SESUG Paper 236-2018

Web Scraping in SAS: A Macro-Based Approach

Jonathan W. Duggins, North Carolina State University;
Jim Blum, University of North Carolina Wilmington

ABSTRACT

Web scraping has become a staple of data collection due to its ease of implementation and its ability to provide access to wide variety of data, much of it freely available. This paper presents a case study that retrieves data from a website and stores it in a SAS data set. PROC HTTP is discussed and the presented technique for scraping a single page is then automated using the SAS Macro Facility. The result is a macro that can be customized to access data from a series of pages and store the results in a single data set for further use. The macro is designed with a built-in delay to help prevent server overload when requesting large amounts of information from a single site. The macro is designed for both academic and industry use.

INTRODUCTION

Web scraping can be defined as an automated process for taking data presented on a web server and storing in a format suited to further analysis. This process varies in complexity based on the formatting used to display the data and, depending on the amount of HTML encoding included on the page and the structure used to display the data, extracting the data can be tedious. However, taking advantage of the fact that HTML files are text files, the SAS DATA step provides significant functionality for parsing the file. Once the file is parsed, you must often reshape the results and SAS offers multiple tools to facilitate this process, including data step arrays, functions, and procedures such as the TRANSPOSE Procedure and the SQL Procedure. The following sections outline one way to leverage SAS for web scraping, with possible alternatives and potential pitfalls discussed as appropriate. Familiarity with non-SAS concepts such as HTML encoding and HTML tags, as well as SAS tools including the SAS DATA step, PROC SQL, and the SAS Macro Facility are beneficial.

DETERMING THE DATA SOURCE

Before beginning any web scraping process, including those outlined below, ensure the target site allows scraping. Navigate to the intended website and access the robots.txt file, which is used to provide web crawlers, both human and robot alike, with important access information. For a simple primer on how to understand the contents of a robots.txt file, refer to https://www.promptcloud.com/blog/how-to-read-and-respect-robots-file, which addresses some of the basic concepts. In addition to telling whether scraping the site is allowed, it also indicates any restrictions such as time of day, frequency, or forced delays.

Once you identify a suitable website, it is important to realize that the data may be stored in various formats for web-based dissemination. One of the simplest is for a website to link directly to a file. As an example, consider the file ERA.txt which is located on the author's personal page and contains comma-delimited data for every pitcher in Major League Baseball from 2017. Of course, one approach to converting the data is to download the file locally and use a DATA step to access the information. However, an alternative is to direct SAS to the web location itself using the FILENAME statement with the URL option, followed by a DATA step to parse the file, as shown below.

```
FILENAME era URL "https://www4.stat.ncsu.edu/~duggins/SESUG2018/ERA.txt";
DATA era;
  INFILE era DSD FIRSTOBS = 2;
  INPUT MaskedLeague : $1. ERA;
RUN;
```

As mentioned previously, HTML files are simply text files as well, so you can use this method for scraping data from more traditional web sources that contain any HTML encoding. However, in those cases the HTTP Procedure is a more robust alternative, and has been updated in SAS 9.4 to improve its efficiency. The remaining examples demonstrate the use of PROC HTTP and focus on pages with HTML encoding.

PARSING A SINGLE PAGE

In many cases, the data of interest is located on a single page. In such cases the primary challenge is not accessing the file, but instead lies in parsing the HTML tags to obtain the data of interest. The case study presented here focuses on a popular web comic, xkcd, located at www.xkcd.com. In particular, the archive (www.xkcd.com/archive) lists every comic published by its author, Randall Munroe. The archive page does not appear to give any other information about the comics; however, viewing the page source shows that in addition to the name of each comic strip, the comic number and publication date are stored in the metadata. Saving this information to a SAS data set is thus a two-step process: access the page, then parse out the necessary information.

ACCESSING THE ARCHIVE'S SOURCE CODE

PROC HTTP is useful for connecting to the webpage and reading the HTML source code into a SAS data set, as is demonstrated in the example below:

```
FILENAME source TEMP;
PROC HTTP
    URL = "https://xkcd.com/archive/"
    OUT = source
    METHOD = "GET";
RUN;
```

The FILENAME statement is used to create a temporary (logical-only) location for SAS to store the website's contents. When invoking the HTTP Procedure, the URL= option indicates the target website and the OUT= option indicates the file where the information retrieved from the website is to be stored. The METHOD= option indicates what type of request method PROC HTTP should send to the website identified in the URL= option. While there are several HTTP request methods available in PROC HTTP, GET is the most basic, simply retrieving the source code from the web page designated in the URL= option. For other available methods and their effects, see the SAS Help Documentation.

PARSING THE ARCHIVE'S SOURCE CODE

Once the source code is stored in the temporary file, a DATA step is used to parse the results. In the current case study, the source file is over 8000 lines long, but only approximately 25% of them contain information about comics. The HTML code for one comic is shown below.

```
<a href="/2029/" title="2018-8-6">Disaster Movie</a><br/>
```

A single anchor tag provides multiple pieces of information, including: a link to the comic via its number in the HREF= attribute, the publication date is in the TITLE= attribute, and the comic title is found between the opening and closing anchor tags. Fortunately, the same format is used for each comic. However, since new comics are added three times per week, the page is dynamic and thus line numbers are not a reliable way to access the desired information. Instead, logical conditions must be set to access the exact lines of interest. Due to the flexibility of the DATA step there are a variety of ways to implement the parsing, with one possible approach shown below:

```
DATA work.xkcdArchive(DROP = line);
  FORMAT num 4. date DATE9.;
  LENGTH title $ 50;
  INFILE source LENGTH = recLen LRECL = 32767;
  INPUT line $VARYING32767. recLen;
  IF FIND(line, '<a href=') GT 0 AND FIND(line, 'title=') GT 0 THEN DO;
      num = INPUT(scan(strip(line),2,'/'),4.);
      date = INPUT(scan(strip(line),4,'"'),YYMMDD10.);
      title = SCAN(strip(line),2,'<>');
      OUTPUT;
    END;
RUN;
```

Step 1: Accessing the File

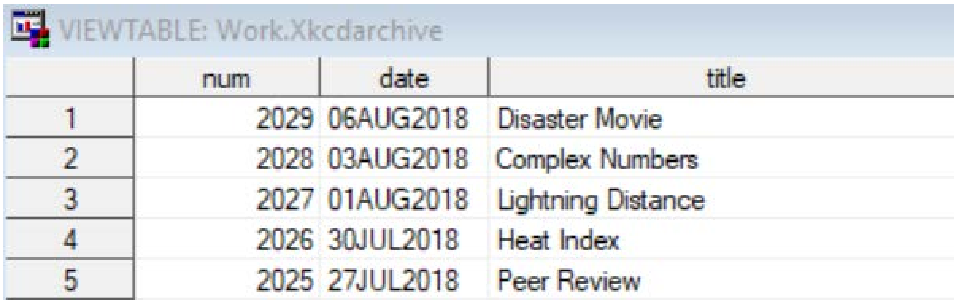
The INFILE statement refers to the earlier temporary file, source, and sets the logical record length to 32,767. (SAS 9.4 ships with this as the default value for varying length files, but the value was 256 in earlier versions. This option is included to help with backwards compatibility of the program.) Since the lines in an HTML file may vary wildly in length, the LENGTH= option allows you to store the length of the current line in the temporary variable recLen. Pairing this temporary variable with the $VARYING informat in the INPUT statement allows the length of each line being read in to be tracked in the DATA step. This approach has two advantages:

1. using varying-widths is more efficient than a fixed-width approach for varying length records, and
2. if the line exceeds 32,767 characters a single trailing at (@) can be used in conjunction with the recLen value to step through the record until all text is read.

Step 2: Selecting the Appropriate Lines

After the INFILE and INPUT statements access the source code for the archive page, an IF-THEN statement determines which lines of HTML code contain the desired information. As mentioned above, each line of interest used an <a> tag with the HREF= and TITLE= attributes. The IF-THEN statement conditions on this by using the FIND function to select only these rows for further parsing. Three variables are then created: comic number (num), date of publication (date), and comic title (title). Each assignment statement makes use of the SCAN function to identify which piece of the source code should be stored in a given variable. SCAN and SUBSTR are powerful functions for data extraction since HTML coding commonly uses delimiters such as angle brackets and forward slashes.

Finally, the OUTPUT statement is used to ensure that only these parsed records are output to the resulting data set. The initial LENGTH and FORMAT statements, as well as the DROP= data set option, ensure the resulting data set represents the source data appropriately, while the INPUT functions and informats used in the definitions of Num and Date ensure that the resulting data set can be correctly sorted if needed. Figure 1 shows the partial results.


VIEWTABLE: Work.Xkcdarchive

	num	date	title
1	2029	06AUG2018	Disaster Movie
2	2028	03AUG2018	Complex Numbers
3	2027	01AUG2018	Lightning Distance
4	2026	30JUL2018	Heat Index
5	2025	27JUL2018	Peer Review

Figure 1 Partial results from DATA step parsing the archive web page.

Potential Pitfalls

As mentioned above, this is only one approach to scraping this particular web page. Other functions such as INDEX instead of FIND may be preferable or different logical conditions for selecting the appropriate rows could be used. In particular, the logical conditions often contribute to the fragility of a web scraping program; if the targeted website does not use consistent HTML coding then the scraping program must account for that. In this case study, the assumption is that the website will never use HREF= instead of href= or Title= instead of title= in its <a> tags.

Functions such as UPCASE are common in situations such as this for matching character strings just as COMPRESS may be beneficial in case additional spaces were included between the <a and href= portions of the HTML source code. Before writing a DATA step to parse the HTML source code, the coding structure used in the website must be known. For larger sites, implementing data integrity or other validation checks is often necessary to help ensure the data set produced is accurate.

PARSING MULTIPLE PAGES

Of course, often data is broken across multiple web pages instead of being stored at a single URL. This is common for large data sets such as those scraped for analyzing CDC records, improving a real estate pricing algorithm, or running a fantasy baseball league. In order to scrape data in this scenario, it must first be determined how the website indicates the various pages. Many pages use parameters such as *?page=1* appended to the end of the URL. For the case study in use here, each comic is accessed by using the comic's number, *e.g.* www.xkcd.com/2029 is the direct link to comic number 2029. This pagination provides a natural way for incorporating macro variables into the process of scraping data that spans a range of pages.

The process for scraping multiple pages is similar to the process for a single page: access the file, then select the appropriate lines. Construction of a macro allows for application of the process to all pages to be scraped. However, with data broken across multiple pages, the logical conditions necessary to select the necessary lines can become much more complicated. As before, there are multiple ways to parse this HTML source code, and the following sections step through one approach that utilizes a SAS macro to access each comic's page and PROC SQL for parsing and combining the appropriate lines from the HTML source code.

CONSTRUCTING THE MACRO

The macro detailed below has three main components: obtaining the source code for a specific page number, storing the source code as a SAS data set, and parsing the source code to obtain the data of interest.

Step 1: Macro Variables

To begin the process of creating a macro that handles pagination, you can use PROC SQL to create a list of pages you plan to access via the macro. The xkcdArchive data set created above, by design, contains the comic numbers which the website uses in the URL to identify pages. The SQL Procedure shown below places the comic numbers into a single global macro variable, indexList, and puts the number of comics into the global macro variable indexCount.

```
PROC SQL NOPRINT;
  SELECT DISTINCT num, count(DISTINCT num)
  INTO :indexList separated BY ' ', :indexCount
  FROM work.xkcdArchive
  ;
QUIT;
```

The indexCount variable is necessary because the number of comics is not equal to the largest value in the indexList variable. (For example, there is no comic 404 – presumably due to the conflict with the 404 error page that would be generated.) Next, the macro statement shown below identifies four keyword parameters to set macro variables useful for scraping multiple pages from a single site.

```
%MACRO scrape(start=1, stop=&indexCount, out=, append=, sleep=5);
```

The Start and Stop variables are useful for testing purposes as well as for cases where future scraping processes do not need to revisit pages scraped previously. The Out variable is used to identify the data set in which the final records should appear. Append is used to determine whether the pages scraped by the macro should be appended to the specified data set or if the data set should be overwritten. The final variable, sleep, controls the amount of delay between subsequent requests to the website (Recall the website's robots.txt file may indicate the required delay amount.) The implementation of the delay is discussed below.

The %IF-%THEN/%ELSE statements allow for possible casing and input variations on 'Yes' and 'No' for identifying whether appending or replacing should occur. In other cases, an error message indicates the expected user-provided values. (Note that the Work library is hardcoded throughout this case study. In practice, you may find it useful to include an additional keyword parameter indicating the library.)

Additionally, based on the value of Append conditional logic determines whether the Out data set should be deleted.

```
%IF %SYSFUNC(COMPRESS(%UPCASE(%SUBSTR(&Append,1,1)))) = N %THEN %DO;
      PROC DATASETS LIBRARY = Work NOLIST;
        DELETE All;
        RUN;
      QUIT;
    %END;
    %ELSE %PUT QC_ERROR: Append status must be Yes or No. &=append;
```

Within the macro a %DO loop is used to carry out the iterations, as shown in the following (partial) code.

```
%DO i = &Start %TO &Stop;
   %LET index = %SCAN(&indexList, &i);
```

The %LET statement sets the value of the macro variable Index, which is used to identify the current page to be read in the following PROC HTTP step.

Step 2: Accessing and Storing a Page’s Source Code

Next, use code similar to that from the previous section to access the page’s source code.

```
FILENAME source TEMP;
PROC HTTP
    URL = "https://xkcd.com/&index/"
    OUT = source
    METHOD = "GET";
RUN;
```

The PROC HTTP step above functions identically to the one used to scrape the archive page earlier; however, the URL is updated to include the Index macro variable to make the page reference dynamic.

```
DATA work.xkcdRawTemp;
  INFILE source LENGTH = len LRECL = 32767;
  INPUT line $VARYING32767. len;
  LineNum = _n_;
RUN;
```

Unlike the DATA step used in the previous section to parse the archive page, this DATA step simply stores the source code, no parsing or cleaning is carried out. Also, note the addition of the LineNum variable. As discussed previously, some pages contain information across multiple lines of HTML source code that need to be combined into a single record, and this variable will be used to address that issue.

Step 3: Parsing the Source Code

As with any data manipulation process, successful web scraping requires a solid understanding of the raw data, or page source in this case. Often, the data of interest lies between the opening and closing body tags and may even be contained in a single <div> or <table> section of the source code. To provide further insight into the source code involved in this case study, a snippet from a typical page’s source code is shown below. As the permanent link indicates, this source code is from comic number 500.

```
<div id="ctitle">Election</div>
<img src="//imgs.xkcd.com/comics/election.png" title="Someday I&#39;ll be
rich enough to hire Nate Silver to help make all my life decisions.
&#39;Should I sleep with her?&#39;  &#39;Well, I&#39;m showing a 35%
chance it will end badly.&#39;" alt="Election" />
</div>
Permanent link to this comic: https://xkcd.com/500/<br />
Image URL (for hotlinking/embedding):
https://imgs.xkcd.com/comics/election.png
```

```
<div id="transcript" style="display: none">[[Character sits at his
computer desk, staring at his computer.]]
It&#39;s over.
After twenty months it&#39;s finally over.
I don&#39;t have to be an election junkie anymore.
[[Closeup of character&#39;s face and screen.]]
I don&#39;t have to care about opinion polls, exit polls, margins of
error, attack ads, game-changers, tracking polls, swing states, swing
votes, the Bradley effect, or &lt;name&gt; the &lt;occupation&gt;.
I&#39;m free.
[[Character staring at his computer screen, full shot.]]
[[Character types on his computer.]] &lt;&lt;Tap Tap&gt;&gt;
[[On screen]]Google  &quot;2012 polling statistics&quot;
{{Title text: &quot;Someday I&#39;ll be rich enough to hire Nate Silver to
help make all my life decisions.  &#39;Should I sleep with her?&#39;
&#39;Well, I&#39;m showing a 35% chance it will end badly.&#39;
&quot;}}</div>
```

For this case study, several elements are of interest.

1. The comic's title – included in the first <div> tag in the snippet
2. The hover text –included in the <img> tag
3. The alternative hover text – also included in the <img> tag
4. The link for the comic – included on its own line
5. The link for embedding – included on its own line
6. The transcript – spans from <div> that includes the word transcript until the end of the snippet.

The transcript was of particular interest since it provided a textual interpretation of the comic that could be used for screen readers or in other scenarios to facilitate accessibility of the pages. In particular, note that several items in the transcript, such as apostrophes, quotes, and angle brackets, are in character entity form. For example, &gt; appears as > when the text is displayed in a web browser.

Furthermore, since the transcripts may use a different number of lines for each comic, a static approach based on line numbers is not sufficient. This snippet of code indicates the need to read multiple lines from the source code but combine them into a single record. There are certainly multiple ways to leverage SAS in this case, but an SQL-based approach is detailed here using the queries and in-line views below. The five individual in-line views are covered first.

View A

```
(SELECT HTMLDECODE(SCAN(line,2,'<>')) AS title
  FROM work.xkcdRawTemp
  WHERE line CONTAINS 'div id="ctitle"') AS a,
```

In the first in-line view, the title is obtained from appropriate <div> tag. The HTMLDECODE function is used to ensure any character entity references are properly rendered. The SCAN function parses the <div> tag to obtain only the title, with one row returned by this in-line view for each page.

View B

```
(SELECT DISTINCT HTMLDECODE(SCAN(line,4,'"')) AS Hover,
        HTMLDECODE(SCAN(line,6,'"')) AS AltHover
   FROM work.xkcdRawTemp
   WHERE line contains '<img src='
         AND
         line contains 'title') AS b,
```

The second in-line view obtains the hover text and alternate hover text from the <img> tag. The DISTINCT keyword is used to ensure only one set of hover text is returned per page, thus guaranteeing the in-line view only returns a single row. The compound logical condition is used to filter out other images, besides the comic itself which is titled, from the selection. This only works because other images, such as the xkcd.com log, are presented without titles.

Views C and D

```
(SELECT HTMLDECODE(STRIP(SCAN(line,6,' <'))) AS PermLink
  FROM work.xkcdRawTemp
  WHERE line CONTAINS 'Permanent link to this comic:') AS c,

(select HTMLDECODE(STRIP(SCAN (line,5,' '))) AS EmbedLink
   FROM work.xkcdRawTemp
   WHERE line CONTAINS 'Image URL (for hotlinking/embedding)') AS d
```

The next two in-line-views simply request the rows containing the URL for the link to the comic's webpage (View C) and the link to the comic's image file (View D). Both return a single row.

View E

```
(SELECT LineNum, HTMLDECODE(line) AS test /*View E*/
   FROM work.xkcdrawTemp
   WHERE (LineNum GE
          (SELECT MIN(LineNum) /*Inner Query 1*/
            FROM work.xkcdRawTemp
            WHERE line CONTAINS '<div id="transcript"')
          AND
          LineNum LE
           (SELECT MIN(LineNum) /*Inner Query 2*/
             FROM work.xkcdRawTemp
             WHERE line CONTAINS '</div>'
                   AND
                   LineNum GE
                    (SELECT MIN(LineNum) /*Inner Query 3*/
                      FROM work.xkcdRawTemp
                      WHERE line CONTAINS '<div id="transcript"')))) AS e
```

In order to capture the entire transcript, the final in-line view includes subqueries. Unlike Views A through D where conditional logic can identify the exact line in the HTML source code to store, View E must select lines of source code based on relative position. Specifically, the transcript begins with the <div> tag for which the ID attribute includes the word 'transcript' and it continues until the closing </div> tag. Since those line numbers are not fixed, conditional logic is required to identify the appropriate values of LineNum. The subqueries above do so by:

1. Inner Query 1: Find smallest line number of all lines containing a transcript-related <div> tag.
2. Inner Query 2: Find smallest line number for all lines containing a </div> tag *as long as that value is larger than the line number given by Inner Query 1 (which is repeated as Inner Query 3)*
3. View E: Select all lines of source code between values #1 and #2.

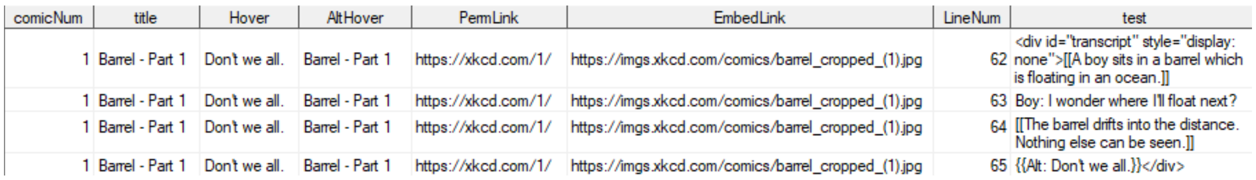
View E does not return a single row. Instead it returns a dynamic number of rows based on the length of the transcript provided on the webpage – some of which span hundreds of lines of HTML source code.

Combining the In-Line Views

The five in-line views detailed above select various types of information with Views A through D selecting single rows using conditional logic alone while View E selects multiple rows from an arbitrary position on the webpage since the position changes as the page number changes. This produces a situation where a full Cartestian product of the five tables, with one row per table for the first four tables and multiple rows for the fifth table, is necessary to combine all the information. SQL can again be used for this manipulation via the following pseudocode.

```
PROC SQL NOPRINT;
 CREATE TABLE work.xkcdParsedTemp AS
   SELECT &index AS comicNum, a.*, b.*, c.*, d.*, e.*
     FROM <INSERT VIEWS A – E HERE>
 ;
QUIT;
```

The result of the HTTP, DATA, and SQL steps for comic number 1 is shown in Figure 2 below.



comicNum	title	Hover	AltHover	PermLink	EmbedLink	LineNum	test
1	Barrel - Part 1	Don't we all.	Barrel - Part 1	https://xkcd.com/1/	https://imgs.xkcd.com/comics/barrel_cropped_(1).jpg	62	<div id="transcript" style="display: none">[[A boy sits in a barrel which is floating in an ocean.]]
1	Barrel - Part 1	Don't we all.	Barrel - Part 1	https://xkcd.com/1/	https://imgs.xkcd.com/comics/barrel_cropped_(1).jpg	63	Boy: I wonder where I'll float next?
1	Barrel - Part 1	Don't we all.	Barrel - Part 1	https://xkcd.com/1/	https://imgs.xkcd.com/comics/barrel_cropped_(1).jpg	64	[[The barrel drifts into the distance. Nothing else can be seen.]]
1	Barrel - Part 1	Don't we all.	Barrel - Part 1	https://xkcd.com/1/	https://imgs.xkcd.com/comics/barrel_cropped_(1).jpg	65	{{Alt: Don't we all.}}</div>

Figure 2 Results of scraping first comic from the archive.

As mentioned above, the LineNum variable tracks separate lines of the transcript for a single comic. Notice the Test variable holds the transcript but has additional text such as HTML tags, brackets, braces, and indicators such as "Alt:" which are not part of the actual transcript. Removing the extraneous text is the next step in the scraping process for this case study.

Step 4: Data Cleaning

As Figure 2 shows, the parsed data requires further data manipulations to obtain the transcript. You can choose to do this within each iteration of the %DO loop or once the data from all pages has been combined. The following DATA step makes use of FIND, COMPRESS, STRIP, and SUBSTR to handle initial unwanted text such as the <div> tag and opening brackets and braces. Conditional logic is used to avoid unnecessarily applying the SUBSTR functions. The final statement makes use of the TRANWRD function to remove certain elements from the text.

```
DATA work.xkcdClean;
  SET work.xkcdParsedTemp;
  IF FIND(test,'[[') GT 0 THEN DO;
      startsq = FIND(test,'[[') + 2;
      test = STRIP(COMPRESS(SUBSTR(test,startsq),'[]'));
    END;
  IF FIND(test,'{{') GT 0 OR FIND(upcase(test), 'ALT TEXT') GT 0 THEN DO;
      startcu = FIND(test,':') + 1;
      test = STRIP(COMPRESS(SUBSTR(test,startcu),'{}'));
    END;
  IF FIND(test,'<div') GT 0 THEN test = STRIP(SCAN(TEST,2,'>'));
test = STRIP(TRANWRD(TRANWRD(TRANWRD(test,']]',''),'}}',''),'</div>',''));
RUN;
```

Some programmers may wish to use regular expressions to simply select the text between sets of brackets, braces, or angle brackets. However, since the transcript text is manually entered, cases exist where double braces were used to open a section but only a single brace was used in closing. Brackets, braces, and angle brackets are also occasionally used within the text of the transcript. Hence, a combination of positional approaches are used based on SAS functions.

Step 5: Appending and Sleeping

A simple APPEND or DATASETS Procedure provides an efficient way to combine data scraped from multiple pages.

```
PROC APPEND BASE = work.all DATA = work.xkcdParsedTemp;
RUN;
```

Recall the keyword parameter, Sleep, in the definition of the Scrape macro. Before scraping subsequent pages, a delay needs to be incorporated using this macro variable. SAS provides a SLEEP routine, available via the DATA step, that you can use to accomplish this task.

```
DATA _NULL_;
  CALL SLEEP(&Sleep,1);
RUN;
```

The SLEEP routine stops SAS from executing the program for a specified amount of time. It accepts two arguments: the first is the duration and the second is the units. When specifying the units, a value of 1 indicates seconds, while a value of 0.01 indicates hundredths of seconds. In the code presented above, the value of the macro variable Sleep determines the number of seconds to delay the execution of the program.

Step 6: Concluding the Macro

For the case study presented here, the HTTP step to access the HTML source code, the DATA step to store the source code, the SQL step to parse the source code, the DATA step to clean the parsed source code, the APPEND step to combine the results from multiple pages within a website, and the null DATA step to implement the delay form the entirety of the %DO loop. Once the null DATA step is executed, the %END statement concludes the %DO loop which scrapes across pages.

The final step is to use the TRANSPOSE Procedure to ensure that each comic is associated with a single observation in the SAS data set. Use of the ID statement guarantees the transcript's lines are kept in order during the transposition.

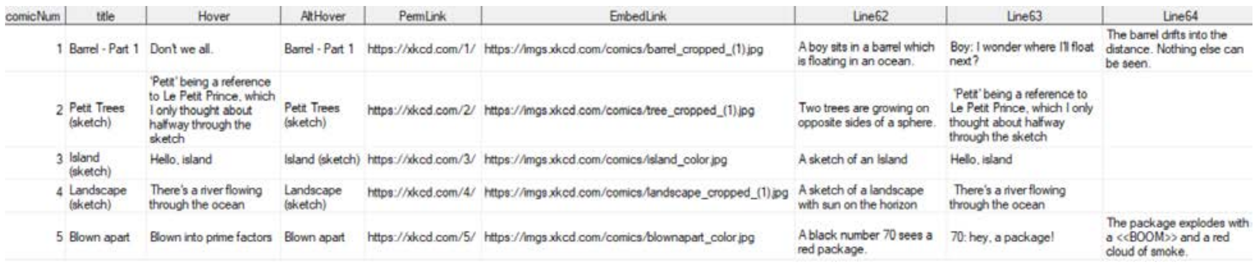
```
PROC TRANSPOSE DATA = work.All
               OUT = work.&out (DROP = _name_)
               PREFIX = Line;
   BY comicnum title hover altHover permLink EmbedLink;
   ID linenum;
   VAR test;
RUN;
%MEND; /*End of Scrape macro*/
```

The statement below calls the full macro. Recall the default macro variables Start and Stop were given default values of 1 and &indexCount, respectively. As such, this call scrapes all comics in the xkcd archive, creates a new data set named xkcdParsed (or overwrites it) and uses a delay of 0.1 seconds between each iteration of the %DO loop.

```
%scrape(out = xkcdParsed, append = n, sleep = 0.1)
```

Partial Results

Figure 3 shows the partial results from execution of the macro. Comparing to Figure 2, you can see the extraneous text has been removed from the transcript content and each comic appears as a single observation.



comicNum	title	Hover	AltHover	PermLink	EmbedLink	Line62	Line63	Line64
1	Barrel - Part 1	Don't we all.	Barrel - Part 1	https://xkcd.com/1/	https://imgs.xkcd.com/comics/barrel_cropped_(1).jpg	A boy sits in a barrel which is floating in an ocean.	Boy: I wonder where I'll float next?	The barrel drifts into the distance. Nothing else can be seen.
2	Petit Trees (sketch)	'Petit' being a reference to Le Petit Prince, which I only thought about halfway through the sketch	Petit Trees (sketch)	https://xkcd.com/2/	https://imgs.xkcd.com/comics/tree_cropped_(1).jpg	Two trees are growing on opposite sides of a sphere.	'Petit' being a reference to Le Petit Prince, which I only thought about halfway through the sketch	
3	Island (sketch)	Hello, island	Island (sketch)	https://xkcd.com/3/	https://imgs.xkcd.com/comics/island_color.jpg	A sketch of an Island	Hello, island	
4	Landscape (sketch)	There's a river flowing through the ocean	Landscape (sketch)	https://xkcd.com/4/	https://imgs.xkcd.com/comics/landscape_cropped_(1).jpg	A sketch of a landscape with sun on the horizon	There's a river flowing through the ocean	
5	Blown apart	Blown into prime factors	Blown apart	https://xkcd.com/5/	https://imgs.xkcd.com/comics/blownapart_color.jpg	A black number 70 sees a red package.	70: hey, a package!	The package explodes with a <<BOOM>> and a red cloud of smoke.

Figure 3 Partial Results for first 5 scraped pages

POTENTIAL PITFALLS

Any web scraping process, regardless of implementation, is likely to be quite fragile. Some approaches, such as considering casing and spacing, avoiding assumptions about line numbers on which the content exists, and selecting only distinct rows are used the macro above in an attempt to produce robust code for this case study. However, additional issues need to be considered. For example, HTML tags were assumed to be lowercase in all scenarios, which is not always the case for other websites. Additional applications of UPCASE or other appropriate character functions can produce even more robust code. However, any change to the source code by the website can result in additional programming time necessary to maintain a web scraping program.

Efficiency problems may arise as well due to the continuous increase in the number of pages to scrape from this site. As a result, PROC TRANSPOSE must operate on an increasingly large data set. Furthermore, as Figure 3 demonstrates, the number of lines of transcript is not constant. Alternatives to PROC APPEND that don't depend as heavily on a stable set of variables could alleviate the issue by allowing the PROC TRANSPOSE to appear inside, rather than outside, the loop. Currently, with nearly 2100 pages to scrape the macro takes approximately 12 minutes to execute. This includes using options such as NONOTES and NOSOURCE to reduce runtime.

CONCLUSION

Web scraping continues to gain popularity as a method for data collection. As demonstrated here, SAS is a powerful tool that provides a variety of options for obtaining data in this manner. While the case study here focused on a single approach using SQL to parse the HTML source code and join appropriate lines, followed by a DATA step for cleaning purposes, other approaches are also available. See Hemedinger, 2017 for an alternative approach using the SET statement's POINT= option in the DATA step rather than SQL for selecting information. In both cases, conditional logic and access to regular expressions allow SAS to emulate the CSS selectors available in other web scraping programs such as Python.

By leveraging the SAS macro facility, it is possible to scrape data stored both on single and multiple pages. Both SQL and the DATA step, either individually or in conjunction, allow for the conditional selection of records. This is of particular importance for websites that span multiple pages but do not use consistent HTML code to display the data across those pages. The variety of ways in which data is stored on the web often require development of individualized programs for each site to be scraped. However, as tools like PROC HTTP continue to receive updates and as additional resources become available, the flexibility of SAS continues to be a substantial resource in web scraping.

REFERENCES

Hemedinger, Chris. "How to scrape data from a web page using SAS" *The SAS Dummy*. December 4, 2017. Available at https://blogs.sas.com/content/sasdummy/2017/12/04/scrape-web-page-data/.

Koshy, Jacob. "How to Read and Respect Robots.txt" *Prompt Cloud.* March 3, 2017. Available at https://www.promptcloud.com/blog/how-to-read-and-respect-robots-file.

ACKNOWLEDGMENTS

I would like to thank my friends and colleagues, Travis Gray and Jeremy Strickland, for the discussions regarding web scraping applications outside of academia.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Jonathan W. Duggins
NC State University
jwduggin@ncsu.edu
https://www4.stat.ncsu.edu/~duggins/