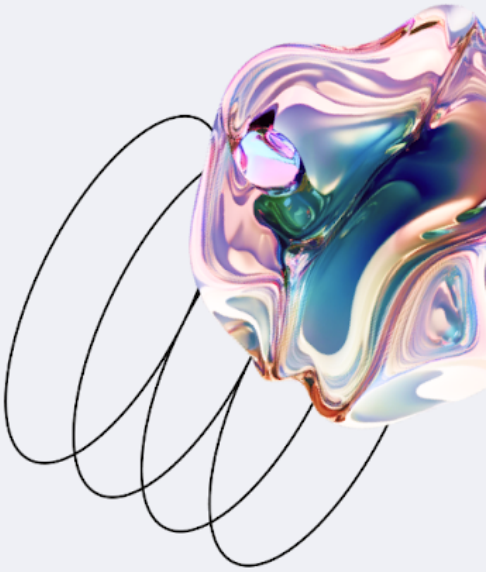
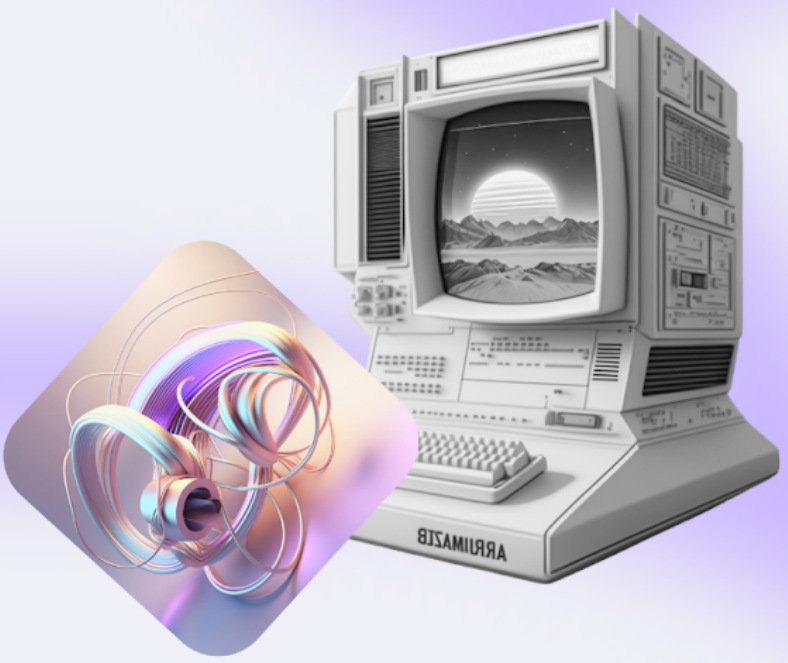
GeekBrains



Scrapy. Парсинг фото и файлов

Сбор и разметка данных



Оглавление

Введение

Добро пожаловать на вторую лекцию, посвященную Scrapy.

Современный мир основан на данных. Способность собирать и анализировать большие данные может стать решающей для многих приложений. Веб-скрейпинг — извлечение данных из интернета — один из основных методов сбора данных. И он не ограничивается только текстом. Поэтому сегодня мы займемся скрейпингом изображений и файлов из интернета.

Скрейпинг изображений играет важную роль для машинного обучения. Алгоритмы машинного обучения часто полагаются на большие датасеты для обучения и проверки своих моделей. Часто эти датасеты состоят из изображений, аудиофайлов или другого мультимедийного контента. Так что сбор фотографий и файлов из интернета может быть эффективным способом создания таких баз данных, особенно для задач, связанных с распознаванием изображений, обработкой естественного языка или анализом аудио.

Скрейпинг фотографий и файлов важен для агрегации контента, поскольку помогает обеспечить богатый и увлекательный пользовательский опыт. Например, агрегатор новостей может собирать изображения и видео по теме новостных статей, а музыкальная платформа — аудиофайлы и обложки альбомов.

Кроме того, скрейпинг фотографий и файлов может быть полезен для решения задач анализа данных. Например, организации могут захотеть проанализировать изображения с платформ соцсетей, чтобы выявить тенденции, закономерности или настроения среди пользователей. А исследователи могут анализировать аудиофайлы, чтобы изучить особенности речи, акценты или лингвистические особенности.

Таким образом, скрейпинг фото и файлов — важный навык для специалиста по сбору данных, который пригодится для решения многих задач.

Проблемы скрейпинга фото и файлов

Сбор фотографий и файлов из интернета сопряжен с целым рядом проблем.

Разные форматы файлов. Популярные форматы изображений — JPEG, PNG и GIF, файлов — PDF, DOCX и MP3. Есть и множество других. При скрейпинге файлов важно определить правильный формат и обработать их соответствующим образом.

Разные форматы могут требовать разных методов обработки или библиотек. Обеспечить совместимость — непростая задача: некоторые форматы могут быть запатентованы, другие требуют специального ПО для открытия или обработки. Все это усложняет процесс скрейпинга.

Размер файлов. Размеры файлов могут значительно отличаться. Изображения высокого разрешения или большие документы занимают много места и требуют большой пропускной способности. Скрейпинг больших файлов может увеличить время загрузки и потребовать значительных ресурсов, что повлияет на производительность проекта.

Важно учитывать размер файла при разработке веб-скрейпера и применять стратегии для эффективной обработки больших файлов. Среди них — асинхронная загрузка файлов или сжатие данных в процессе скрейпинга.

Требования к хранению. Хранение фотографий и файлов — сложная задача из-за объема данных, особенно при работе с большими датасетами или фотографиями высокого разрешения. Требования к хранению могут повлиять как на емкость локального хранилища, так и на стоимость хранения данных на облачных платформах.

Важно спланировать потребности в хранении данных и изучить разные варианты хранения (локальное или облачное) в зависимости от потребностей проекта.

Обеспечение целостности данных. При скрейпинге фотографий и файлов важно убедиться, что загруженные данные точные и полные. Например, проверить правильности загрузки файлов, выяснить, есть ли поврежденные или неполные файлы, обеспечить точность и согласованность метаданных.

Item Pipelines

Item pipelines (конвейеры элементов) — это серия классов Python, которые определяют, как должны обрабатываться полученные данные перед сохранением или экспортом. Каждый класс пайплайна отвечает за выполнение определенной задачи обработки данных — очистки, проверки или преобразования.

Классы пайплайнов выполняются последовательно, причем выход одного пайплайна передается в качестве входа в следующий по порядку.

Item pipelines — это важнейший компонент архитектуры Scrapy.

Как работают Item pipelines?

Процесс веб-скрейпинга в Scrapy состоит из нескольких этапов: от отправки запросов на целевой сайт и извлечения данных с помощью пауков до обработки собранных данных с помощью Item pipelines.

Когда паук Scrapy извлекает данные с сайта и создает элемент, этот элемент проходит через Item pipelines в порядке, указанном в настройках Scrapy.

Каждый класс Item pipelines обрабатывает элемент определенным образом, например очищает данные, проверяет поля или преобразует данные в другой формат. Затем обработанный элемент передается следующему Item pipelines в последовательности. Если конвейер отбрасывает элемент (решает не обрабатывать его дальше), элемент не передается последующим конвейерам.

Когда элемент проходит через все Item pipelines, он либо сохраняется, либо экспортируется в зависимости от конфигурации в настройках Scrapy.

Элемент в Scrapy — это структура данных Python, которая используется для хранения соскрейпленных данных с веб-сайта. Обычно это класс, который наследуется от класса Item в Scrapy и определяет набор полей, соответствующих данным, которые вы хотите извлечь с целевого сайта. Каждое поле представляет собой интересующую часть информации, например заголовок, URL изображения, описание или любые другие важные данные.

В общем, элемент в Scrapy — это контейнер для хранения структурированных данных, которые вы извлекаете с сайта с помощью паука. Это способ организовать и представить собранные данные в виде объекта Python, который может быть обработан Item pipelines и механизмами хранения Scrapy.

Встроенные классы FilesPipeline и ImagesPipeline

В Scrapy есть встроенные классы для эффективного решения задач, связанных со скрейпингом фотографий и файлов — FilesPipeline и ImagesPipeline.

FilesPipeline — это конкретная реализация Item pipelines, встроенного в Scrapy.

FilesPipeline — это встроенный класс Scrapy, который автоматизирует процесс загрузки файлов с сайтов. Он может работать с разными форматами файлов и управляет процессом загрузки файлов за вас, обеспечивая правильное и последовательное их хранение.

FilesPipeline позволяет указать место, где будут храниться загруженные файлы: в локальной файловой системе или на облачном сервисе. Кроме того, пайплайн хранит метаданные файлов, такие как пути к файлам и URL-адреса загрузки, что позволяет отслеживать собранные данные.

ImagesPipeline — это версия FilesPipeline, ориентированная на работу с файлами изображений.

Подобно FilesPipeline, ImagesPipeline автоматически загружает файлы изображений с указанных URL-адресов.

ImagesPipeline предоставляет встроенные функции для обработки загруженных изображений, такие как создание иконок, изменение размера или преобразование в разные форматы. Как и FilesPipeline, ImagesPipeline позволяет настроить место хранения загруженных изображений и хранить метаданные.

Crawl Spider

Сегодня мы познакомимся с типом паука, который называется «краулер» (Crawl Spider).

Мы уже создавали паука с помощью команды scrapy genspider, которая использует базовый шаблон. В этом шаблоне у нас был класс, который наследуется от класса Spider. Этот класс должен иметь набор свойств, таких как name, allowed_domains, список start_urls или метод start_requests. И, наконец, он должен иметь, по крайней мере, один метод parse().

Вся эта структура и представляет собой шаблон.

```
class CountriesSpider(scrapy.Spider):
   name = 'countries'
   allowed_domains = ['tradingeconomics.com']
   start_urls =
['https://tradingeconomics.com/country-list/inflation-rate?contin
ent=world']

   def parse(self, response):
       countries = response.xpath("//td/a")
```

Кроме базового шаблона, в Scrapy есть три других. Чтобы увидеть все доступные шаблоны, выполним в терминале команду:

```
scrapy genspider -l
```

```
(GeekBrain) petrrubin@MacBook-Pro-Petr trading_economics % scrapy genspider -l
Available templates:
  basic
  crawl
  csvfeed
  xmlfeed
```

- basic — базовый шаблон, Scrapy использует его по умолчанию.
- crawl — с ним мы будем работать сегодня.
- csvfeed — шаблон для скрейпинга CSV-файлов.
- xmlfeed — шаблон для скрейпинга XML-документов.

Перейдем к реализации краулера. На этом занятии будем работать с сайтом zebrs.com/categories/smartphones.

Наша задача — извлечь все ссылки на все смартфоны, открыть каждую из них, а затем извлечь фото, название и цену товара.

Пайплайн изображений требует Pillow 7.1.0 или более поздней версии. Он используется для создания иконок и нормализации изображений в формат JPEG/RGB.

Сначала создадим проект с помощью команды scrapy startproject. Назовем его zebrs:

```
scrapy startproject zebrs
```

Теперь перейдем в директорию zebrs — cd zebrs. Сгенерируем паука с помощью команды scrapy genspider и укажем шаблон, который хотим использовать. Для выбора шаблона указываем аргумент -t, а затем его имя. Также зададим адрес целевой страницы сайта (его можно изменить позже):

```
scrapy genspider -t crawl zebrs_imgs www.zebrs.com
```

Переходим в VS Code и в меню откроем наш проект: File → Open Folder...

Теперь посмотрим файл паука, которого мы только что создали. Откроем zebrs_imgs.py.

На этот раз у нас есть класс паука, который наследуется от класса CrawlSpider, поэтому он отличается от предыдущих пауков, с которыми мы работали.

У нас все еще есть свойство name, allowed_domains, start_urls. Но к ним добавилось еще одно свойство — rules, которое представляет собой кортеж. Этот кортеж содержит один экземпляр класса Rule и определяет, как паук должен переходить по ссылкам и какая функция callback (обратного вызова) должна использоваться для парсинга содержимого посещаемых страниц.

У объекта Rule есть 3 аргумента:

- **Класс LinkExtractor** — это компонент Scrapy, который извлекает ссылки с веб-страницы.

  В нашем случае параметр allow имеет значение регулярного выражения (r'Items/'). Это значит, что LinkExtractor будет извлекать только те ссылки, которые содержат подстроку 'Items/' в своем URL.

- **Callback** — параметр обратного вызова определяет, какую функцию паук должен использовать для анализа содержимого посещенных страниц.

  В нашем примере паук будет вызывать функцию 'parse_item', когда встретит ссылку, соответствующую регулярному выражению в LinkExtractor.

- **Параметр follow** имеет значение True. Это указывает, что паук должен продолжать рекурсивно следовать по ссылкам, извлеченным LinkExtractor, посещая и выполняя парсинг страниц до тех пор, пока не будет найдено ни одной подходящей ссылки.

Кроме аргумента allow, у нас есть аргумент deny. В этом случае он значит: если ссылка содержит слово Items, пожалуйста, не переходите по ней.

Другой способ определить, какие ссылки следует извлекать или переходить по ним, — использование выражений XPath или селекторов CSS.

Таким образом, вместо использования аргументов allow и deny мы можем установить restrict_xpaths — это XPath (или список XPath), определяющий области внутри ответа response, из которых должны быть извлечены ссылки. Например, если мы определим этот аргумент как следующее выражение: //a[@class='active'], то это значит, что нужно следовать всем элементам «a», у которых атрибут class равен ‘active’. При этом не нужно добавлять /@href, потому что объект экстрактора ссылок будет автоматически искать атрибут href.

Здесь же вместо выражений XPath можно использовать CSS-селекторы, но вместо restrict_xpaths используется аргумент restrict_css, в котором мы определяем набор CSS-селекторов.

Перейдем к веб-странице в Chrome. Вспомним: наша задача — извлечь все ссылки на все смартфоны, открыть каждую из них, затем извлечь фото, название и цену товара.

Первое, что нам нужно сделать, — написать явное выражение, которое будет получать ссылки на все товары.

Откроем инструмент разработчика (Ctrl + Shift + I). Выберем блок, содержащий первый смартфон, найдем тег <a>. Он внутри тега , содержащего класс class='position-relative teaser-item-div'. Напишем XPath-выражение для выбора ссылки на страницу смартфона:

```
//div[@class='position-relative teaser-item-div']/a
```

Мы получили 18 элементов, что соответствует количеству смартфонов на странице. Скопируем это выражение XPath, добавим в код и внутрь объекта LinkExtractor, а также поменяем значение переменной start_urls на ссылку на целевую страницу:

```
class ZebrsImgsSpider(CrawlSpider):
    name = 'zebrs_imgs'
    allowed_domains = ['www.zebrs.com']
    start_urls = ['https://www.zebrs.com/categories/smartphones']

    rules = (

Rule(LinkExtractor(restrict_xpaths=("//div[@class='position-relat
ive teaser-item-div']/a")), callback='parse_item', follow=True),
    )
```

Далее внутри метода parse_item сначала удалим содержимое и добавим функцию print, чтобы посмотреть результат скрейпинга:

```
    def parse_item(self, response):
        print(response.url)
```

Возвращаемся в терминал и запускаем паука:

```
scrapy crawl zebrs_imgs
```

В ответ мы должны получить список ссылок на все страницы со смартфонами.

Далее в методе parse_item пишем следующий код:

```
    def parse_item(self, response):
        loader = ItemLoader(item=ZebrsItem(), response=response)
        loader.default_input_processor = MapCompose(str.strip)
```

loader = ItemLoader(item=ZebrsItem(), response=response) — эта строка создает экземпляр класса ItemLoader — вспомогательного класса, предоставляемого Scrapy, чтобы облегчить заполнение элементов данными, полученными в результате скрейпинга. Параметр item устанавливается в экземпляр класса ZebrsItem, а параметр response — в текущий объект ответа. Указывая эти два параметра, мы говорим ItemLoader использовать этот конкретный экземпляр элемента для хранения данных и извлекать данные из этого конкретного ответа. Разумеется, еще нужно импортировать данные классы:

```
from scrapy.loader import ItemLoader
from ..items import ZebrsItem
```

Теперь разберем строку loader.default_input_processor = MapCompose(str.strip). default_input_processor — это атрибут ItemLoader, определяющий входной процессор по умолчанию, который будет использоваться для всех полей элемента.

Входные процессоры — это функции, которые получают необработанные извлеченные данные и обрабатывают их перед сохранением в полях элемента. В нашем случае в качестве входного процессора используется функция MapCompose, которая применяет ряд функций (в данном случае только str.strip) к извлеченным данным. Функция str.strip используется для удаления из извлеченных данных пробелов, идущих впереди и позади.

Ее также нужно импортировать:

```
from itemloaders.processors import MapCompose
```

Следующий шаг — получение данных по каждой ссылке. Для этого вернемся в браузер и откроем первую ссылку на смартфон.

Напишем XPath-выражение для извлечения названия. Название находится внутри тега <h1>. Пишем XPath-выражение для получения текста:

```
//h1/text()
```

Далее переносим это выражение в код.

```
loader.add_xpath('name', '//h1/text()')
```

Эта строка добавляет извлеченный текст в поле name элемента ZebrsItem.

Следующее — цена. Обратите внимание, что для некоторых товаров цены указаны со скидками. Поэтому цена без скидки и цена со скидкой в HTML-коде имеют разный путь XPath. Чтобы решить эту проблему, можно написать условие: пробуем получить текущую цену и, если значение переменной окажется пустой строкой, то делаем скрейпинг по другому выражению XPath:

```
def parse_item(self, response):
        loader = ItemLoader(item=ZebrsItem(), response=response)
        loader.default_input_processor = MapCompose(str.strip)

        loader.add_xpath('name', '//h1/text()')

        price_text_danger = response.xpath('//div[@class="me-2
product-price"]/span[@class="text-danger"]/text()').get()
        if price_text_danger:
            loader.add_value('price', price_text_danger)
        else:
            loader.add_xpath('price', '//div[@class="me-2
product-price"]/text()')
```

Наконец, находим в HTML-коде ссылку на изображение и пишем соответствующее XPath-выражение:

```
relative_image_urls = response.xpath('//div[@class="text-center
d-none d-sm-block dsktp-zoomer"]/ul/li/img/@src').getall()
```

Обратите внимание, что мы должны использовать метод getall(), чтобы получить список ссылок.

Мы получаем относительные пути к изображениям, поэтому с помощью метода urljoin преобразуем наши относительные URL в абсолютные:

```
absolute_image_urls = [urljoin("https://www.zebrs.com", img_url)
for img_url in relative_image_urls]
loader.add_value('image_urls', absolute_image_urls)
```

Напомним: функция urljoin принимает два аргумента — базовый URL ("https://www.zebrs.com") и относительный URL (например, "/path/to/image.jpg"). Она объединяет эти два URL для создания абсолютного URL. И используем генератор списка, который перебирает все элементы списка relative_image_urls, применяя к каждому функцию urljoin, и строит новый список, содержащий абсолютные URL-адреса изображений.

Конечно, нужно импортировать библиотеку:

```
from urllib.parse import urljoin
```

Наконец, добавим yield:

```
        yield loader.load_item()
```

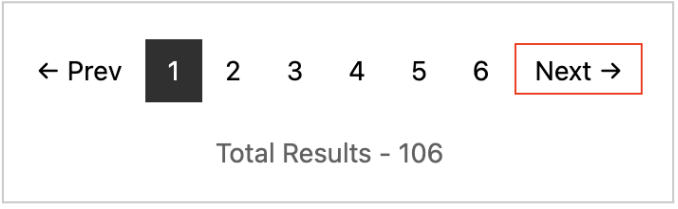
Важно! Название поля image_urls должно быть именно таким, нельзя называть его по-другому. А также absolute_image_urls должно быть списком абсолютных ссылок, а не относительных.

Дополнительно мы можем использовать файл items.py. Давайте пропишем в него все поля, которые мы собираемся скрейпировать:

```
class ZebrsItem(scrapy.Item):
    "title" = scrapy.Field()
    "price" = scrapy.Field()
    "image_urls" = scrapy.Field()
    "images" = scrapy.Field()
```

Здесь важно прописать дополнительно поле “images” — это обязательное условие для скрейпинга изображений.

Теперь посмотрим, как мы можем добраться до всех остальных страниц, чтобы выполнить скрейпинг всех картинок. Внизу страницы мы видим, что у нас есть кнопка Next.



Воспользуемся инструментом поиска и кликнем на ссылке Next.

Как видно, у нас есть элемент <a> с атрибутом rel=”next”, напишем соответствующее XPath-выражение:

```
//a[@rel='next']
```

Скопируем это выражение XPath и добавим его в наш код. Добавим еще один объект Rule. Внутри LinkExtractor добавим restrict_xpaths, содержащее наше выражение. Далее мы можем установить follow = True. Однако, поскольку мы не собираемся указывать метод callback, мы можем опустить этот аргумент, потому что значение true установится по умолчанию.

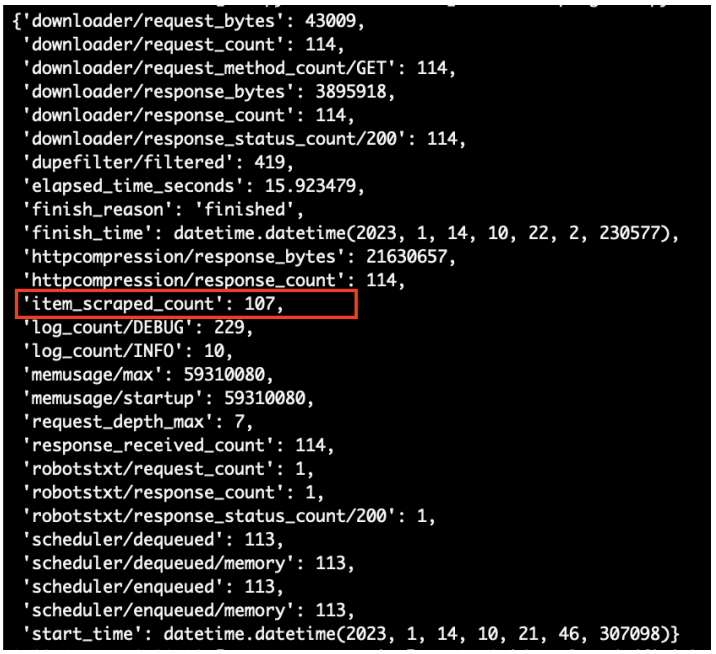
```
   rules = (

Rule(LinkExtractor(restrict_xpaths=("//div[@class='position-relat
ive teaser-item-div']/a")), callback='parse_item', follow=True),
       Rule(LinkExtractor(restrict_xpaths=("//a[@rel='next']")))
   )
```

Важная вещь, которую следуем иметь в виду: порядок следования объектов Rule имеет значение. Если мы поменяем местами Rule, и второй поставим на первое место, то он попытается сразу посетить следующую страницу (вторую), а первую пропустит.

Причина, по которой во втором объекте Rule не нужно указывать метод callback: когда мы посещаем каждую следующую страницу, первый объект Rule будет вызываться автоматически. Поэтому метод parse_item будет вызван и нет необходимости вызывать его снова.

Сохраняем код и запускаем из терминала. Видим, что item_scraped_count = 107, то есть мы выполнили скрейпинг 107 объектов.



Что еще нужно сделать — прописать User Agent пользователя. Это можно сделать в файле settings.py либо присвоив новое значение переменной USER_AGENT:

```
# Crawl responsibly by identifying yourself (and your website) on the
user-agent

#USER_AGENT = 'zebrs (+http://www.yourdomain.com)'
```

А если нужно изменить несколько headers запроса, то это можно сделать в DEFAULT_REQUEST_HEADERS.

```
# Override the default request headers:

#DEFAULT_REQUEST_HEADERS = {

#   'Accept':
'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',

#   'Accept-Language': 'en',

#}
```

Один из способов узнать свой User Agent — просто загуглить в Chrome «my user agent». Так и сделаем. Скопируем значение и вставим в значение нашей переменной.

Последнее, что нужно сделать, чтобы начать скрейпинг изображений, — включить медиапайплайн. Чтобы включить медиапайплайн, нужно добавить его в файл настроек settings.py.

Здесь нужно найти параметр ITEM_PIPELINES, откомментировать его и для скрейпинга изображений использовать следующее значение:

```
ITEM_PIPELINES = {'scrapy.pipelines.images.ImagesPipeline': 1}
```

Для скрейпинга файлов используется следующий код:

```
ITEM_PIPELINES = {'scrapy.pipelines.files.FilesPipeline': 1}
```

Вы также можете одновременно использовать пайплайн файлов и пайплайн изображений.

Затем настроим параметр целевого хранилища, то есть путь, по которому будут сохраняться картинки. Иначе пайплайн останется отключенным, даже если вы включите его в параметр ITEM_PIPELINES.

Для пайплайна изображений установим параметр IMAGES_STORE — можно прописать его в самом низу файла:

```
IMAGES_STORE = '/path/to/valid/dir'
```

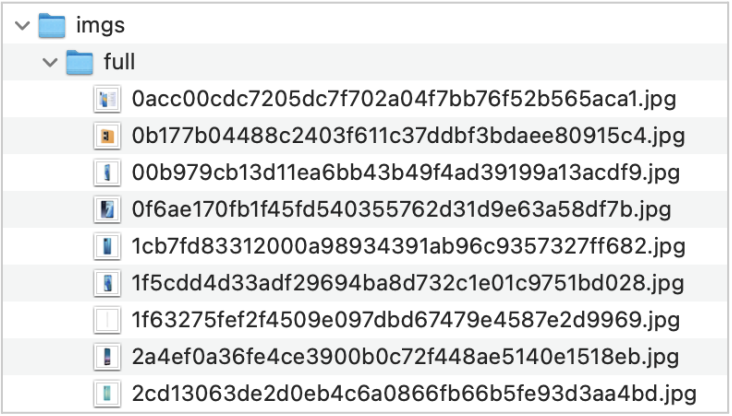
Для пайплайна файлов устанавливается параметр FILES_STORE:

```
FILES_STORE = '/path/to/valid/dir'
```

Можно запустить краулер и вывести результат в CSV-файл:

```
scrapy crawl zebrs_img -L WARN -o products.csv
```

После завершения работы краулера посмотрим папку с изображениями.



Изображения получены, но проблема в том, что имена файлов представляют собой URL-хэш. Это неудобно для использования. Давайте пока удалим папку imgs со скачанными файлами.

В VS Code открываем файл pipelines.py и удаляем все, что в нем есть. Нам нужно написать собственный пайплайн.

Импортируем метод ImagesPipeline:

```
from scrapy.pipelines.images import ImagesPipeline
```

Давайте рассмотрим метод file_path, который в scrapy работает по умолчанию.

```
def file_path(self, request, response=None, info=None, *, item=None):
       image_guid = hashlib.sha1(to_bytes(request.url)).hexdigest()
       return f'full/{image_guid}.jpg'
```

В этом коде request.url — это URL изображения, которое мы получаем. Этот URL конвертируется в байтовое представление, а затем, с помощью библиотеки hashlib и метода hexdigest(), преобразуется в строковый объект двойной длины, содержащий только шестнадцатеричные цифры. Иными словами — каждый URL преобразуется в sha1 — хэш, который уникален так же, как уникален URL.

Теперь, если мы хотим изменить способ именования файлов, нужно изменить эту строку. Важная вещь — то, что один из аргументов file_path — item, элемент. Это то, что мы скрейпируем и передаем в yield. В нашем случае: “title”, “price”, “image_urls”.

Например, если бы мы хотели использовать в качестве имени файла “title”, то взяли бы item[“title”]:

```
def file_path(self, request, response=None, info=None, *,
item=None):
       item["title"]
```

Редактируем наш код:

```
import hashlib
from scrapy.pipelines.images import ImagesPipeline

class CustomImagesPipeline(ImagesPipeline):
   def file_path(self, request, response=None, info=None, *,
item=None):
       image_guid =
```

```
hashlib.sha1(request.url.encode()).hexdigest()
        return f"{item['name']}-{image_guid}.jpg"
```

В примере создан пользовательский класс под названием CustomImagePipeline, который наследуется от ImagesPipeline Scrapy.

Метод file_path использует поле name элемента и SHA1-хэш URL изображения для создания уникального имени файла. Вы можете дополнительно настроить его, используя любую информацию из объектов элемента или ответа, чтобы создать имя файла, соответствующее вашим потребностям.

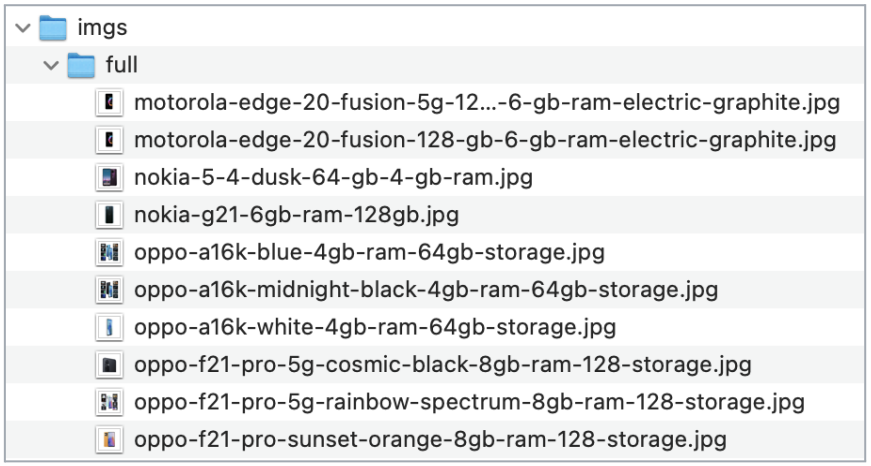
Строка image_guid = hashlib.sha1(request.url.encode()).hexdigest() генерирует уникальный идентификатор для каждого изображения на основе его URL. Цель создания уникального идентификатора — избежать конфликтов имен при сохранении изображений. Давайте разберем эту строку шаг за шагом:

- request.url — это URL обрабатываемого изображения. Метод encode() вызывается для строки URL, чтобы преобразовать ее в число. Это необходимо, поскольку функция хэширования, используемая на следующем этапе (hashlib.sha1()), ожидает в качестве входных данных числовой объект
- Функция hashlib.sha1() принимает на вход закодированный URL и создает новый объект хэша SHA-1. SHA-1 — это широко используемая криптографическая хэш-функция, которая создает 160-битное (20-байтовое) хэш-значение. Цель использования хэш-функции — создать фиксированное по размеру, уникальное и детерминированное представление входных данных (в данном случае URL изображения).
- Метод hexdigest() вызывается на объекте хэша SHA-1 для преобразования хэш-значения в шестнадцатеричную строку. Длина полученной строки — 40 символов. Это уникальный идентификатор изображения (image_guid).

Последнее — в файле настроек нужно отредактировать строку ITEM_PIPELINES:

```
ITEM_PIPELINES = {'zebrs.pipelines.CustomImagesPipeline': 1}
```

Сохраняем изменения, переходим в терминал и запускаем краулер. После окончания процесса проверим результат. Как видно, имена файлов представлены в виде названия продуктов:



Еще одна вещь — в результате скрейпинга мы получаем также URL каждого изображения. Если мы хотим, чтобы в выдаче не было URL-адресов, то в файле settings.py используем параметр FEED_EXPORT_FIELDS для определения полей для экспорта, их порядка и выходных имен.

Например, если мы хотим получить в выдаче только название продукта и его цену, то определяем параметр следующим образом:

```
FEED_EXPORT_FIELDS = ['title', 'price']
```

При этом элементы в списке можно менять местами.

У скрейпинга файлов та же логика, что и у скрейпинга изображений.

Заключение

В завершении лекции поговорим о ресурсах и следующих шагах, которые помогут расширить знания и получить новые навыки в области веб-скрейпинга и извлечения данных.

Официальная документация Scrapy — это бесценный ресурс, где можно узнать больше о функциях и возможностях фреймворка. Документация охватывает разные аспекты Scrapy, включая установку, учебники, продвинутые темы и ссылки на API.

Когда вы освоите основы Scrapy, изучите продвинутые темы — работа с сайтами на JavaScript, использование прокси-серверов и развертывание пауков на облачных платформах — Scrapy Cloud или Scrapinghub.

Освоив эти темы, вы станете опытным веб-скрейпером, сможете решать сложные задачи по веб-скрейпингу и извлекать ценные данные из разных источников.

Что можно почитать еще?

1. Hidden APIs with Scrapy — easy JSON data extraction
2. Item Loaders in Scrapy
3. Downloading and processing files and images — Scrapy 2.10.0 documentation
4. How to download Files with Scrapy ? — GeeksforGeeks