Web Scraping with Scrapy

- **Name:** Aditya N Bhatt
- **College:** Manipal School of Information Sciences
- **Branch:** AI & ML
- **Roll No:** 231057015
- **Year:** 2023-24

Table of Contents

1. Overview Summary
2. Case 1: Single-Page Web Scraping
    - Objective
    - Project Structure
    - Items Code (items.py)
    - Pipeline Code (pipelines.py)
    - Settings Code (settings.py)
    - Spider Code (amazon_spider.py)
    - Conclusion
3. Case 2: Single Page Search Results Web Scraping
    - Objective
    - Project Structure
    - Items Code (items.py)
    - Pipeline Code (pipelines.py)
    - Settings Code (settings.py)
    - Spider Code (amazon_search_spider.py)
    - Conclusion
4. Case 3: Multiple Page Search Results Web Scraping
    - Objective
    - Project Structure
    - Items Code (items.py)
    - Pipeline Code (pipelines.py)
    - Settings Code (settings.py)
    - Spider Code (amazon_multiple_pages_spider.py)
    - Conclusion
5. How to Contact Me

Overview Summary

This report provides a comprehensive guide to web scraping using Scrapy, a powerful and versatile web crawling and scraping framework. The report covers three main cases, each demonstrating different aspects of web scraping and Scrapy implementation.

Case 1: Single-Page Web Scraping

Project Structure (items, pipelines, settings, spider)

In this section, we define the project structure for Case 1, which includes the items, pipelines, settings, and spider components.

Items Code (items.py):

```
import scrapy

class AmazonItem(scrapy.Item):
    title = scrapy.Field()
    price = scrapy.Field()
```

Explanation: In the `items.py` file, we define a custom item class called `AmazonItem`. This class represents the structure of the data we want to scrape. It contains two fields: "title" and "price." These fields will be used to store the scraped data.

Pipeline Code (pipelines.py):

```
class AmazonPipeline:
    def process_item(self, item, spider):
        # Perform data processing or validation here, if needed
        return item
```

Explanation: The `pipelines.py` file contains a custom pipeline class called `AmazonPipeline`. Pipelines are used to process scraped items. In the `process_item` method, you can implement data processing or validation logic for each scraped item before it's saved or further processed.

Settings Code (settings.py):

```
BOT_NAME = 'amazon_scraper'
FEED_EXPORT_ENCODING = 'utf-8'
NEWSPIDER_MODULE = 'amazon_scraper.spiders'
ROBOTSTXT_OBEY = True
SPIDER_MODULES = ['amazon_scraper.spiders']

# Configure pipeline
ITEM_PIPELINES = {
    'amazon_scraper.pipelines.AmazonPipeline': 300,
}
```

Explanation: In the `settings.py` file, we configure various settings for the Scrapy project. These settings include:

- `BOT_NAME` : The name of the Scrapy bot.
- `FEED_EXPORT_ENCODING` : Encoding for exported data (UTF-8 in this case).
- `NEWSPIDER_MODULE` : Module for new spiders.
- `ROBOTSTXT_OBEY` : Whether to obey the website's `robots.txt` rules.
- `SPIDER_MODULES` : List of spider modules.
- `ITEM_PIPELINES` : Configuration to specify the item pipeline and its priority.

Spider Code (amazon_spider.py):

```
import scrapy
from amazon_scraper.items import AmazonItem

class AmazonSpider(scrapy.Spider):
    name = 'amazon_spider'
    start_urls = ['https://www.amazon.in/Adidas-Mens-Regular-T-Shirt-H13881_Red/dp/B09NYKPR58/ref=sr_1_6?keywords=manchester+united+

    def parse(self, response):
        title = response.css('span#productTitle::text').get()
        price = response.css('span#priceblock_ourprice::text').get()

        item = AmazonItem()
        item['title'] = title.strip() if title else None
        item['price'] = price.strip() if price else None

        yield item
```

Explanation: The `amazon_spider.py` file contains the Scrapy spider class `AmazonSpider` . This class is responsible for web scraping. In the

`parse` method:

- We send an HTTP request to the specified URL.
- We use CSS selectors (`response.css()`) to extract the product title and price from the page.
- We create an `AmazonItem` object and store the extracted data in its fields.
- Finally, we use `yield` to pass the item through the configured pipeline for further processing.

Conclusion (Case 1)

In Case 1, we successfully created a Scrapy project for single-page web scraping from Amazon. We defined the item structure, set up a pipeline for data processing, and configured project settings accordingly. The spider was implemented to extract product titles and prices from the provided URL.

This project structure provides a foundation for more complex web scraping tasks, with the flexibility to customize data extraction and processing as needed.

Case 2: Single Page Search Results Web Scraping

(Items, Pipeline, Settings, and Spider)

In this section, we describe the key components and code for Case 2, which involves scraping data from a single page of search results on Amazon.

Project Structure (items, pipelines, settings, spider)

Items Code (items.py):

```
import scrapy

class AmazonItem(scrapy.Item):
    title = scrapy.Field()
    price = scrapy.Field()
```

Explanation: In the `items.py` file, we define a custom item class called `AmazonItem`, which specifies the structure of the data to be scraped. This class contains two fields: "title" and "price," where we will store the scraped information.

Pipeline Code (pipelines.py):

```
class AmazonPipeline:
    def process_item(self, item, spider):
        # Perform data processing or validation here, if needed
        return item
```

Explanation: The `pipelines.py` file contains a custom pipeline class called `AmazonPipeline`. Pipelines are responsible for processing scraped items. The `process_item` method allows us to apply data processing or validation logic to each item before further processing.

Settings Code (settings.py):

```
BOT_NAME = 'amazon_scraper'
FEED_EXPORT_ENCODING = 'utf-8'
NEWSPIDER_MODULE = 'amazon_scraper.spiders'
ROBOTSTXT_OBEY = True
SPIDER_MODULES = ['amazon_scraper.spiders']

# Configure pipeline
ITEM_PIPELINES = {
    'amazon_scraper.pipelines.AmazonPipeline': 300,
}
```

Explanation: In the `settings.py` file, we configure various project settings, including:

- `BOT_NAME`: The name of the Scrapy bot.
- `FEED_EXPORT_ENCODING`: Encoding for exported data (UTF-8).
- `NEWSPIDER_MODULE`: Module for new spiders.
- `ROBOTSTXT_OBEY`: Whether to obey `robots.txt` rules.
- `SPIDER_MODULES`: List of spider modules.
- `ITEM_PIPELINES`: Configuration specifying the item pipeline and its priority.

Spider Code (amazon_search_spider.py):

```
import scrapy
from amazon_scraper.items import AmazonItem

class AmazonSearchSpider(scrapy.Spider):
    name = 'amazon_search_spider'
    start_urls = ['https://www.amazon.in/s?k=manchester+united+jersey']

    def parse(self, response):
        products = response.css('div.s-main-slot > div')

        for product in products:
            title = product.css('span.a-size-base-plus a-color-base a-text-normal::text').get()
            price = product.css('span.a-price span.a-offscreen::text').get()

            item = AmazonItem()
            item['title'] = title.strip() if title else None
            item['price'] = price.strip() if price else None

            yield item
```

Explanation: The `amazon_search_spider.py` file contains the Scrapy spider class `AmazonSearchSpider`. In the `parse` method:

- We send an HTTP request to the specified search results URL.
- We use CSS selectors (`response.css()`) to extract product titles and prices from the search results page.
- We iterate through each product, create an `AmazonItem` object, and store the extracted data in its fields.

- Finally, we use `yield` to pass each item through the configured pipeline for further processing.

Conclusion (Case 2)

In Case 2, we successfully created a Scrapy project for single-page web scraping from Amazon search results. We defined the item structure, set up a pipeline for data processing, and configured project settings accordingly. The spider was implemented to extract product titles and prices from the search results page for Manchester United jerseys.

This project structure serves as a foundation for more complex web scraping tasks, with the flexibility to customize data extraction and processing as needed.

Case 3: Multiple Page Search Results Web Scraping

(Items, Pipeline, Settings, and Spider)

In this section, we describe the key components and code for Case 3, which involves scraping data from multiple pages of search results on Amazon.

Project Structure (items, pipelines, settings, spider)

Items Code (items.py):

```
import scrapy

class AmazonItem(scrapy.Item):
    title = scrapy.Field()
    price = scrapy.Field()
```

Explanation: In the `items.py` file, we define a custom item class called `AmazonItem`, specifying the structure of the scraped data. This class contains two fields: "title" and "price," where we will store the scraped information.

Pipeline Code (pipelines.py):

```
class AmazonPipeline:
    def process_item(self, item, spider):
        # Perform data processing or validation here, if needed
        return item
```

Explanation: The `pipelines.py` file contains a custom pipeline class called `AmazonPipeline`. Pipelines are responsible for processing scraped items. The `process_item` method allows us to apply data processing or validation logic to each item before further processing.

Settings Code (settings.py):

```
BOT_NAME = 'amazon_scraper'
FEED_EXPORT_ENCODING =

 'utf-8'
NEWSPIDER_MODULE = 'amazon_scraper.spiders'
ROBOTSTXT_OBEY = True
SPIDER_MODULES = ['amazon_scraper.spiders']

# Configure pipeline
ITEM_PIPELINES = {
    'amazon_scraper.pipelines.AmazonPipeline': 300,
}
```

Explanation: In the `settings.py` file, we configure various project settings, including:

- `BOT_NAME`: The name of the Scrapy bot.
- `FEED_EXPORT_ENCODING`: Encoding for exported data (UTF-8).
- `NEWSPIDER_MODULE`: Module for new spiders.
- `ROBOTSTXT_OBEY`: Whether to obey `robots.txt` rules.
- `SPIDER_MODULES`: List of spider modules.
- `ITEM_PIPELINES`: Configuration specifying the item pipeline and its priority.

Spider Code (amazon_multiple_pages_spider.py):

```
import scrapy
from amazon_scraper.items import AmazonItem

class AmazonMultiplePagesSpider(scrapy.Spider):
    name = 'amazon_multiple_pages_spider'
    start_page = 1
    end_page = 7  # Number of pages to scrape
    base_url = 'https://www.amazon.in/s?k=manchester+united+jersey&page={page}&ref=sr_pg_{page}'

    def start_requests(self):
        for page in range(self.start_page, self.end_page + 1):
            url = self.base_url.format(page=page)
            yield scrapy.Request(url, callback=self.parse)

    def parse(self, response):
        products = response.css('div.s-main-slot > div')

        for product in products:
            title = product.css('span.a-size-base-plus a-color-base a-text-normal::text').get()
            price = product.css('span.a-price span.a-offscreen::text').get()

            item = AmazonItem()
            item['title'] = title.strip() if title else None
            item['price'] = price.strip() if price else None

            yield item
```

Explanation: The `amazon_multiple_pages_spider.py` file contains the Scrapy spider class `AmazonMultiplePagesSpider`. In this spider:

- We define the range of pages to scrape (from `start_page` to `end_page`).
- In the `start_requests` method, we generate multiple page URLs and send requests for each page.
- In the `parse` method, we extract product titles and prices from each page of search results. The spider is designed to scrape a specified range of pages, in this case, from page 1 to 7.

Conclusion (Case 3)

In Case 3, we successfully created a Scrapy project for multiple-page web scraping from Amazon search results. We defined the item structure, set up a pipeline for data processing, and configured project settings accordingly. The spider was implemented to extract product titles and prices from a range of search results pages for Manchester United jerseys.

This project structure allows for efficient scraping of data from multiple pages and can be adapted for various web scraping scenarios.

How to Contact Me

If you have any questions, need further assistance, or would like to get in touch with me, please feel free to reach out through the following contact methods:

Email: You can email me at adityab24840@gmail.com for any inquiries or assistance related to the report or the provided Scrapy projects.

LinkedIn: You can connect with me on LinkedIn at Aditya N Bhatt for professional networking and discussions.

GitHub: To access the Scrapy project codes and other technical resources, you can visit my GitHub repository at adityab24840.

I am always happy to assist you and answer any questions you may have. Your feedback and inquiries are highly valued.