This post will compare Python’s BeautifulSoup package to R’s rvest package for web scraping. We’ll also talk about additional functionality in **rvest** (that doesn’t exist in **BeautifulSoup**) in comparison to a couple of other Python packages (including **pandas** and **RoboBrowser**).

**Getting started**

BeautifulSoup and rvest both involve creating an object that we can use to parse the HTML from a webpage. However, one immediate difference is that BeautifulSoup is just a web parser, so it doesn’t connect to webpages. **rvest**, on the other hand, can connect to a webpage and scrape / parse its HTML in a single package.

In BeautifulSoup, our initial setup looks like this:

# load packages

from bs4 import BeautifulSoup

import requests

# connect to webpage

resp = requests.get(""https://www.azlyrics.com/b/beatles.html"")

# get BeautifulSoup object

soup = BeautifulSoup(resp.content)

In comparison, here’s what using **rvest** is like:

# load rvest package

library(rvest)

# get HTML object

html\_data <- read\_html("https://www.azlyrics.com/b/beatles.html")

**Searching for specific HTML tags**

Next, let’s take our parser objects and find all the links on the page.

If you’re familiar with **BeautifulSoup**, that would look like:

links = soup.find\_all("a")

In rvest, however, we use syntax similar to **dplyr** and other tidyverse packages by using %>%.

links <- html\_data %>% html\_nodes("a")

urls <- links %>% html\_attr("href")

In **BeautifulSoup**, we use the *find\_all* method to extract a list of all of a specific tag’s objects from a webpage. Thus, in the links example, we specify we want to get all of the anchor tags (or “a” tags), which create HTML links on the page. If we wanted to scrape other types of tags, such as *div* tags or *p* tags, we just need to switch out “a” with “div”, “p”, or whatever tag we want.

# get all div tags

soup.find\_all("div")

# get all h1 tags

soup.find\_all("h1")

With **rvest**, we can get specific tags from HTML using *html\_nodes*. Thus, if we wanted to scrape different tags, such as the div tags or h1 tags, we could do this:

# scrape all div tags

html\_data %>% html\_nodes("div")

# scrape header h1 tags

html\_data %>% html\_nodes("h1")

**Getting attributes and text from tags**

In **BeautifulSoup**, we get attributes from HTML tags using the *get* method. We can use a list comprehension to get the *href* attribute of each link (the *href* attribute of a link is its destination URL).

urls = [link.get("href") for link in links]

To get other attributes, we just need to change our input to the *get* method.

# get the target attribute from each link

[link.get("target") for link in links]

# get the ID attribute of each div tag

[div.get("id") for div in soup.find\_all("div")]

Using **rvest**, the *html\_attr* function can be used to get attributes from tags. So to get the URL of each link object we scrape, we need to specify that we want to get the *href* attribute from each link, similarly to **BeautifulSoup**:

urls <- links %>% html\_attr("href")

Likewise, if we want to scrape the IDs from the div tags, we can do this:

html\_data %>% html\_nodes("div") %>% html\_attr("id")

Notice how, like other tidyverse packages, we can chain together multiple operations.

If we want to scrape the text from each of the links, we can use *html\_text*:

links %>% html\_text()

**BeautifulSoup’s** way of accomplishing this is by using the *text* method of a tag object:

[link.text for link in links]

**Scraping HTML tables**

Let’s look at another example for scraping HTML tables.

We can scrape HTML tables using **rvest’s** *html\_table* method. This method will extract all tables found on the input webpage. The *fill = TRUE* parameter is specifying that we want to fill any rows that have less than the maximum number of columns in a table with NAs. The tables will be stored as a list of data frames.

city\_data <- read\_html("http://www.city-data.com/city/Florida.html")

city\_data %>% html\_table(fill = TRUE)

Scraping tables with **BeautifulSoup** into a data frame object is a bit different. One way we can scrape tables with Python is to loop through the tr (row) or td (data cell in table) tags. But the closest analogy of **rvest’s** functionality here is to use **pandas**:

import pandas as pd

pd.read\_html("http://www.city-data.com/city/Florida.html")

Like using *html\_table*, this will return a list of data frames corresponding to the tables found on the webpage.

**Browser simulation with rvest**

An additional feature of **rvest** is that it can perform browser simulation. **BeautifulSoup** cannot do this; however, Python offers several alternatives including **requests\_html** and **RoboBrowser**.

With **rvest**, we can start a browser session using the *html\_session* function:

site = "https://www.azlyrics.com/b/beatles.html"

session <- html\_session(site)

With our session object, we can navigate to different links on the page, just like a real web browser. There’s a couple ways of doing this. One is to input the index of the link we want to go to. For example, to navigate to the third link of the page, we would write the below code:

session %>% follow\_link(3)

Here’s a couple more examples:

# navigate to the 5th link on the page

session %>% follow\_link(5)

# navigate to the 10th link on the page

session %>% follow\_link(10)

You can also simulate clicking on links based off text. For example, the below code will navigate to the first link containing the text “Sun”. The input is case sensitive.

session %>% follow\_link("Sun")

You can use the session object to navigate directly to other webpages using the *jump\_to* function.

session %>% jump\_to("https://www.azlyrics.com/a.html")

If we use the **RoboBrowser** package in Python to somewhat replicate the R code above, we could write this:

from robobrowser import RoboBrowser

# create a RoboBrowser object

browser = RoboBrowser(history = True)

# navigate to webpage

browser.open("https://www.azlyrics.com/b/beatles.html")

# get links

links = browser.get\_links()

# follow link

browser.follow\_link(links[5])

# follow different link

browser.follow\_link(links[10])

The *open* method used above is analogous to **rvest’s** *jump\_to* function. The *follow\_link* method in **RoboBrowser** serves a similar purpose to the function of the same name in **rvest**, but behaves a little differently. This method takes a *link object* as input, rather than the index of a link, or text within a link that you’re searching for. Thus, we can use the *links* object above to specify a link that we want to “follow” or click. If we want to click on a link based off text like we did in the **rvest** example above, we could write the below code.

# filter the list of links to only links containing "Sun" in their text

sun\_links = filter(lambda link: "Sun" in link.text, links)

# Click on the first link containing the word "Sun"

browser.follow\_link(next(sun\_links))