How not to get lost in asynchronous Scrapy world.

**Introduction (problem statement)**

One of the typical tasks is to scrape a big aggregator site, which contains tens or even hundreds of thousands records of contractors or firms data, for example. Such big sites often have the following features:

- search capability input box

- categories (of contractors or firms) pages

- firm listing page

- paging links (on firm listing pages)

- individual firm pages

Throughout the article, we will use a fancy “Who supplies what” aggregator site example

In this article, we will cover the topics that are not usually highlighted in the mans:

1. How to maintain persistency across scraping sessions

2. How to establish a good reporting during (and between) sessions.

*What do we mean by persistency and why it is important.*

If we’re talking about scraping a site, which contains hundreds of thousands records, **it is unlikely that we will do all our job in a single session**. Several reasons may be for that – perhaps we want to inspect the portion of the data collected before proceeding further (which is definitely a good practice!) or we know that, if we exceed some amount of requests the site will temporary ban us so we need to stop and recharge the session after waiting some time, etc.

That is why we need to maintain persistency in our scraper - to be able to start exactly from the point where we finished before. This way we need the mechanisms to handle the following use cases:

- skip already processed categories

- skip already processed pages within firm listing of a category

- avoid duplicates in scraped items

Of course, we must consider the situation when we suddenly broke our session on whatever step of our process. Never can we expect that data or webpages are always even and therefore we may expect at every point an error, which we could not ever imagine. Most likely, we will get a sudden stop while processing a page, which can be illustrated with the following diagram:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Category 1 | | | | | | | | | | | | | | | | | | | | Category 2 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Category 3 | | | | | | | | | | | | | | | | | | |
| Page 1 | | | | Page 2 | | | | | Page 3 | | | | | Page 4 | | | | | | Page 1 | | | | | | | Page 2 | | | | | | | Page 3 | | | | | | | Page 4 | | | | | | | Page 5 | | | | | | | Page 1 | | | | | Page 2 | | | | | Page 3 | | | | | Page 4 | | | |
|  |  |  |  | |  |  |  |  | |  |  |  |  | |  |  |  |  |  | |  |  |  |  |  |  | |  |  |  |  |  |  | |  |  |  |  |  | 1 | | 2 | 3 | 4 | 5 | 6 |  | |  |  |  |  |  |  | |  |  |  |  | |  |  |  |  | |  |  |  |  | |  |  |  |

Suppose we broke down here

(last saved item is 3)

Based on this picture, when we respawn our scraping session, obviously we should:

- entirely skip Category 1 – this is what the first use case is about

- skip pages from 1 to 3 of the Category 2 – this is the second use case

- avoid duplicates of firms 1, 2, 3 when we started from uncompleted Page 4 – this is the use case 3.

The latter use case – duplicates filter – every scraper usually implements, because almost no customer would like to have duplicates in their data.

The importance of a good reporting is evident. The best thing we may expect from reporting is to try to predict when we finish, maybe not in terms of hours and minutes, but just how much work we did not complete yet. But scraper reporting is a consequence of persistency – we are already collecting a lot of “job state” information to maintain the former, so basically it is our choice what job state information to show and where we are not starting to be too verbose.

**Overview of the things needed to implement persistency and reporting.**

If we would like our scraper to do things mentioned in the Problem Statement we need to know when to save some changes to the scraping job state and, as a counterpart, to know in which way to load job state at session restart. This leads to the following two big groups:

A. Mechanisms to recognize when a specific condition occurs.

B. Mechanisms to load a saved job state to start scraper from the point we finished before.

For most efficiency, I prepared an overview of the mechanisms and actions needed to implement them (I omit the phrase “save to database” in section A because it is evident). Next, we will review them one by one.

|  |  |
| --- | --- |
| **A. Recognize when a specific condition occurs** | **B. Load saved job state** |
| A.1. Trigger when a page is complete | B.1. Skip the page if it is already processed |
| a. Obtain knowledge how many items are on the page.  b. Know, from which page and category an item came from.  c. Count items somewhere in a “category-page-items count” table. If the count reaches total on the page, than page is completed. | a. Check if the page item came from is in the page seen list for this category.  b. If yes, mark rules from that page to individual firms as blocked – in order do not extract such links and do not emit requests. |
| A.2. A category is completed | B.2. Do not feed the completed categories into start\_requests of the spider |
| a. Discover what the last page number is.  b. (for reporting only) Report a “category completed” message when the amount of pages completed is equal to the last page number. | a. When querying categories related database table, check if the amount of pages in page\_seen field is equal to last\_page number and filter out such records. |
| A.3. Skip duplicates in case a firm occurs several times on different categories or pages | B.3. Skip duplicates when scraping previously uncompleted page |
| a. Check against ids seen table to watch out if an item duplicated somewhere else | a. Load already stored firm IDs  b. Mark item as duplicated and bypass this empty item directly to pipeline (because we need to count it anyway – otherwise the “page completed” trigger will not work)  c. Mark the request to this item details to be skipped – thus avoid the request from being yielded. |

Before proceed to the steps description, we must clarify two points, which concerns almost every step from the above table: packet labeling and concept of Rules.

*Packet labeling – how to forward information from one request/response to another.*

For example, consider the step A.1.b. “Know, from which page and category an item came from”. How we are going to know that? Why this question is not trivial?

Actually, we cannot save such information (category, page) in our spider instance fields - please remember that Scrapy world is completely asynchronous – seemed very easy to get lost among flows of numerous "packets". Every request/response would save to these fields in a random order so we will have there anything, except actual data!

How to deal with this issue in our unpredictable world?

The idea behind is the same like in computer networks - every packet is traveling across the net independently among milliards of them, but every one has special labels so routers or computers alongside the path know how to handle a packet and every recipient knows how to reassemble them into a message exactly the same as sender wanted to tell.

Thanks God, Scrapy developers provide special dictionary, request.meta and response.meta exactly for this purpose. There we can put whatever information we need. Quite common practice, advised by the manual, is to put there reference to Item class instances.

To keep out job state information grouped, we will put all the information related to the job state of our scraper in request.meta['job'] subdictionary. Let’s call this trick **“packet labeling”** for further discussions.

*Rules: out-of-the-box way to extract links to walk on the site*

When we are crawling on the site, we use site hyperlinks for that. Therefore we need, as a first step of our scraping, to collect them somehow. Scrapy developers already have done this – they provided us convenient thing called CrawlSpider. This is a subclass of a “plain” spider, which introduces very important concept of **Rules**.

We just specify css or xpath criteria to find hyperlinks (thus define a **Rule**) and CrawlSpider extracts appropriate hyperlinks and emits corresponding requests. Rules play important role in realization of the mechanisms mentioned above and we will see them in details later on.

Now we are ready to discuss each step in full detail.

**A.1.a. Obtain knowledge how many items are on the page.**

Sometimes sites indicate amount of items on page explicitly. But what if not? There is an elegant technique to do so via Scrapy functionality. Since we are using CrawlSpider, corresponding Rule link extractor collects all the links to individual items for us. Why not to count them?

The only thing is for that we need to override Scrapy’s standard library method \_requests\_to\_follow of CrawlSpider class in site-packages/scrapy/spiders/crawl.py file. The overridden method is just copy-and-paste of standard one, but with two insertions to extend the functionality. Please do not touch now the first extension – check if Rule is deactivated, - we will discuss it later.

The whole listing of this overridden method follows.

def \_requests\_to\_follow(self, response):

if not isinstance(response, HtmlResponse):

return

seen = set()

*# added functionality - check if rule is deactivated*

switchOff = response.meta.get('switchedOffRule', -1)

for n, rule in enumerate(self.\_rules):

if n != switchOff: *# added functionality – do if rule is not deactivated*

links = [lnk for lnk in rule.link\_extractor.extract\_links(response)

if lnk not in seen]

if links and rule.process\_links:

links = rule.process\_links(links)

*# added functionality - set amount of links got in the response*

linksGot = len(links)

response.meta['job']['linksGot'] = linksGot

*# added functionality ends*

for link in links:

seen.add(link)

r = self.\_build\_request(n, link)

yield rule.process\_request(r)

You see that this method is a core of CrawlSpider functionality. It gets html Response, extracts links in a list and emits new Requests for these links.

One subtle question is: in our \_requests\_to\_follow we have access and can do “packet labeling” only to Response object, but we do not control outgoing Requests. How to pass meta information to them?

Remember that all Requests the spider emits are passed through spider middleware process\_spider\_output method, which has access to both Reguest and the Response, inspired by that request. Key feature in Scrapy is that from Request to its resulting Response all meta information passed automatically (inside the framework), but form Response to new Requests does not. We have to do it manually:

def process\_spider\_output(self, response, result, spider):

for i in result:

if isinstance(i, Request):

i.meta['job'].update(response.meta['job'])

But why it is so? To answer, we will touch here a little bit an essence of Scrapy. The goal of the framework is to harvest web pages and collect necessary information from them. For this, the framework emits requests, receives responses and produces either the Items or new requests.

<picture>

We see that any single request always produces single response as his product. Therefore they are strictly coupled, resulting response always has a link to source request and meta information is copied. But what will be done after response is received – is a spider decision. It can either forward a response for parsing thus produce an Item, or it can push it through the screen of Rules and emit secondary requests. Thus, no strict connection exists between response and further requests, therefore to pass or not to pass meta info is upon developer’s decision.

We see that as a result of the techniques above, the amount of firma links contained on a page is always available for us.

**A.1.b. Know, from which page and category an item came from**

We already saw that we can transfer housekeeping meta information across request-response chains via “packet labeling” technique. And this is perfect, because some pieces of information appear on a different stages of crawling. Here we will see examples of this.

One such piece of information is the page number. All my experience shows, that in 90% of the cases when sites provide information with paging, they use an URL of the form: www.whosupplieswhat.com/lists/category?query=print&page=2, where page number exists as URL argument after “?”. How to use it: when link extractor finds next-page link, it spawns the request. But before this request submitted to middleware, Scrapy provides a hook called process\_request, on which we can hang a request treatment procedure. The procedure takes a request as an argument and must return it back when done. We simply extract page number from request URL and stuff it into the request meta.

class MySpider(CrawlSpider):

def setPage(request):

*# to regexp page number from URL here*

*# …*

request.meta['job'] = {'page': page}

return request

rules = ( …

# go to next page

Rule(LinkExtractor(restrict\_xpaths='<xpath>'), process\_request=setPage)

… )

Rule callback process\_request and “packet labeling” are good recipes to get an early-stage metadata from request URL.

Same technique may be applied to extract category name.

**A.1.c. Count items somewhere in a “category-page-items count” table. If the count reaches total on the page, than page is completed**

Count items somewhere, but where exactly? The word “somewhere” from the title introduces an important concept of spider **extensions**. There are a lot of about them in the Scrapy documentation, the idea behind is: “in case you need some outside functionality – write an extension.”

How the requests from Rules put in a queue!!

By default, Scrapy CrawlSpider uses deep-first traversal. Last in – first out. Therefore if we want do discover the last page as soon as possible, **put paging rule at the bottom of the rules list**.

Count item processed

We know from which page and category item came from

We know how many items are on page.

if firmasOnPage == item['linksGot']

means page is completed

When the completed page is the last page in a category.

There is no “next page” link – that means we are on the last page of the firma listing

While scraping a big aggregator site often it is good idea to have clear reporting about

what your scraper is doing at the moment and

from which query and category the data it scraped came.

Suppose we have an aggregator site with search input box. We enter some pattern, "print" for example and the site responded with let's say "refined" categories it contains, which may relate to our word "print":

Printing machines

3-D printing

Textile print

Offset print

... etc

Next, every category links to an index page, which contains a listing of the firms related to that category.

Last step is to go to each firm's individual page and grab all data related to this firm.

But what if the customer asks to save, within each firm record also an initial query which leaded to this firm ("print" in our example), along with category name (“Printing machines" in our case)? That would make sense. Or, it may be some information exists on an index page (total amount of firms per category or something else), which is also worth to save in a data output.

For better understanding, let's recall how Scrapy would work on our project and where we can catch the flows of requests/responses to keep track on them.

We will subclass not from "plain" spider but from CrawlerSpider, which deals with any sort of "index pages" in a more convenient way.

The crawler reads start\_urls value

It makes initial request to start url

The response received is passed to link extractors for eating

If a rule of link extractor triggers, the spider emits a request for this link.

Now, if we specified a callback procedure - that means that we reached the target firm details page and can create information item and fill it with data scraped.

Opposite, if callback did not set, we assume that a new page requested by the link is again a some sort of an index page. The scraper then follows the link and again apply the set of its link extractor rules to the response.

start\_urls = [www.example.com/querypage?q=print]

Request

Response - refined categories page

Rule 0: on the category page extract links to index pages;

Rule 1: on the index page extract links to individual firms;

Rule 2: on the index page extract links to next index page.

Rule 0 triggers - request to index page is emited, with label meta['rule'] = 0

We receive a response with index page and again, we feed it to our 3 Rules

This time, Rule 1 triggers - request to individual firm page is emmited with with label meta['rule'] = 1

Because we set a callback for Rule 1, the response is transferred to data scraping procedure. No more requests will spawn from this chain, we've got data!

(Optional) In case the index is spreaded over multiple pages, Rule 2 triggers, next index page fetched, where again the process with Rule 1 - Rule 2 repeats.

You can imagine how huge the amount of chains may be!

If page already seen - add special flag to response, which will prohibit Rule 1 from being activated.

For totals count and for-updates scraping: table which item id was seen on which category and page. Ids allowed to duplicate. Later on, during update scraping, spider will load a page with seen elements for every active chain.