Programming in Python:

Web Scraping Project

Bo J. Taylor

Western Governors University

Programming in Python: Web Scraping Project

The objective of this assessment is to scrap all hyperlinks embedded in the hypertext markup language (HTML) of the United States (US) Census Bureau Population and Housing Unit Estimates page. The final product is a comma separated value (CSV) file of all unique Uniform Resource Locators (URL) with absolute paths.

My first step was to retrieve the HTML content to be parsed (Figure 1). To do that, I used both the OS and Requests libraries. The using the OS library, my script determines if the HTML file already exists in the current working directory via the os.path.isfile() function. If the file does not, then the content is retrieved using functions provided by the Requests library. The HTML file is then constructed through the .open(), .iter\_content(), and .write() functions to open a new file, iterate over a specified number of bytes of the page’s data, and write that data to the local HTML file, respectively.

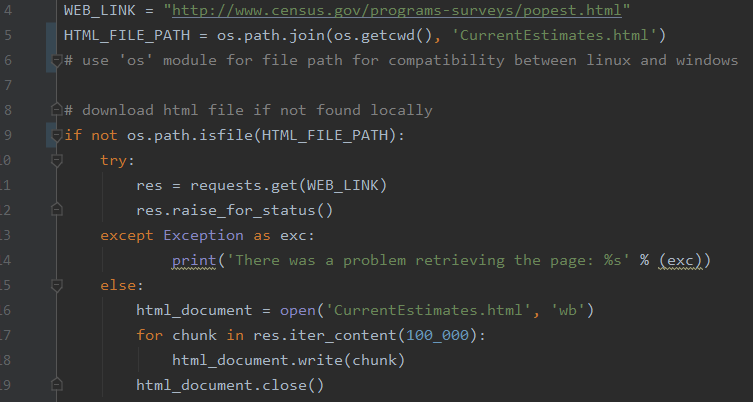


Figure 1. HTML retrieval. This figure contains a snapshot of the code responsible for fetching the website’s HTML content and storing it locally.

After downloading the HTML content to a local file, I then created a BeautifulSoup object. The BeautifulSoup (BS) library provides tools to simplify parsing and working with HTML (Sweigart, 2015). Creating a BS object was as simple as passing the file to the BS constructor and indicating the content to be parsed was HTML.

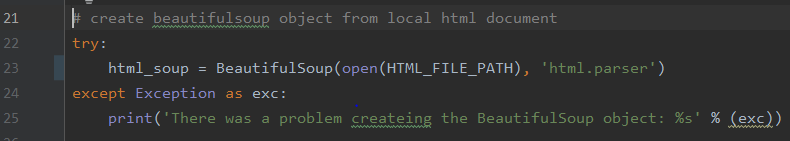


Figure 2. BeautifulSoup object creation. This figure contains code handling the initialization.

Once the BS object was created, I then generated a list of all links in the page. Using the .find\_all() function, I iterated through all the “a” HTML tags, retrieving the URL value with the .get() function. The Urllib library provides a simple function that handles absolute and relative URL’s. By passing the source HTML URL and the URL of each site in this iteration loop, I used this function to append only absolute URL’s to the list of links.

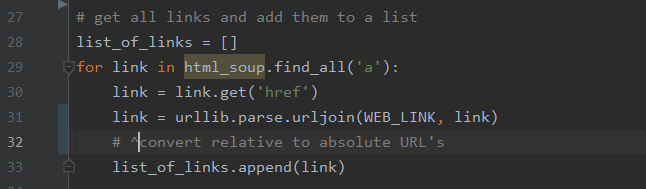


Figure 3. Link parsing. This figure illustrates handling relative URL’s while all links are parsed.

In order to remove any duplicate values, I used the set() function. This function is passed a list of variables, and returns a list of only unique values. Since the urljoin() function in the previous step did not return potentially hidden duplicates with a trailing backslash in the URL, I did not have to take additional steps to modify URL’s prior to passing them to the set() function. I did, however, need to remove the hash value trailing at the end of any URL’s. I used a regular expression to identify any link ending with a trailing i.d. tag. Links without a tag were appended to a list of filtered links, unaltered. Links that did have a trailing i.d. tag were also appended to the filtered link list, but only the portion of the link that didn’t match the expression of a trailing i.d. tag.

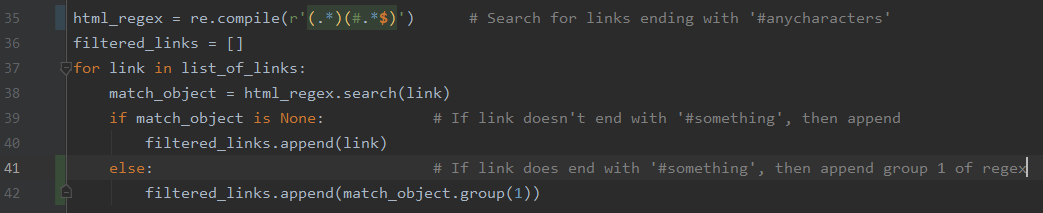


Figure 4. Regex link filtering. This code identifies links with a trailing i.d. tag and omits that portion of the link.

After all links were cleaned up, the final step was to remove duplicates and organize them. The sort() function easily removed any duplicate values. By passing the list in as an argument and assigning the output to the same list, line 44 of the code removes duplicate links. The following code in line 45 simply organized the links in the list alphabetically.



Figure 5. Link sorting. This code removes duplicate values and sorts them.

References

Sweigart, A. (2015). Automate the boring stuff with Python: Practical programming for total beginners. San Francisco, CA: No Starch Press.