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 (measures to record persistency)** | **B. Load saved job state (measures to load and check persistency 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. So 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.

**B.1.a. Organize a page seen storage**

The straightforward way to store processed pages is to create separate table where primary key is category id and another field is a page number itself. But first, I do not like to add extra complexity to our database and second, we have lack of visualization and lose the overview in this case. Instead, I just have string field in categories table, where processed pages are represented by numbers comma separated, like ‘1,2,3,5,14’.

Then, to load it into list, we will do something like this:

list(map(lambda x: int(x), pagesAsText.split(',')))

The values in our text string are always sorted ascending. To ensure this, we perform sorting when adding a new page seen value:

pageSeenList = <assign to this variable the list from job state table>

pageSeenList.append(page)

pageSeenList.sort()

<assign to (overwrite) job state table> = pageSeenList # because the list is

# modified by sorting

outputForDb = ','.join(map(lambda x: str(x), entry))

Now to check that the response received aimed for the page already processed is as easy as doing “in” operator against pages seen list.

**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.

**B.1.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**

All responses appeared are the results of some Rules in our spider. Every response carries the Rule number, because of which it was born. The spider must always walk across the pages, but it should not crawl to firm details on pages, which were already processed. Therefore, we must never block “paging requests” but should block firm details requests.

For crawling spiders emit requests (which is the spider output) and receive responses (which is the spider input). The spider middleware has corresponding methods process\_spider\_output and process\_spider\_input.

Now the picture – spider receives response, forward it to rules (link discovering system) and outputs requests for rules triggered. The idea behind is to prohibit operation of rule to firm details. Remember we modified (overrode) in A.1.a the standard library method \_requests\_to\_follow: we added the following fragment:

*# 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*

*# rest of the code*

With this we effectively suppress unwanted requests.

Now let’s touch the logic how to set switchedOffRule field. As new requests are evaluated based on a response, evidently we should introduce this code in the process\_spider\_input. We check if response rule number indicates a category page and set switchedOffRule field to the number we are going to deactivate:

def process\_spider\_input(self, response, spider):

seen = self.jobState.ifPageSeen(category, page)

if seen:

response.meta['switchedOffRule'] = <nomber of rule to deactivate>

return None

**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 custom functionality – write an extension.”

Personally I implemented via extension:

- all interaction of spider with the database

- all reporting messages

- a table with firm IDs which are already in the database (duplicates handling)

- “category-page-items count” table which is named jobState

The structure of the latter table is the following

{category: {pageSeen: [], pages: {1:4, 2:46, ...}, last: 2, total: 50},  
 category: {pageSeen: [], pages: {5:8, 3:77, ...}, last: 3, total: 88},  
 .....  
}

where:

pageSeen – list of all completed pages;

pages – a dictionary where keys are page numbers and values are firm counters. Each time item has came, the pipeline signals to extension, which in turn increases the counter and returns new value.

This is the point where pipeline compares the updated on-page counter with the total of firms on that page and if yes, adds the processed page to the pageSeen list. I did it in the pipeline like this:

nm = item['nameInUrl']

pg = item['page']

*# register item in pages seen*

firmasOnPage = self.jobState.increaseOnPageCounter(nm, pg)

**if** firmasOnPage == item['linksGot']:

self.jobState.addPageSeen(nm, pg)

Here self.jobState is the pointer to the extension (remember all stuff is implemented in the extension) and addPageSeen procedure actually does the job.

As Scrapy is able to create multiple concurrent asynchronous requests, it could be happen that we will process several pages within a category at the same time. Therefore instead of providing counter for one pages, we must orgainze an array of page counters. The field last of our structure will be discussed later.

**A.2.a. Discover what the last page number is**

When walking across pages our spider uses “next page” links. It “clicks” such links again and again until it reaches the last page. There, “last page” link will be disabled and we will use the fact that our link extractor would be unable to find it. REFACTOR: This could be easily done in \_requests\_to\_follow method, which we are already familiar with. I will provide the method fragment:

# *TODO remove the last page recognition somewhere in middleware*  
# added functionality - if no "next page links" found, means  
# that response's page is the last one  
respRule = response.meta.get('rule', -32)  
**if** (n == 0) **and** (respRule **in** [-1, 0]):  
 **if not** links:  
 name = response.meta['job']['nameInUrl']  
 pg = response.meta['job']['page']  
 self.jobState.thisIsTheLastPage(name, pg)

We must discuss here very important point regarding sequence in which Scrapy fires requests. It is good idea if we can detect the last page of a category as early as possible. This way we should force Scrapy to walk across pages first and do actual scraping job only after that. From the documentation we know that by default Scrapy performs deep-first DOM tree traversing. One solution is to alter this behavior to breadth-first search but I think this way the framework will fetch the first items from every page, than second items from every page and so on. In fact, I did not do such alteration and used another approach instead.

How deep-first search works in essence? It uses LIFO buffer which is actually a stack to store requests. Now suppose for simplicity that we have two Rules in our CrawlSpider rules tuple:

rule 0: find links to individual firm pages, go there, parse them and produce items;

rule 1: find “next page” link and go to the next page.

We are on the first page. At first, rule 0 requests spawned and stuffed into the stack (scheduled). Next, rule 1 link found and corresponding request put in the stack too. Now, the first popped request is to go to page 2. We go there and the process repeats. You see that this way we are doing mostly next-page requests, while firm page requests are still waiting to be popped1.

Therefore the motto is: “**put the next-page rule on the last position**!”

The “feature” of this behavior, due to nature of a stack is when we reached the last page, Scrapy starts to produce items also from the last page and thus yields them in reversal order (in terms of pages). It does not matter from database point of view, but when your customer would like to check your work, he will definitely look into the first page on website to compare with your production output. It is not good if we have to explain him that first-page data are somewhere on the bottom and please go to the last page on the site or scroll down.

Keep this issue in mind, supply page number field to your database and sort the production output accordingly to avoid unnecessary discussions.

**B.2.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**

The title of this section is self-explanatory. What I’m going to touch here is how to “inject dependencies” and how to feed the spider with start requests – we must switch on the starter to turn our engine on.

There is a standard way described in documentation to plug the extension in our system.

@classmethod

def from\_crawler(cls, crawler):

ext = cls(crawler)

crawler.signals.connect(ext.spider\_opened, signal=signals.spider\_opened)

crawler.signals.connect(ext.spider\_closed, signal=signals.spider\_closed)

return ext

The method grabs Crawler instance and connects signals. In turn, we inject to spider the link to our extension:

def spider\_opened(self, spider):

spider.jobState = self # inject the link

# open a database connection

# load jobState structure

# load ids\_seen table

This makes all functionality of our extension available to spider.

The spider\_closed method is used to properly close database connection when we are done.

**A.2.b. (for reporting only) Report a “category completed” message when the amount of pages completed is equal to the last page number**

In the introduction I mentioned that a reporting is just a consequence of persistence – for this step we just need to emit a message when processed pages counter is equal to the last page number. All the mechanisms for this we already discussed above.

I put this logic in addPageSeen method discussed in A.1.c like this:

**if** len(<pageSeen list>) == self.jobState[nameInUrl]['last']:

logger.info('Category "{0}" completed'.format(nameInUrl))

**A.3.a. Check against ids seen table to watch out if an item duplicated somewhere else**

In this very step nothing special to discuss because the technique is perfectly described in the documentation (duplicates pipeline). More interesting and sophisticated is how to skip duplicates when scraping previously uncompleted page. This is a subject of use case B.3 so let’s proceed to the next section B.3.a.

**B.3.a. Load already stored firm IDs**

It is nothing to do to create a query from database and load the result into memory. More interesting is to shortly touch how to obtain the IDs. The site you are parsing also has a backend, also has a database therefore item IDs are available. In most cases. Like in section A.1.b, the ID is available as an argument to URL. So again I hooked up a procedure to process\_request of a corresponding Rule and filled request meta field.

**B.3.b. Mark item as duplicated and bypass this empty item directly to pipeline**

I already mentioned in the overview table that even if an item is a duplicate, we must register it anyway. Otherwise all our bookkeeping to detect page completion will not work. Seemed like we have already done this job in A.3.a, but let us think of the following:

- It is good idea to suppress unnecessary requests

- If we remove requests, corresponding Item would not be produced. Straightforward idea is to place additional procedure call to count these unborn items in place where we killed the request. But this is not good, because DRY! Our duplicate detection system is placed in pipeline and we should do the all the job just in one place. What should we do is to produce special blank item.

Because we get some housekeeping information in early stage – page, category name, firm ID – it seems logic to create items already in middleware and then pass it further to be fulfilled with payload. I will provide process\_spider\_output method in whole:

**def** process\_spider\_output(self, response, result, spider):  
 **for** i **in** result:  
 **if** isinstance(i, Request):  
 i.meta['job'].update(response.meta['job'])  
 willRequestByRule = i.meta.get('rule')  
 **if** willRequestByRule == 1:  
 firmaId = i.meta['job']['firmaId']  
 bi = MyItem(dict(firmaId=firmaId,  
 nameInUrl=i.meta['job']['nameInUrl'],  
 page=i.meta['job']['page'],  
 linksGot=i.meta['job']['linksGot'],  
 isDuplicate=0)  
 )  
 **if** self.jobState.ifItemExists(firmaId):  
 bi['isDuplicate'] = 1  
 i.meta['discard'] = 1  
 **yield** bi  
 **else**:  
 i.meta['item'] = bi  
 **if not** i.meta.get('discard'):  
 **yield** i  
 **else**:  
 **yield** i

By if willRequestByRule == 1 we check that request arrived is aimed to firm details page. Here we create an Item and assume by default that it is not a duplicate (isDuplicate=0).

But if it actually is, besides setting duplicate field for Item, we must also mark the request. Why we cannot delete the request right now is because the method is designed based on **yield** keywords and everything would not work if we do it another way.

This way an item will be counted properly and we will not break the DRY principle.