию

Переходим в раздел JavaScript и ставим переключатель на «Запретить сайтам использовать JavaScript».



Теперь, если мы вернемся на сайт IMDb и обновим страницу, то в HTML-коде будет отображаться только разметка, которую может получить библиотека Python requests. Есть сайты, полностью построенные на JavaScript, так что мы просто не увидим контента с этой отключенной опцией. Скрейпингом подобных сатов мы займемся позже в курсе.

Перейдем в среду разработки. Для начала отправим HTTP-запрос и проверим, что все работает.

```
import requests

resp =
requests.get(url='https://www.imdb.com/chart/moviemeter/?ref_=nv_
mv_mpm', headers = {
    'User-Agent' : 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0
Safari/537.36'
})

print(resp.status_code)
```

Указание вашего агента пользователя в аргументе запроса — хороший тон по ряду причин.

Во-первых, некоторые веб-сайты могут блокировать запросы, которые не имеют действительного агента пользователя или имеют необычный агент пользователя.

Во-вторых, некоторые сайты могут предоставлять разное содержимое или HTML-разметку в зависимости от пользовательского агента запрашивающего браузера.

Наконец, у некоторых сайтов могут быть соглашения об использовании, которые запрещают скрейпинг или извлечение данных. Указывая действительный пользовательский агент в запросе на скрейпинг, вы идентифицируете себя как законного пользователя сайта, а не скрепера или бота, который может нарушать условия обслуживания.

Вы можете найти свой пользовательский агент с помощью Chrome DevTools:

1. Откройте Chrome DevTools: щелкните правой кнопкой мыши в любом месте страницы и выберите «Просмотреть код».
2. В окне DevTools нажмите на значок с тремя точками (⋮) в правом верхнем углу окна или нажмите клавишу Esc, чтобы открыть консоль DevTools.
3. Введите в консоли `navigator.userAgent` и нажмите Enter.

В консоли отобразится строка агента пользователя — текстовая строка, которая идентифицирует используемый браузер и операционную систему.

Начнем с построения дерева. Импортируем `html` из `lxml` и создадим переменную `tree`:

```
import requests
from lxml import html

resp = requests.get(url='https://www.imdb.com/chart/moviemeter/?ref_=nv_mv_mpm',
headers = {
    'User-Agent' : 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36'
})

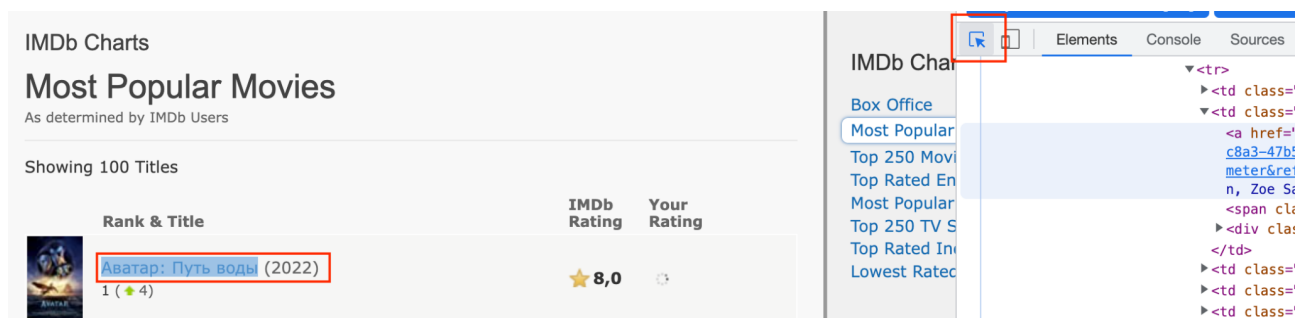
tree = html.fromstring(html=resp.content)
```

Третья строка кода создает парсинг-дерево `lxml HTML` из содержимого объекта ответа.

Метод `html.fromstring()` используется для создания нового объекта `lxml.etree._Element` из HTML-содержимого ответа, который хранится в атрибуте `content` объекта `resp`.

Этот объект `lxml.etree._Element` представляет собой корень HTML-документа и может быть использован для извлечения данных из документа с помощью выражений XPath.

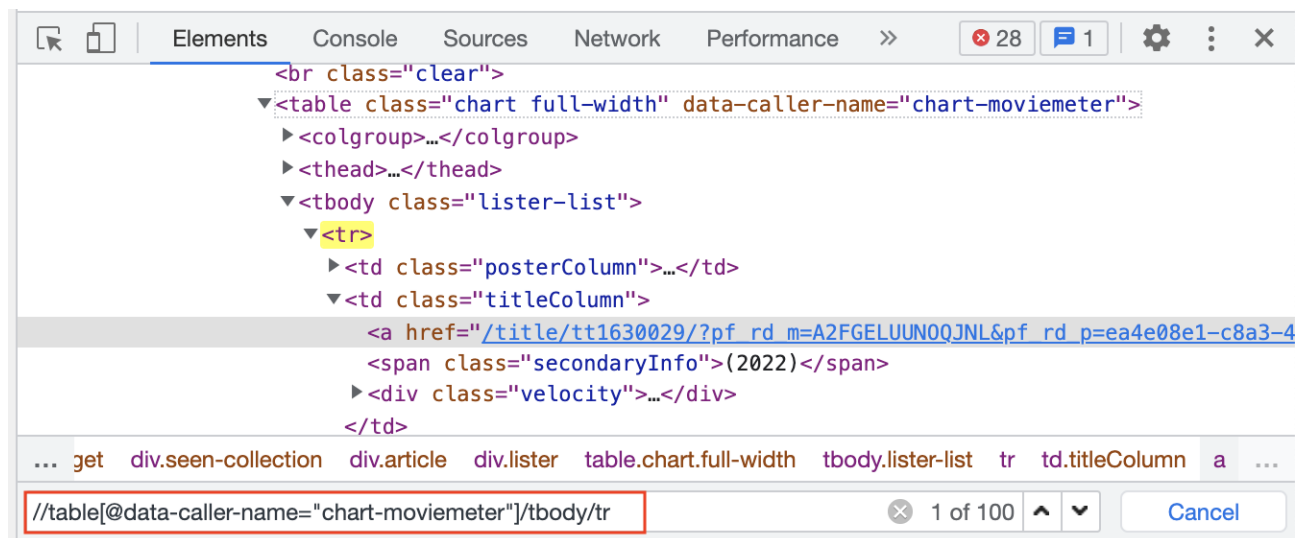
Возвращаемся к странице сайта (убедитесь, что JavaScript отключен). Выбираем Selection tool и наводим курсор на название первого фильма:



Вы можете увидеть, что элемент с названием находится в таблице, которая имеет аргументы class и data-caller-name. Далее следуют элементы <colgroup> и <thead> которые для нас не важны, а затем <tbody> — собственно таблица.



Нажимаем Ctrl + F и в появившейся строке поиска вводим выражение XPath — путь до строки, относящийся к первому фильму:



Копируем выражение XPath, переходим в среду разработки. Создадим переменную `movies` и скопируем все в выражение `xpath`:

```
movies = tree.xpath("//table[@data-caller-name='chart-moviemeter']/tbody/tr")
```

Можно сократить это выражение до:

```
movies = tree.xpath("//tbody/tr")
```

Эта переменная будет представлять собой список, содержащий все элементы типа `tr`, из которых нам нужно получить содержимое. Создадим цикл, в котором будем проходить по списку фильмов:

```
for movie in movies:
```

Теперь вернемся к Chrome, первый `<td>`-элемент — это первый столбец таблицы, содержит изображение фильма, второй `<td>` — второй столбец. Внутри второго тега `<td>` есть тег ссылки `<a>`, который, кроме прочего, содержит текстовое значение названия фильма.

Напишем XPath-выражение для доступа к названию фильма:

```
//tbody/tr/td[@class='titleColumn']/a/text()
```

Возвращаемся в среду разработки и внутри цикла добавим ключ:

```
'name' : movie.xpath("./td[@class='titleColumn']/a/text()")[0]
```

Обратите внимание, поскольку мы теперь имеем дело не с переменной дерева (`tree`), а с элементом списка, то мы должны использовать точку в начале каждого выражения XPath. Также, поскольку возвращается список, мы берем только первый элемент списка, поэтому добавляем `0` в квадратных скобках.

Теперь вернемся к Chrome и выясним, как мы можем выбрать год выпуска фильма:

```
//tbody/tr/td/span[@class='secondaryInfo']
```

Возвращаемся в среду разработки и добавляем метку `release_year` с нашим `xpath`-выражением:

```

for movie in movies:
    m={
        'name' : movie.xpath("//td[@class='titleColumn']/a/text()")[0],
        'release_year' :
movie.xpath("//td/span[@class='secondaryInfo']/text()")[0]}

```

Повторяем те же действия для положения (номера) в рейтинге, а также для изменения рейтинга (стрелка вверх/вниз) и для количества пунктов, на которые изменился рейтинг.

```

for movie in movies:
    m={
        'name' : movie.xpath("//td[@class='titleColumn']/a/text()")[0],
        'release_year' :
movie.xpath("//td/span[@class='secondaryInfo']/text()")[0],
        'position' : movie.xpath("//td/div[@class='velocity']/text()")[0],
        'titlemeter' : movie.xpath("//span[contains(@class, 'global-sprite
titlemeter')]/@class"),
        'position_change' :
movie.xpath("//div/span[@class='secondaryInfo']/text()[2]") }

```

Обратите внимание на то, что изменение рейтинга находится в теге `` и обозначается последним значением атрибута `class` — `up` или `down`. Чтобы получить доступ ко всему содержимому `<class>`, мы используем выражение XPath `@class`. Однако есть фильмы, у которых рейтинг никак не изменился, в таких случаях тег `` просто отсутствует. При этом нам будет возвращаться просто пустой список. Разберемся с этим чуть позже.

Теперь создаем пустой список `all_movies = []` вне цикла, а в цикле добавляем в каждой итерации соскрейпленные элементы в список. Наконец, выведем список, а также длину списка, чтобы свериться с количеством полученных элементов. Оно должно соответствовать количеству фильмов на странице сайта.

```

import requests
from lxml import html

resp = requests.get(url='https://www.imdb.com/chart/moviemeter/?ref_=nv_mv_mpm',
headers = {
    'User-Agent' : 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36'
})

tree = html.fromstring(html=resp.content)

movies = tree.xpath("//tbody/tr")

all_movies = []

for movie in movies:
    m={
        'name' : movie.xpath("./td[@class='titleColumn']/a/text()")[0],
        'release_year' :
movie.xpath("./td/span[@class='secondaryInfo']/text()")[0],
        'position' : movie.xpath("./td/div[@class='velocity']/text()")[0],
        'titlemeter' : movie.xpath("./span[contains(@class, 'global-sprite
titlemeter')]/@class"),
        'position_change' :
movie.xpath("./div/span[@class='secondaryInfo']/text()[2]")
    }

    all_movies.append(m)

print(all_movies)
print(len(all_movies))

```

После успешного запуска кода мы получаем элементы списка в следующем виде:

```

[{'name': 'Avatar: The Way of Water', 'release_year': '(2022)',
'position': '1\n', 'titlemeter': ['global-sprite titlemeter up'],
'position_change': ['\n4)']}

```

Как видно, результат содержит ряд лишних элементов: скобки, специальные символы, а строку ['global-sprite titlemeter up'] для удобства можно было бы сократить до up или down.

Давайте по порядку. Чтобы убрать скобки из 'release_year', используем метод strip. Также год желательно перевести из типа string в тип integer:

```

'release_year' :
int(movie.xpath("./td/span[@class='secondaryInfo']/text()")[0].s
trip('()'))

```

Метки 'position' содержат символ перевода строки \n. Кроме того, у фильмов, у которых не изменилось положение в рейтинге, эта строка будет содержать фразу «no change». Давайте воспользуемся методом split() и возьмем только первый элемент, то есть номер позиции, а также изменим тип данных на integer:

```
'position' :
int(movie.xpath("./td/div[@class='velocity']/text()")[0].split()
[0])
```

С изменением позиции все несколько сложнее, так как у некоторых фильмов не изменился рейтинг, и поэтому наш код в таких случаях будет возвращать пустой список. Кроме того, мы хотим извлечь из ['global-sprite titlemeter up'] только слово «up». Можно было бы воспользоваться методом .split()[-1], но тогда мы будем получать ошибку в случае, если рейтинг не изменился и нам вернулся пустой список. Решить проблему можно с помощью Python-блока try-except. Давайте создадим функцию, которая будет обрабатывать исключения:

```
def get_titlemeter(list_element):
    try:
        return(list_element[0].split()[-1])
    except:
        return "no change"
```

Теперь в цикле будем вызывать функцию, используя в качестве аргумента наше XPath-выражение:

```
'titlemeter' :
get_titlemeter(movie.xpath("./span[contains(@class,
'global-sprite titlemeter')]/@class"))
```

Наконец, 'position_change' нужно избавить от управляющих символов, а также обработать исключение в случае, если позиция не изменилась, и вернуть 0.

Воспользуемся тем же приемом и напишем функцию, которая будет обрабатывать исключения:

```
def get_position_change(list_element):
    try:
        return(int(list_element[0].strip()[:-1]))
    except:
        return 0
```


Следующий этап — добавление данных в MongoDB. Напишем функцию для добавления полученных данных в облачное хранилище:

```
from pymongo import MongoClient

def insert_to_db(list_movies):
    client = MongoClient("mongodb://localhost:27017")
    db = client["imdb_movies"]
    collection = db["top_movies"]
    collection.insert_many(list_movies)
    client.close()
```

И после цикла вызываем функцию:

```
insert_to_db(all_movies)
```

Заключение

Парсинг и скрейпинг HTML с помощью lxml и XPath — это мощная техника извлечения данных с веб-сайтов. Библиотека lxml предоставляет быстрый и эффективный способ парсинга HTML, а XPath позволяет гибко и точно выбирать элементы и атрибуты в подобных документах. Комбинируя эти инструменты с Python и другими библиотеками анализа данных, можно собирать и обрабатывать большие объемы данных из интернета для использования в исследованиях, анализе и других приложениях.

Что можно почитать еще?

1. [Документация XPath](#)
2. [Документация LXML](#)