Scrapy Beginners Series Part 2: Cleaning Dirty Data & Dealing With Edge Cases

In **Part 1** of this **Python Scrapy 5-Part Beginner Series** we learned how to build a basic scrapy spider and get it to scrape some data from a website.

Web data can be messy, unstructured, and have lots of edge cases. So it is important that your Scrapy spiders are robust and deal with messy data.

So in **Part 2: Cleaning Dirty Data & Dealing With Edge Cases** we're going to show you how to make your Scrapy spiders more robust and reliable:

Need help scraping the web?

Then check out **ScrapeOps**, the complete toolkit for web scraping.

Proxy Manager   Scraper Monitoring   Job Scheduling

Python Scrapy 5-Part Beginner Series

- **Part 1: Basic Scrapy Spider -** We will go over the basics of Scrapy, and build our first Scrapy spider. (Part 1)
- **Part 2: Cleaning Dirty Data & Dealing With Edge Cases -** Web data can be messy, unstructured, and have lots of edge cases. In this tutorial we will make our spider robust to these edge cases, using Items, Itemloaders and Item Pipelines. (This Tutorial)
- **Part 3: Storing Our Data -** There are many different ways we can store the data that we scrape from databases, CSV files to JSON format, and to S3 buckets. We will explore several different ways we can store the data and talk about their Pro's, Con's and in which situations you would use them. (Part 3)
- **Part 4: User Agents & Proxies -** Make our spider production ready by managing our user agents & IPs so we don't get blocked. (Part 4)
- **Part 5: Deployment, Scheduling & Running Jobs -** Deploying our spider on a server, and monitoring and scheduling jobs via ScrapeOps. (Part 5)

The code for this project is available on Github here!

Strategies to Deal With Edge Cases

Unfortunately for us, web data is often messy and incomplete which makes the web scraping process a bit more complicated.

For example, when scraping **e-commerce websites**, whilst most products follow a specific data structure in certain circumstances data will being displayed differently:

- There might be a normal price, and a sales price.
- The price might be displayed with sales taxes or VAT included in some circumstances, however, excluded in others.
- If the product is sold out, the products price might be missing.
- Some product descriptions might be in paragraphs, whilst others might be bullet points.

Dealing with edge cases like this is a fact of life when scraping, so we need to come up with a way to deal with it.

In the case of the Chocolate.co.uk website we are scraping for this series, if we inspect the data we can see a couple issues:

- Some products don't display prices if the product is **out of stock** or the price is missing.
- The product prices are in pounds, but we would like them in US Dollars.
- The product urls are relative URLs, not absolute URLs.
- Some products are duplicated.

```
## csv file

name,price,url
100% Dark Hot Chocolate Flakes,£8.50,/products/100-dark-hot-chocolate-flakes
70% Dark Hot Chocolate Flakes,£8.50,/products/70-dark-hot-chocolate-flakes
Almost Perfect,'<span class=""price"">
                <span class=""visually-hidden"">Sale priceFrom £2.50',/products/almost-perfect
Assorted Chocolate Malt Balls,£8.95,/products/assorted-chocolate-malt-balls
Blonde Chocolate Honeycomb,£8.95,/products/blonde-chocolate-honeycomb
Blonde Chocolate Honeycomb - Bag,£7.50,/products/blonde-chocolate-sea-salt-honeycomb
Dark Chocolate Malt Balls,£8.95,/products/dark-chocolate-malt-balls
Dark Chocolate Orange Peel,£8.95,/products/dark-chocolate-orange-peel
Dark Chocolate Truffles,£17.50,/products/dark-chocolate-truffles
Espresso,£4.95,/products/espresso-coffee-dark-chocolate-bar
Gift Voucher,"From £10.00
",/products/gift-voucher
Gin & Tonic,£4.95,/products/gin-and-tonic-chocolate-bar
```

There are a number of options to deal with situations like this:

Options	Description
Try/Except	You can wrap parts of your parsers in **Try/Except** blocks so if there is an error scraping a particular field, it will can revert to a different parser.
Conditional Parsing	You can have your spiders check the HTML response for particular DOM elements and use specific parsers depending on the situation.
Item Loaders	Item Loaders allow you to clean & process data as you are parsing it.
Item Pipelines	With Item Pipelines you can design a series of post processing steps to clean, manipulate and validate your data before storing it.
Clean During Data Analysis	You can parse data for every relevant field, and then later in your data analysis pipeline clean the data.

Every strategy has it's pros & cons, so it is best to get familar with every method so you can choose the best option for your particular situation when you need it.

In this project, we're going to focus on using **Item Loaders** and **Item Pipelines** as they are the most powerful options available to us within Scrapy, and solve most data processing issues.

Organizing Our Data With Scrapy Items

Up until now we've been yielding our data in a dictionary. However, the preferred way of yielding data in Scrapy is using its **Item** functionality.

Scrapy Items are simply a predefined data structure that holds your data. Using Scrapy Items has a number of advantages:

- More structured way of storing data.
- Enables easier use of Scrapy Item Pipelines & Item Loaders.
- Ability to configure unit tests with Scrapy extensions like Spidermon.

So the next step we're going to do is switch to using Scrapy Items in our `chocolatespider`.

Creating an Item is very easy. Simply create a Item schema in your `items.py` file.

This file is usually auto generated when you create a new project using scrapy and lives at the same folder level as where you have the settings.py file for your scrapy project.

```
import scrapy

class ChocolateProduct(scrapy.Item):
   name = scrapy.Field()
   price = scrapy.Field()
   url = scrapy.Field()
```

Then in our `chocolatespider.py` file, import the Item schema and update our spider to store the data in the Item.

```
import scrapy
from chocolatescraper.items import ChocolateProduct

class ChocolateSpider(scrapy.Spider):

  #the name of the spider
  name = 'chocolatespider'

  #these are the urls that we will start scraping
  start_urls = ['https://www.chocolate.co.uk/collections/all']

  def parse(self, response):

      products = response.css('product-item')

      product_item = ChocolateProduct()
```

```
        for product in products:
            product_item['name'] = product.css('a.product-item-meta__title::text').get()
            product_item['price'] = product.css('span.price').get().replace('<span class="price">\n             <span class="visually-hidden">Sale price</span>','').replace('</span>','')
            product_item['url'] = product.css('div.product-item-meta a').attrib['href']
            yield product_item

        next_page = response.css('[rel="next"] ::attr(href)').get()

        if next_page is not None:
            next_page_url = 'https://www.chocolate.co.uk' + next_page
            yield response.follow(next_page_url, callback=self.parse)
```

Pre Processing Data With Scrapy Item Loaders

To take things one step further we are going to use Scrapy Item loaders.

Item Loaders provide an easier way to populate items from a scraping process.

Whereas in the previous section, we populated the Item directly. Scrapy Item Loaders provide a much more convenient mechanism for populating them during the scraping process, by automating some common tasks like parsing the raw extracted data before assigning it.

Example	Description
Removing Characters	Removing extra tags or special characters from the parsed data. For example, removing currency signs.
Type Coversion	Converting a **string** to an **int**.
URL Conversion	Changing URLs from relative to absolute URLs.
Combining Fields	Combine 2 or more pieces of information that scraped into one field.
Replacing Values	Replacing one value for another. For example, replacing a `$` sign for a `£` sign.
Unit Conversion	Converting the units of a value. For example, converting a string with `120 grams` to `0.12 KG`.
Appending Data	Adding a value to the front or end of an item value. For example, adding `"kilograms"` to the end of a number

Not only does using Item Loaders make cleaning your data easier, they also make the spider itself easier to read by moving all the xpath/css references to the Item Loader.

That's enough theory, let's create an Item Loader below and you'll understand better the usecases that item loaders are made to handle.

Example Item Loader

For our tutorial we are going to configure our Item Loader to:

1. Remove the `£` sign from the data we're scraping.
2. Convert the scraped **relative** urls to full **absolute** urls.

Remember that Item Loaders provide an easier way to populate the item without writing all our code to clean up the item inside of our spider. In this item loader we are going to just remove the `£` sign using the python split function inside as we don't need this in our data going forward.

To create our Item Loader, we will create a file called `itemsloaders.py` and define the following Item Loader:

```
from itemloaders.processors import TakeFirst, MapCompose
from scrapy.loader import ItemLoader

class ChocolateProductLoader(ItemLoader):

    default_output_processor = TakeFirst()
    price_in = MapCompose(lambda x: x.split("£")[-1])
    url_in = MapCompose(lambda x: 'https://www.chocolate.co.uk' + x )
```

Processors for specific fields are defined using the `_in` and `_out` suffixes for input and output processors. In our example, we are declaring input processors for both the `price` and `url` fields.

- **Price:** The `price` input processor will split any value passed to it on the `£` sign and use the second value.
- **Url:** The `url` input processor will append the passed **relative** url to base url.

Now that we have both our **Itemloader** and our **Item** defined we can combine them both in our spider which would then be updated to look like the following:

```
import scrapy
from chocolatescraper.itemloaders import ChocolateProductLoader
from chocolatescraper.items import ChocolateProduct


class ChocolateSpider(scrapy.Spider):

   # The name of the spider
   name = 'chocolatespider'

   # These are the urls that we will start scraping
   start_urls = ['https://www.chocolate.co.uk/collections/all']

   def parse(self, response):
       products = response.css('product-item')

       for product in products:
            chocolate = ChocolateProductLoader(item=ChocolateProduct(), selector=product)
            chocolate.add_css('name', "a.product-item-meta__title::text")
            chocolate.add_css('price', 'span.price', re='<span class="price">\n             <span class="visually-
hidden">Sale price</span>(.*)</span>')
            chocolate.add_css('url', 'div.product-item-meta a::attr(href)')
            yield chocolate.load_item()

       next_page = response.css('[rel="next"] ::attr(href)').get()

       if next_page is not None:
           next_page_url = 'https://www.chocolate.co.uk' + next_page
           yield response.follow(next_page_url, callback=self.parse)
```

As we can see in our above spider code. When we use **Item Loaders** and **Items** with Scrapy it forces us to simplify and clean up our code.

Using **Items** also reduces errors that we can make, as the spider won't run if for example, we have a spelling mistake in the name of one of our chocolate attributes - eg. `prce` instead of `price`.

Processing Our Data With Scrapy Item Pipelines

Now that we have our spider using the **Items** and **Item Loaders**, we will use **Item Pipelines** to manipulate the data we have scraped before saving it. (Data storage is covered in Part 3).

Using **Item Pipelines** we are going to do the following:

- Convert the `price` from a string to a float (so we can then multiply it by the exchange rate).
- Convert the `price` from British Pounds to US Dollars
- Drop items that currently have no price (due to being sold out).
- Check if an Item is a duplicate and drop it if it's a duplicate.

Converting The Price

Once an Item has been scraped by the spider, it is sent to the `**Item Pipeline**` for validation and processing.

Each Item Pipeline is a Python class that implements a simple method called `process_item`.

The `process_item` method takes in an Item, performs an action on it and decides if the item should continue through the pipeline or be dropped.

In the pipeline below we are going to take the **ChocolateProduct** Item, convert the price to a float, then convert the price from pounds sterling to dollars by multiplying the price scraped by the exchange rate.

The pipelines live inside the pipelines.py file in your project - at the same folder level as the `itemloaders.py` and `items.py` files we were working with above.

```
from itemadapter import ItemAdapter
from scrapy.exceptions import DropItem


class PriceToUSDPipeline:

    gbpToUsdRate = 1.3

    def process_item(self, item, spider):
        adapter = ItemAdapter(item)

        ## check is price present
        if adapter.get('price'):

            #converting the price to a float
            floatPrice = float(adapter['price'])

            #converting the price from gbp to usd using our hard coded exchange rate
            adapter['price'] = floatPrice * self.gbpToUsdRate

            return item

        else:
            # drop item if no price
            raise DropItem(f"Missing price in {item}")
```

Now that we have our price converted to dollars lets setup another pipeline to remove any duplicates.

Removing Duplicates

To remove duplicate **ChocolateProduct** Items we will be checking the name of the product and if the name is already present then we will drop the Item (not returned to our output).

This second pipeline class also goes into the same `pipelines.py` file in your project.

```
from itemadapter import ItemAdapter
from scrapy.exceptions import DropItem


class DuplicatesPipeline:

    def __init__(self):
        self.names_seen = set()

    def process_item(self, item, spider):
        adapter = ItemAdapter(item)
        if adapter['name'] in self.names_seen:
            raise DropItem(f"Duplicate item found: {item!r}")
        else:
            self.names_seen.add(adapter['name'])
            return item
```

Enabling Our Pipelines

To enable the **PriceToUSDPipeline** and **DuplicatesPipeline**we just created, we need to add it to the `ITEM_PIPELINES` settings in our `settings.py` file, like in the following example:

```
ITEM_PIPELINES = {
    'chocolatescraper.pipelines.PriceToUSDPipeline': 100,
    'chocolatescraper.pipelines.DuplicatesPipeline': 200,
}
```

The integer values you assign to classes in this setting determine the order in which they run. Items go through from lower valued to higher valued classes. It's customary to define these numbers in the 0-1000 range.

Testing Our Data Processing

When we run the our spider we should see the all the chocolates being crawled with the price now displaying in dollars after our Item Loaders have cleaned the data, and the Item Pipeline has converted the price data and dropped any duplicates.

One line of the output should look something like this:

```
DEBUG: Scraped from <200 https://www.chocolate.co.uk/collections/all>
{'name': 'Blonde Chocolate Honeycomb',
 'price': 11.63,
 'url': 'https://www.chocolate.co.uk/products/blonde-chocolate-honeycomb'}
```

And we can see that we are dropping duplicated Items:

```

 'item_dropped_count': 8,
 'item_dropped_reasons_count/DropItem': 8,
 'item_scraped_count': 74,
```

Next Steps

We hope you have enough of the basics to use items, item loaders and item pipelines in your future projects! If you have any questions leave them in the comments below and we'll do our best to help out!

The next tutorial covers how to store your data using the feed exporters. Since there are many places we may need to send our data be that to a S3 Bucket, a database, a file. We'll cover several of these in Part 3 - Storing Our Data.

Need a Free Proxy? Then check out our **Proxy Comparison Tool** that allows to compare the pricing, features and limits of every proxy provider on the market so you can find the one that best suits your needs. Including the best free plans.