Adding Websites for Retail Assistant to Scrape

Vadim Torgashov

2020

1 Requirements

This project requires the use of these Python libraries: Beautiful Soup, Requests, psutil, pyuser-agent, free-proxy, and PyInstaller. This was also made with Python 3.7, if you use Python 2 your mileage may vary.

2 Checking the Robots.txt

You should always scrape only when given expressed permission by the website owner. The easiest way to check for that is by going on the robots.txt of the website. Simply take the default URL like "www.ebay.com" and add "/robots.txt" on the end. The section you want to check is under "User-Agent", under it will be a list of routes labeled as "Allow" or "Disallow". If the only thing under "User-Agent" is "Disallow: /" or "Disallow: /*" then that means absolutely no crawling or scraping of the website is permitted and you should respect their wishes. However if crawling is allowed on some directories then look at what directory a normal search goes to in order to see if scraping any results page is allowed. However the later sections of this document describe how to get the best URL for the scraper to use. Once you get this URL you should double check to assure that path is also permitted.

3 How to add the website to the GUI

In the file "RetailerAssistant.py" go to variable "retailer_list" at line 19 and add the name of the website as a string to that list. Be aware of exactly how you write the site's name as it will need to match what the scraper uses.

4 How to add the website's webstyle

A webstyle is the database entry that informs the scraper how to find necessary information on a webpage. The file "webstyle_generator.py" is prepared to add, test, and finalize the webstyle entry for your desired website. To begin with you want to make sure the variable "testing" at line 11 is set to True. This assures you run the test and your database entry will be added to a separate file from the actual webstyle database. If you need to reset the default webstyle database delete "webstyles.db" and run "restore_webstyles.py". The main area of concern is the collection of variables going from line 23 to line 46. Firstly set the "retailer name" variable at line 24 to the same string as what you added to the "retailer_list" in the GUI. This includes identical spacing, capitalization, and punctuation. You must now assign the link to variable "website" at line 23.

4.1 Setting the link

To get the best link you should search for an item, set the department/category to "all" (if the website categorizes by departments like Ebay and Amazon), sort by newest items added, and set the items per page to the largest number, and put it on page 2 (some websites have a different URL once you go past page 1). Once you have your link you should isolate where the search term is in the link and replace it with "%s". Then find where the page number is and replace it with "%d". Here is a validly formatted Ebay link as an example: "https://www.ebay.com/sch/i.html?nkw=%s&sop=10&ipg=200&pgn=%d". When you take that link you can replace the "%s" with whatever you want to search for and replace the "%d" with the page number the resulting link should take you to the correct page. One thing to note is that the program simply increases the page number that is inside the link. A lack of a page number in the link causes the program to simply scrape one page and nothing else. Which may or may not work for your needs. If you still want to progress through the result pages without a "%d" in the link then its up to you to create the code needed to traverse the website's result pages.

4.2 Setting up the HTML elements

The remaining variables are all for HTML element names and how far the program has to look. If the HTML element for something like the price has a name or id then assign that name as the "*_html" variable. The format to assign any HTML element is "element,id_type:name". With the element being something like a 'span', the id type is an 'id','class', etc., and the name is what the id type is assigned as. One example would be: "span,class:s-item price". The 'extra_html' variable is used for any unique elements an item listing can have, such as Amazon items that are labeled as Prime. The "*_depth" variables are to only be used if the desired element has no name or id. You can select a parent element that is named and designate how many subelements the program must traverse to find the desired element. The parent element is not counted in the depth count and neither is the desired element. The count is done linearly so if a subelement has its own sub elements they will also be counted. Leave the depth count at 0 if the desired element is the one assigned to its "_html" variable. This can be a bit difficult to setup without good testing so use the file "depth_tester.py" to see if your non-zero depth will work. If an element is not present or not to be included simply leave the variable as "ex". There are far too many variables on purpose. It provides all the options someone would want to search for, but likely won't use all for any given website.

4.3 Running the test

Once you have filled out all the fields you want to scrape you can change the variable "test_search" on line 21 to an item you want to search for on the site. Run the program. The first thing that is printed out should be the link used when scraping the website to make sure the program is formatting is correctly. Then the first scraped 10 listings are printed without formatting so that you can compare and make needed adjustments. Then all formatted listings are printed. Check to make sure what has been scraped is accurate to the real page. If any changes need to be made simply modify the necessary variables. If the parsing that is done by the program is incorrect the remedy to that will be covered in an upcoming section.

4.4 Finalizing the website

Once a successful test has been ran the "testing" variable on line 11 must be switched to False and the program reran again. At this point you can delete the file "data/webstyles_test.db".

5 How to parse incoming data

Some websites might have unique methods of showing their data and coding in every website's formatting is impossible. For starters: the current way data is parsed assumes listing links are an href, any links that start with "/" will be appended to the website's default URL (such as appending "/itm..." to "www.ebay.com"), and getting most data simply involves looking in between the ">" and "<" of any given element. On top of any unique formatting a website may have some items may have unique formatting different than whats normally on the site. Ebay for instance will change the structure of the title element when an item has the label "New Listing". To handle any unique formatting you must go to the file "RetailAssistantScraper.py" and make a subclass of the class "CustomFilter". The name of the class must be: "custom_" + retailer name + "_filter". The class only requires one staticmethod called "filter". The function is to receive the element list formatted by the existing program and the original unmodified list, in that order. This way you can look at how the program formatted the elements and reparse them yourself using the raw unparsed data. Make sure to return the modified list. I would suggest simply taking the reformatted element and overwriting it in the "filtered_listing" list, rather than reformatting the entire "unfiltered_listing" list (unless everything is to be refiltered). If you want an example of all of this refer to the class "custom_Ebay_filter" at line 155.

6 How to edit the listing before adding to database

Once all the data has been correctly parsed and put into a list you can do last minute modifications before it is stored. Such as adding characters you want to be displayed in the GUI, moving certain data to a different field, etc. This process is very similar to what was discussed in the previous section on adding a custom

filter. In the file "RetailAssistantScraper.py" you must create a subclass of the class "CustomFormatting". The class must be named as such: "custom_" + retailer name + "_formatting". There is only one required staticmethod called "format" that takes in the listing. Make sure the function "format" returns the listing array. For a working example refer to the class "custom_Ebay_formatting" on line 425.

7 Testing

All of these additions should be tested before any executables are made. One can simply run "RetailerAssistant.py" and "RetailerAssistantScraper.py". The UI will notify you that there's a lack of an executable scraper. However, everything will still work to test your code. Once a scrape is done and all the listings are present with correct formatting then your code is functional and ready.

8 Putting the whole thing together

Go into the folder where all the .py files are saved using command line. Run these commands to produce the needed .exe files: "pyinstaller –onedir –noconsole RetailerAssistantScraper.py" and "pyinstaller –onefile –windowed RetailerAssistant.py". For optimization purposes RetailerAssistantScraper generates an entire folder. That folder goes in place of RetailerAssistantScraper.py. Due to an issue with PyInstaller the pyuser-agent library isn't automatically put in the executable directory. The folder data/pyuser_agent is to placed within the RetailerAssistant folder. Check the Retailer Assistant repository to see exactly how everything should look.