Notes on Web Scraping

For purposes of BIOS 669, web scraping means programmatically pulling data from the HTML code of a web page. That is, the data has not explicitly been provided as a file that can be downloaded or surfaced via an API. Rather, the data appears in a table on a web page or as part of the text of one or more web pages, and therefore is part of the HTML behind that page or pages, and you want to grab that data into a file for analytical purposes.

These notes focus on examples of four methods for web scraping without going into details of exactly how those methods work. So, as you try one or more of these methods for yourself, you might need to do further research on applying the methods in different situations. But I hope these notes can at least get you started and increase your willingness to jump into the web scraping game.

Whatever web page you are looking at, there is some way that you can see the HTML source code behind the page. In Chrome, the way is to right-click anywhere on the page and select *View page source*. If you do this, notice that HTML is a mark-up language, where every piece of text has associated tags (those things in angle brackets such as <div>, <ul>, <a>, etc.). Any web scraping method involves parsing these lines of HTML at some level, with the parsing based on those tags. In the methods shown in these notes, sometimes the parsing is done in the background and we focus on how to pull a table of data off of the page. In other cases, we use more granular parsing methods to pick off particular numbers or strings rather than entire tables of data.

For reading entire tables, the two methods to be demonstrated use R and JMP. For picking off particular data items, the two methods use SAS and Python, specifically a Python library called Beautiful Soup.

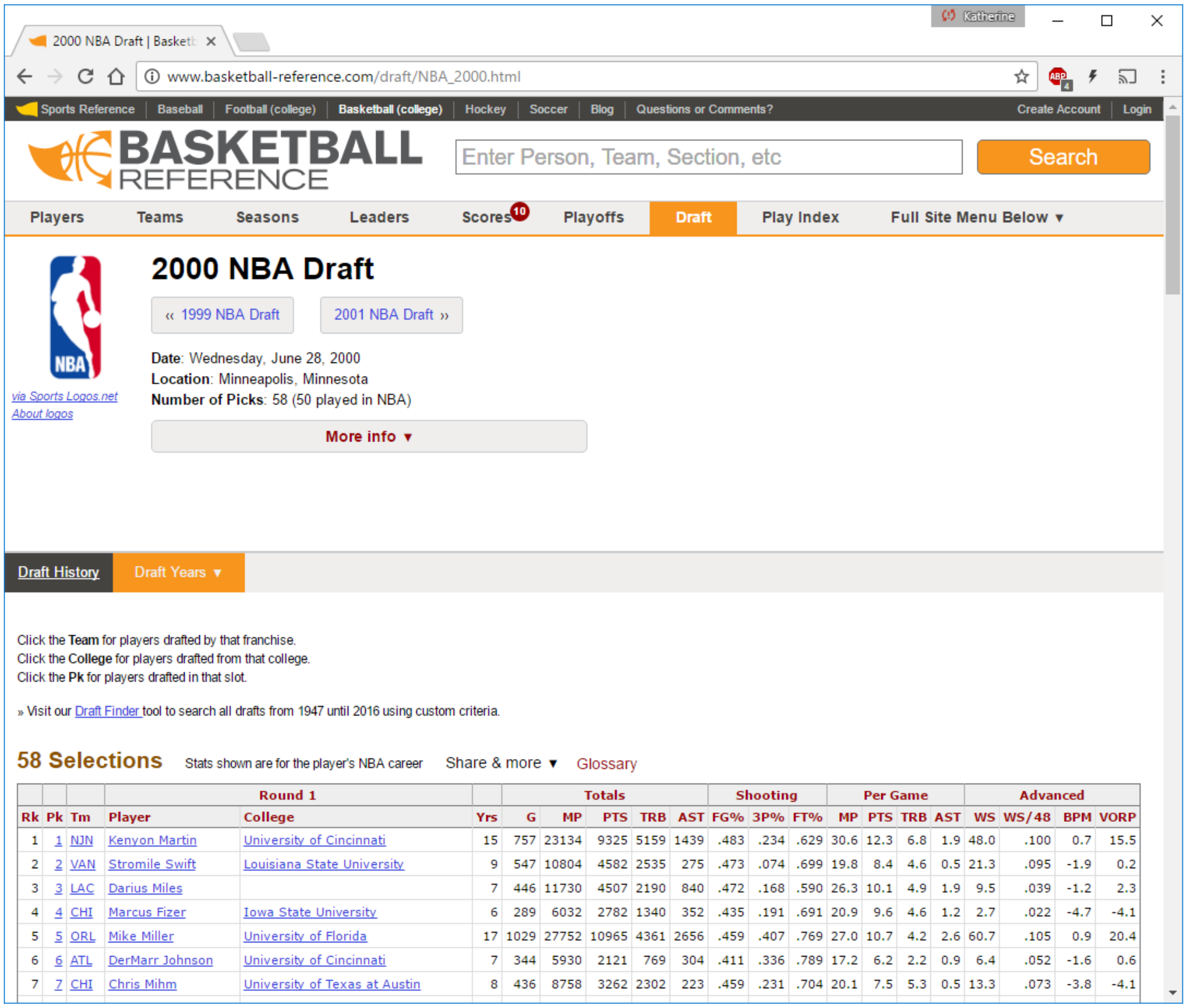
Note: With the Open Data movement, more and more data is available for download or through an API without the need for web scraping. Yay! Someone else compiled the data so that you don't have to! For example, if I were sad that all of my favorite lunch places on Franklin Street had been closing, I could go to the Chapel Hill open data portal (www.chapelhillopendata.org – specifically, https://www.chapelhillopendata.org/explore/dataset/downtown-tenant/?sort=tenant_name) and download a table of downtown tenants as a CSV file or Excel spreadsheet or JSON file ready for import into SAS or R and subsequent analysis (part of town, years open, independent vs. chain, etc.).

Using R

This annotated example illustrates one common and basic way to do web scraping with R (there are others). First I'll show and annotate running the code for one year of data appearing in a table on a web page. Then we will see how to use a loop to execute the same code for multiple years, resulting in more interesting data to analyze. These notes are written as if you are call R from SAS (as described in Resources / References / Calling R from SAS), which simplifies moving data between R and SAS. However, you can certainly use R on its own!

The goal of this example is to pull NBA draft data from the web (thanks to my former student Kevin Liao). The website being scraped is http://www.basketball-reference.com/ .

This is what the web page looks like for the year 2000. The first few lines of the table that we want to obtain show up at the bottom of this screen shot.



2000 NBA Draft

Date: Wednesday, June 28, 2000
Location: Minneapolis, Minnesota
Number of Picks: 58 (50 played in NBA)

Click the **Team** for players drafted by that franchise.
Click the **College** for players drafted from that college.
Click the **Pk** for players drafted in that slot.

» Visit our Draft Finder tool to search all drafts from 1947 until 2016 using custom criteria.

58 Selections Stats shown are for the player's NBA career

			Round 1			Totals					Shooting			Per Game				Advanced			
Rk	Pk	Tm	Player	College	Yrs	G	MP	PTS	TRB	AST	FG%	3P%	FT%	MP	PTS	TRB	AST	WS	WS/48	BPM	VORP
1	1	NJN	Kenyon Martin	University of Cincinnati	15	757	23134	9325	5159	1439	.483	.234	.629	30.6	12.3	6.8	1.9	48.0	.100	0.7	15.5
2	2	VAN	Stromile Swift	Louisiana State University	9	547	10804	4582	2535	275	.473	.074	.699	19.8	8.4	4.6	0.5	21.3	.095	-1.9	0.2
3	3	LAC	Darius Miles		7	446	11730	4507	2190	840	.472	.168	.590	26.3	10.1	4.9	1.9	9.5	.039	-1.2	2.3
4	4	CHI	Marcus Fizer	Iowa State University	6	289	6032	2782	1340	352	.435	.191	.691	20.9	9.6	4.6	1.2	2.7	.022	-4.7	-4.1
5	5	ORL	Mike Miller	University of Florida	17	1029	27752	10965	4361	2656	.459	.407	.769	27.0	10.7	4.2	2.6	60.7	.105	0.9	20.4
6	6	ATL	DerMarr Johnson	University of Cincinnati	7	344	5930	2121	769	304	.411	.336	.789	17.2	6.2	2.2	0.9	6.4	.052	-1.6	0.6
7	7	CHI	Chris Mihm	University of Texas at Austin	8	436	8758	3262	2302	223	.459	.231	.704	20.1	7.5	5.3	0.5	13.3	.073	-3.8	-4.1

Code for pulling data for the 2000 draft:

```
proc iml;
    submit / R;
            library(XML)
            url <- "http://www.basketball-reference.com/draft/NBA_2000.html"
            page <- htmlTreeParse(readLines(url), useInternalNodes=T)
            table <- readHTMLTable(page)$stats
            players <- subset(table, Player!="Player" & Yrs !="Totals")
            players$Draft_Yr <- 2000
    endsubmit;

    call ImportDatasetFromR("Work.NBA_Draft_2000", "players");
quit;
```

Of course, you could bring up R or RStudio and run the R code there instead of running it through SAS!

OK, what is each line of this R code doing?

[1] `library(XML)`

Load the XML library so that we can use this library's functions.

[2] `url <- "http://www.basketball-reference.com/draft/NBA_2000.html"`

This web page contains statistics for players drafted in the year 2000. Store the page's address (that is, its URL) in a variable named *url*.

[3] `page <- htmlTreeParse(readLines(url), useInternalNodes=T)`

In this nested sequence, readLines returns a vector whose elements are each line of the HTML source as a character string. You could submit

```
      test <- readLines(url)
      head(test, n=2)
```

to see the first couple of lines and make sure of this.

This vector is input to the htmlTreeParse function, which reads the lines of HTML into memory in a way that will allow them to be parsed. The parse-able structure in this code is named *page*. Using the option useInternalNodes=T (T for True) produces an object with additional structure and thus more options for parsing.

[4] `table <- readHTMLTable(page)$stats`

This line extracts the tabular data from the parsed HTML page and stores it in an object (specifically, a list – I found this out by submitting *typeof(table)*) named *table*. If we had simply submitted

```
table <- readHTMLTable(page)
```

submitting *names(table)* would reveal a top-level item *stats*. So we go a level deeper to the "real" data and create the list we really want in one step by submitting

```
table <- readHTMLTable(page)$stats
```

Now submitting *names(table)* would reveal the list of items we are really interested in (player name, college, pro team, years in the pros, total points scored in pro career, etc.), so we are good to go forward.

[5] `players <- subset(table, Player!="Player" & Yrs !="Totals")`

Taking this subset simply gets rid of heading rows where the *Player* column has the value "Player" or the *Yrs* column has the value "Totals" (!= means not equal in R).

[6] `players$Draft_Yr <- 2000`

This line adds a column named *Draft_Yr* to the list, and its value is the constant 2000.

[7] `call ImportDatasetFromR("Work.NBA_Draft_2000", "players");`

This is actually SAS code rather than R code and makes a SAS data set named *NBA_Draft_2000* from the R list (also works with data frames) named *players*. Note that most variables in this data set are character rather than numeric so we will have to make numeric versions of them before analysis, but that's not hard to do in SAS (we use the INPUT or INPUTN function).

OK, so now how can we upgrade this code to read the data from several draft years, say 2000 to 2018? This code will do it:

```
proc iml;
    submit / R;
        library(XML)

        years <- 2000:2018
        rdata <- NULL

        for (i in years) {
            url <- paste("http://www.basketball-reference.com/draft/NBA_", i,
".html", sep="")
            page <- htmlTreeParse(readLines(url), useInternalNodes=T)
            table <- readHTMLTable(page)$stats
            players <- subset(table, Player!="Player" & Yrs !="Totals")
            players$Draft_Yr <- i
            rdata <- rbind(rdata, players)
        }

    endsubmit;

    call ImportDatasetFromR("Work.NBA_Draft", "rdata");

quit;
```

What is different in this code?

(1) We set up the *years* array of the list of years we want to read.
(2) We initialize our *rdata* list to NULL before entering the loop.
(3) We use a FOR loop to loop through our array of years, reading data from each year, where *i* is the current year. The remaining differences all take place inside the loop.
(4) We use the paste function to construct the URL values (which have the nicely patterned names `http://www.basketball-reference.com/draft/NBA_year.html`). The R paste function is very useful, kind of like SAS's CATX function except with CATX we specify the separator as the first argument instead of the last.
(5) We set the *Draft_Yr* variable to the current value of *i*, which is the year.
(6) R's rbind is like SAS's SET, and here we use it to add the rows for the current year to the bottom of the *rdata* list.

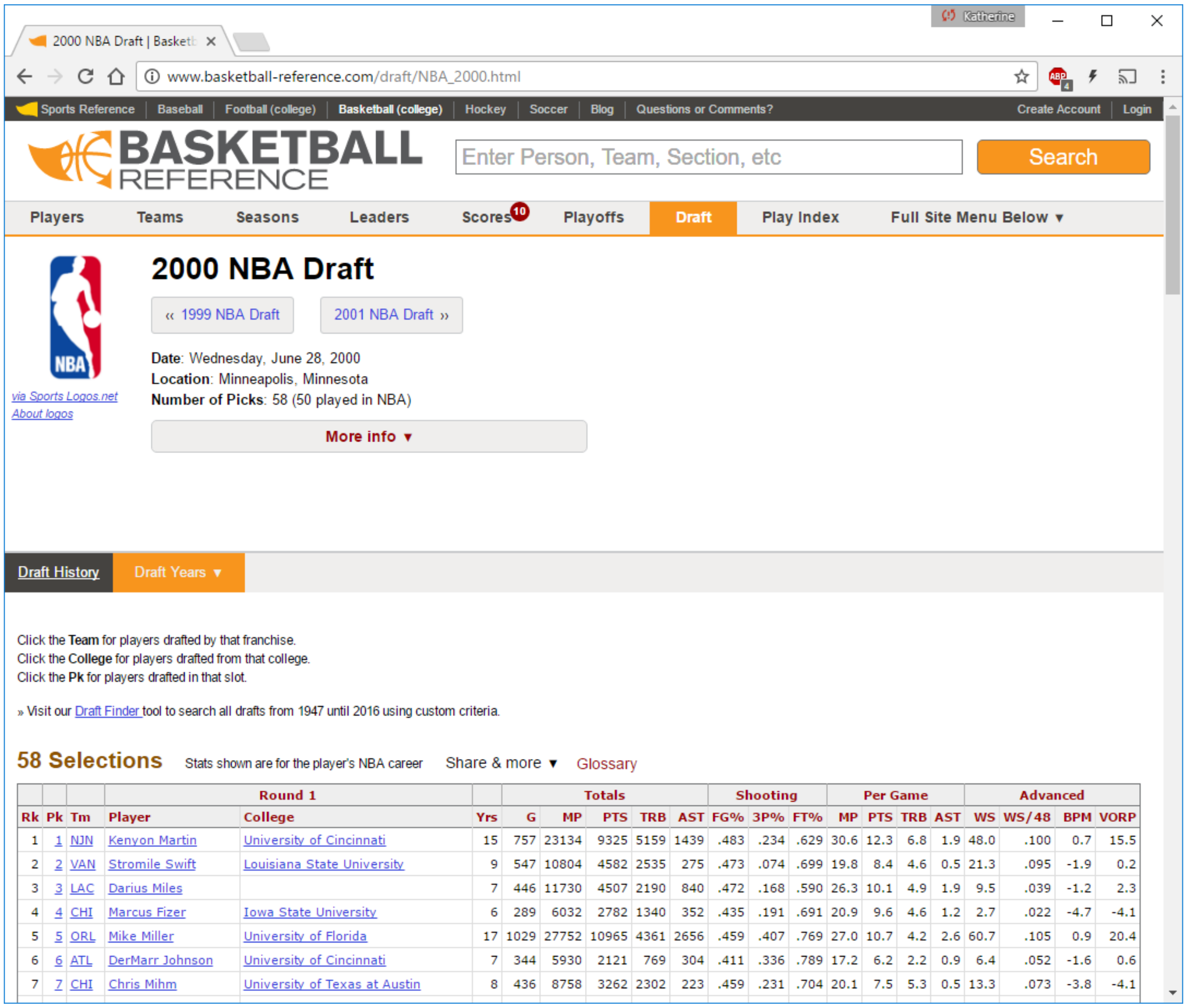
I've also posted some code (*using rvest to scrape espn.sas*) that uses a different package, *rvest*, to scrape UNC basketball data from ESPN's men's college basketball site. Note that this code causes an error for me – even when I run the R code in R Studio, not just when I run the code through SAS – but it works for other people I've asked to test it. [The failure occurs on the *html_table* line, in case you're interested.] *rvest* is a nice package because it lets you programmatically say which table from a page you want to read (tab[[1]], tab[[2]], etc.)

<u>Using JMP to read HTML tables</u>

JMP software is another software product out of SAS Institute, and like SAS, it is available to you for free as a UNC student. You can get versions that run natively on Windows or the Mac (it was originally Mac-only software). You can read more about JMP in the References section of the course Sakai site.

JMP is all-around cool software, and one of the coolest features is its ability to read HTML tables into SAS data sets so easily. Here are some screen shots of using JMP Pro 12 to access the 2000 NBA draft list referenced in the R section above. JMP Pro 13 works in exactly the same way.

As a reminder, here is what that web page looks like.


Katherine
2000 NBA Draft | Basketb
www.basketball-reference.com/draft/NBA_2000.html
Sports Reference | Baseball | Football (college) | Basketball (college) | Hockey | Soccer | Blog | Questions or Comments? | Create Account | Login
BASKETBALL REFERENCE
Enter Person, Team, Section, etc
Search
Players | Teams | Seasons | Leaders | Scores 10 | Playoffs | Draft | Play Index | Full Site Menu Below ▾
NBA
via Sports Logos.net
About logos
« 1999 NBA Draft
2001 NBA Draft »
More info ▾
Draft History
Draft Years ▾
Share & more ▾
Glossary

2000 NBA Draft

Date: Wednesday, June 28, 2000
Location: Minneapolis, Minnesota
Number of Picks: 58 (50 played in NBA)

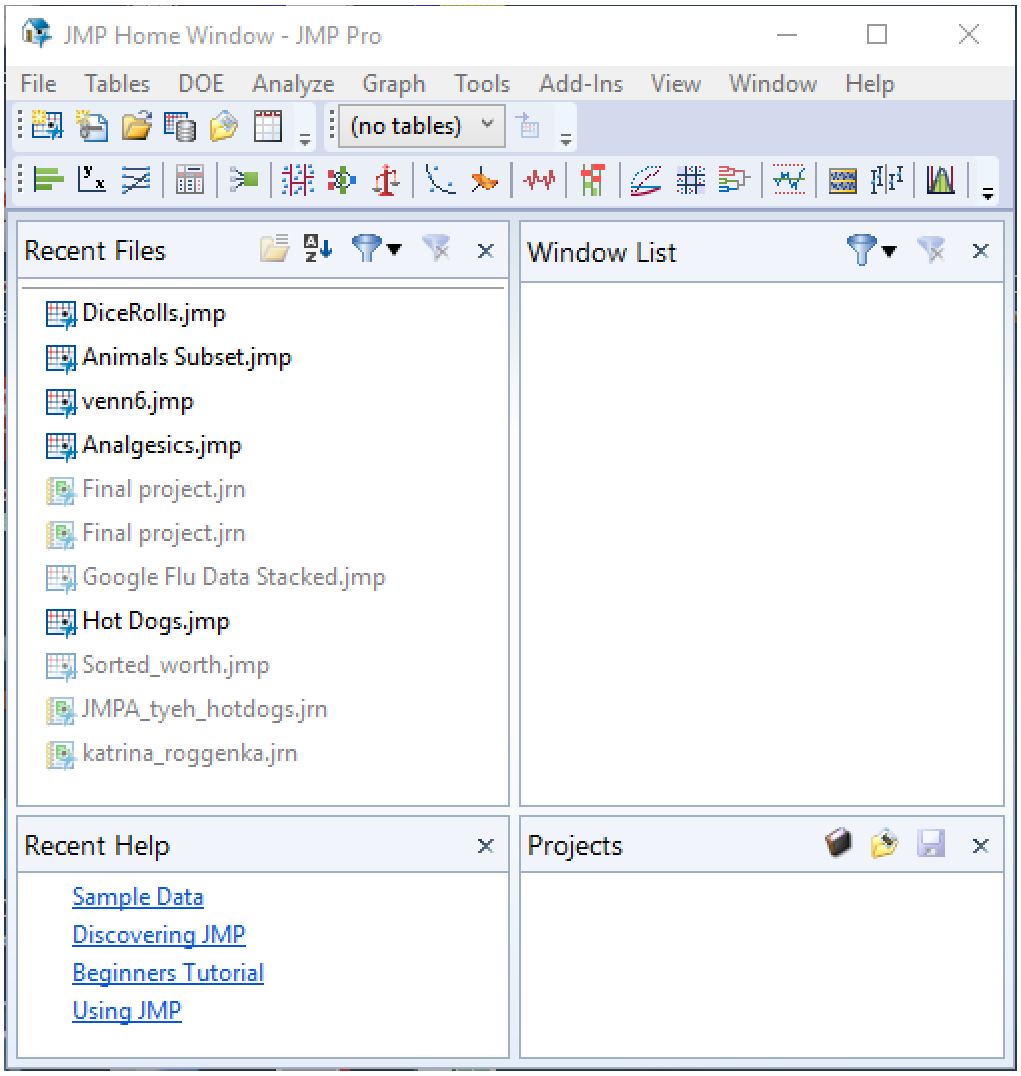
Click the **Team** for players drafted by that franchise.
Click the **College** for players drafted from that college.
Click the **Pk** for players drafted in that slot.

» Visit our Draft Finder tool to search all drafts from 1947 until 2016 using custom criteria.

58 Selections Stats shown are for the player's NBA career

			Round 1			Totals						Shooting			Per Game				Advanced			
Rk	Pk	Tm	Player	College	Yrs	G	MP	PTS	TRB	AST	FG%	3P%	FT%	MP	PTS	TRB	AST	WS	WS/48	BPM	VORP
1	1	NJN	Kenyon Martin	University of Cincinnati	15	757	23134	9325	5159	1439	.483	.234	.629	30.6	12.3	6.8	1.9	48.0	.100	0.7	15.5
2	2	VAN	Stromile Swift	Louisiana State University	9	547	10804	4582	2535	275	.473	.074	.699	19.8	8.4	4.6	0.5	21.3	.095	-1.9	0.2
3	3	LAC	Darius Miles		7	446	11730	4507	2190	840	.472	.168	.590	26.3	10.1	4.9	1.9	9.5	.039	-1.2	2.3
4	4	CHI	Marcus Fizer	Iowa State University	6	289	6032	2782	1340	352	.435	.191	.691	20.9	9.6	4.6	1.2	2.7	.022	-4.7	-4.1
5	5	ORL	Mike Miller	University of Florida	17	1029	27752	10965	4361	2656	.459	.407	.769	27.0	10.7	4.2	2.6	60.7	.105	0.9	20.4
6	6	ATL	DerMarr Johnson	University of Cincinnati	7	344	5930	2121	769	304	.411	.336	.789	17.2	6.2	2.2	0.9	6.4	.052	-1.6	0.6
7	7	CHI	Chris Mihm	University of Texas at Austin	8	436	8758	3262	2302	223	.459	.231	.704	20.1	7.5	5.3	0.5	13.3	.073	-3.8	-4.1

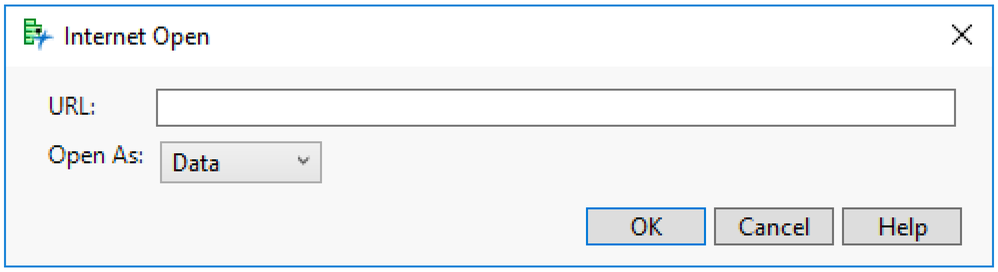
When you bring up JMP, you'll see a window like this (though without this collection of recent files):


JMP Home Window - JMP Pro
File Tables DOE Analyze Graph Tools Add-Ins View Window Help
(no tables)
Recent Files
Window List
Recent Help
Projects

DiceRolls.jmp
Animals Subset.jmp
venn6.jmp
Analgesics.jmp
Final project.jrn
Final project.jrn
Google Flu Data Stacked.jmp
Hot Dogs.jmp
Sorted_worth.jmp
JMPA_tyeh_hotdogs.jrn
katrina_roggenka.jrn

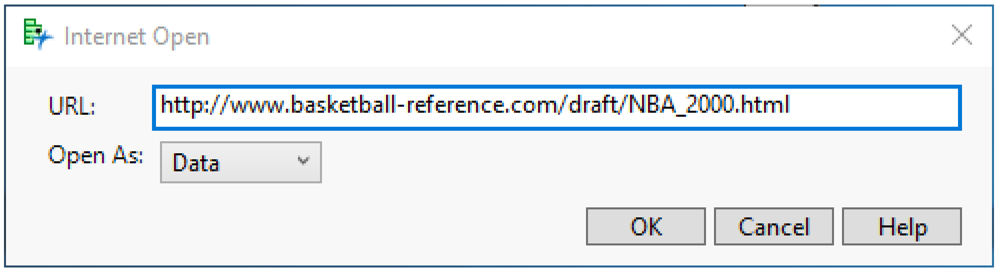
Sample Data
Discovering JMP
Beginners Tutorial
Using JMP

Selecting File -> Internet Open… produces this window:



URL:
Open As: Data

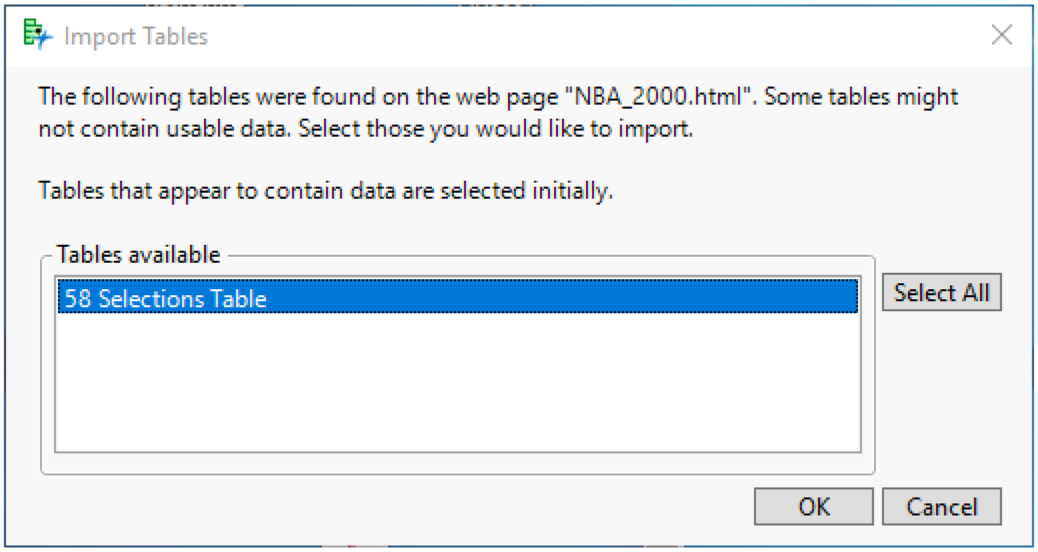
Fill in the URL of the 2000 NBA draft web page.



URL: http://www.basketball-reference.com/draft/NBA_2000.html

Open As: Data

As you might have noticed, this web page contains more than just a table, but JMP realizes that it's any table on a page that is mostly likely to contain data. So when you select OK, JMP presents you with a list of the HTML tables that are on the web page.



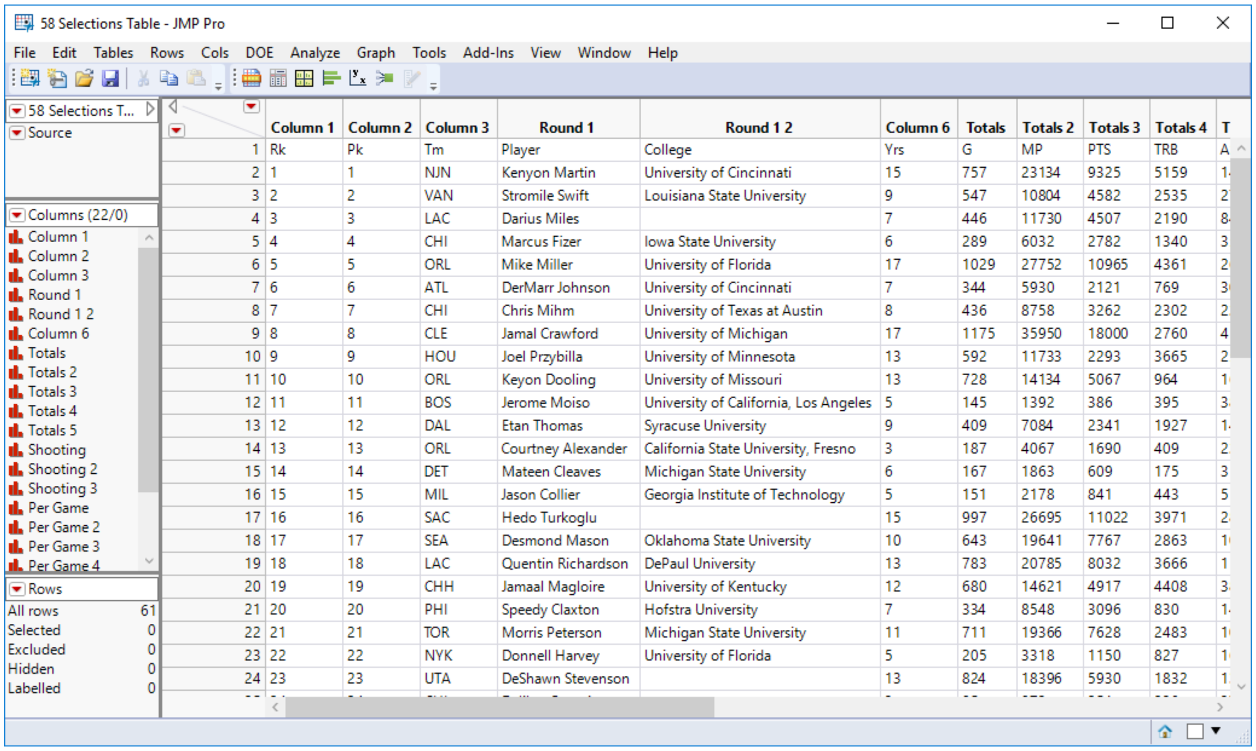
The following tables were found on the web page "NBA_2000.html". Some tables might not contain usable data. Select those you would like to import.

Tables that appear to contain data are selected initially.

Tables available

58 Selections Table

There is only one HTML table on this page so that is all that JMP presents. Leave that row selected and click OK. Voila, JMP reads the HTML table into a JMP table and displays it to you, ready to do any type of analysis of interest.



58 Selections Table - JMP Pro

File Edit Tables Rows Cols DOE Analyze Graph Tools Add-Ins View Window Help

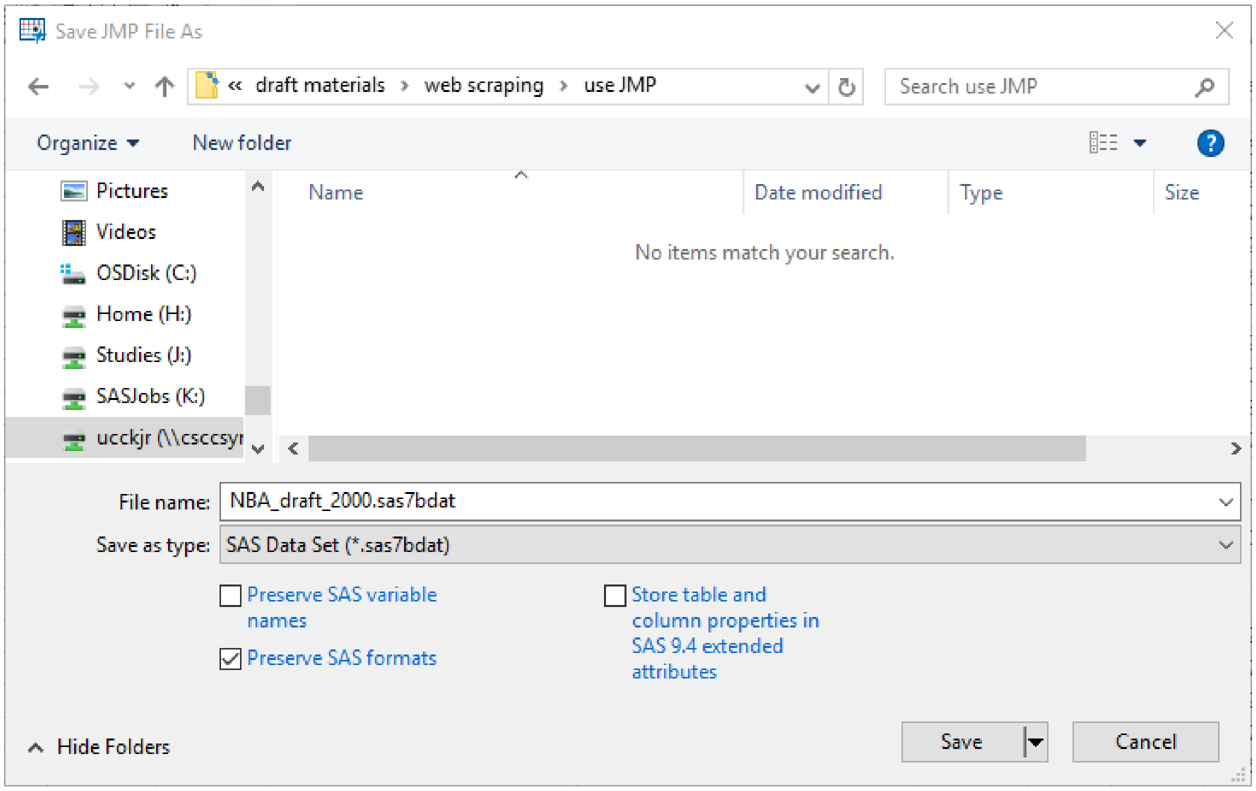
58 Selections T...
Source

Columns (22/0)
Column 1
Column 2
Column 3
Round 1
Round 1 2
Column 6
Totals
Totals 2
Totals 3
Totals 4
Totals 5
Shooting
Shooting 2
Shooting 3
Per Game
Per Game 2
Per Game 3
Per Game 4

Rows	
All rows	61
Selected	0
Excluded	0
Hidden	0
Labelled	0

	Column 1	Column 2	Column 3	Round 1	Round 1 2	Column 6	Totals	Totals 2	Totals 3	Totals 4	T
1	Rk	Pk	Tm	Player	College	Yrs	G	MP	PTS	TRB	A
2	1	1	NJN	Kenyon Martin	University of Cincinnati	15	757	23134	9325	5159	1
3	2	2	VAN	Stromile Swift	Louisiana State University	9	547	10804	4582	2535	2
4	3	3	LAC	Darius Miles		7	446	11730	4507	2190	8
5	4	4	CHI	Marcus Fizer	Iowa State University	6	289	6032	2782	1340	3
6	5	5	ORL	Mike Miller	University of Florida	17	1029	27752	10965	4361	2
7	6	6	ATL	DerMarr Johnson	University of Cincinnati	7	344	5930	2121	769	3
8	7	7	CHI	Chris Mihm	University of Texas at Austin	8	436	8758	3262	2302	2
9	8	8	CLE	Jamal Crawford	University of Michigan	17	1175	35950	18000	2760	4
10	9	9	HOU	Joel Przybilla	University of Minnesota	13	592	11733	2293	3665	2
11	10	10	ORL	Keyon Dooling	University of Missouri	13	728	14134	5067	964	1
12	11	11	BOS	Jerome Moiso	University of California, Los Angeles	5	145	1392	386	395	3
13	12	12	DAL	Etan Thomas	Syracuse University	9	409	7084	2341	1927	1
14	13	13	ORL	Courtney Alexander	California State University, Fresno	3	187	4067	1690	409	2
15	14	14	DET	Mateen Cleaves	Michigan State University	6	167	1863	609	175	3
16	15	15	MIL	Jason Collier	Georgia Institute of Technology	5	151	2178	841	443	5
17	16	16	SAC	Hedo Turkoglu		15	997	26695	11022	3971	2
18	17	17	SEA	Desmond Mason	Oklahoma State University	10	643	19641	7767	2863	1
19	18	18	LAC	Quentin Richardson	DePaul University	13	783	20785	8032	3666	1
20	19	19	CHH	Jamaal Magloire	University of Kentucky	12	680	14621	4917	4408	3
21	20	20	PHI	Speedy Claxton	Hofstra University	7	334	8548	3096	830	1
22	21	21	TOR	Morris Peterson	Michigan State University	11	711	19366	7628	2483	1
23	22	22	NYK	Donnell Harvey	University of Florida	5	205	3318	1150	827	1
24	23	23	UTA	DeShawn Stevenson		13	824	18396	5930	1832	1

In this case, we mostly want to create a SAS data set from this JMP table. We can do this by selecting File -> Save As… . In the window that opens, change the *Save as type* to SAS Data Set (*.sas7bdat), navigate to the location where you want to save the data set for later access, change the *File name* to a name that includes the information that this data is from the year 2000, and select Save.



File name: NBA_draft_2000.sas7bdat

Save as type: SAS Data Set (*.sas7bdat)

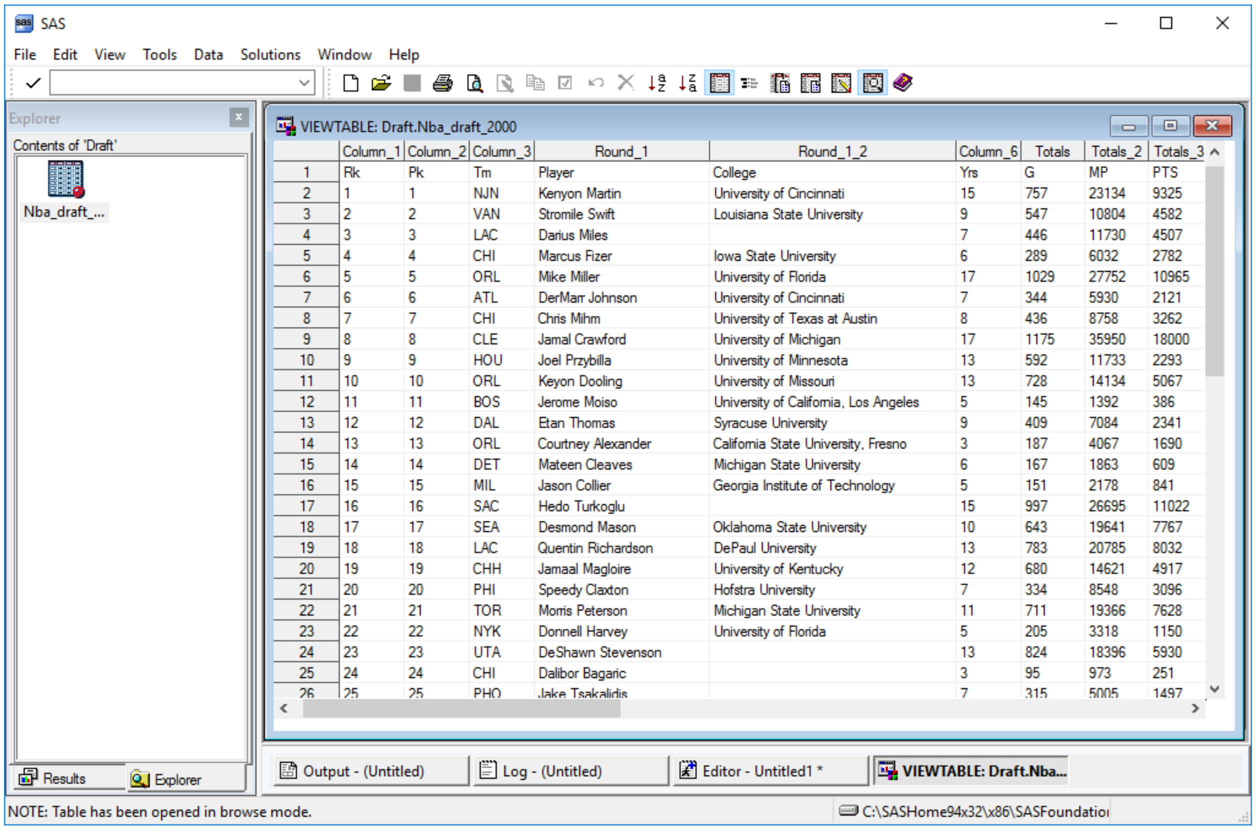
☐ Preserve SAS variable names

✓ Preserve SAS formats

☐ Store table and column properties in SAS 9.4 extended attributes

You might be presented with a JMP Alert dialog window that asks if it's OK to save even though you might lose formulas, formatting information, and metadata. We are OK with this, so select Yes.

If you now go to the folder where you saved the data set, you should see the new SAS data set. If you bring up SAS and assign a libref to the folder, the data set is all ready for you to use for analysis with SAS.



VIEWTABLE: Draft.Nba_draft_2000

	Column_1	Column_2	Column_3	Round_1	Round_1_2	Column_6	Totals	Totals_2	Totals_3
1	Rk	Pk	Tm	Player	College	Yrs	G	MP	PTS
2	1	1	NJN	Kenyon Martin	University of Cincinnati	15	757	23134	9325
3	2	2	VAN	Stromile Swift	Louisiana State University	9	547	10804	4582
4	3	3	LAC	Darius Miles		7	446	11730	4507
5	4	4	CHI	Marcus Fizer	Iowa State University	6	289	6032	2782
6	5	5	ORL	Mike Miller	University of Florida	17	1029	27752	10965
7	6	6	ATL	DerMarr Johnson	University of Cincinnati	7	344	5930	2121
8	7	7	CHI	Chris Mihm	University of Texas at Austin	8	436	8758	3262
9	8	8	CLE	Jamal Crawford	University of Michigan	17	1175	35950	18000
10	9	9	HOU	Joel Przybilla	University of Minnesota	13	592	11733	2293
11	10	10	ORL	Keyon Dooling	University of Missouri	13	728	14134	5067
12	11	11	BOS	Jerome Moiso	University of California, Los Angeles	5	145	1392	386
13	12	12	DAL	Etan Thomas	Syracuse University	9	409	7084	2341
14	13	13	ORL	Courtney Alexander	California State University, Fresno	3	187	4067	1690
15	14	14	DET	Mateen Cleaves	Michigan State University	6	167	1863	609
16	15	15	MIL	Jason Collier	Georgia Institute of Technology	5	151	2178	841
17	16	16	SAC	Hedo Turkoglu		15	997	26695	11022
18	17	17	SEA	Desmond Mason	Oklahoma State University	10	643	19641	7767
19	18	18	LAC	Quentin Richardson	DePaul University	13	783	20785	8032
20	19	19	CHH	Jamaal Magloire	University of Kentucky	12	680	14621	4917
21	20	20	PHI	Speedy Claxton	Hofstra University	7	334	8548	3096
22	21	21	TOR	Morris Peterson	Michigan State University	11	711	19366	7628
23	22	22	NYK	Donnell Harvey	University of Florida	5	205	3318	1150
24	23	23	UTA	DeShawn Stevenson		13	824	18396	5930
25	24	24	CHI	Dalibor Bagaric		3	95	973	251
26	25	25	PHO	Jake Tsakalidis		7	315	5005	1497

NOTE: Table has been opened in browse mode.

C:\SASHome94x32\x86\SASFoundatio

Of course, to download all 19 tables (if we want data from 2000-2018), we would have to do this 19 times (versus using R, where we could set up a loop). Still, this JMP capability is very cool and might be handy if all you need is one or two tables. Something to investigate is that JMP allows you to save a script based on your keystrokes as you complete a task. You can then rerun this script later to repeat that task. If you saved such a script, could you modify it to download the 2001 data, then the 2002 data, etc.?

One great thing about SAS programs is that they provide automatic reproducibility vs. doing things in a windowing environment such as JMP where you accomplish things by pointing and clicking. JMP scripts are one way around this weakness related to reproducibility of JMP activities and sequences.

If you want to try JMP, here are three ways you can obtain or access it. (1) It is available on the BIOS 669 VM at vcl.unc.edu. (2) You can obtain a copy from the university just as with SAS (JMP is much simpler to install). (3) You can download a free 30-day trial from https://www.jmp.com/en_us/software/download-jmp-free-trial.html. Amazingly great JMP learning resources are available in the JMP Learning Library at https://www.jmp.com/en_us/learning-library.html.

Using SAS

Using SAS for web scraping is based on the idea that an HTML file is a text file, which is apparent when you “view the source” of a web page. We read this text file into a SAS data set with one observation per line of the file. Then we can apply the power of the SAS DATA step, especially character or string functions (including regular expression functionality), to parse the records – keying on either regular words or on HTML beginning and ending tags since HTML is a mark-up language - and pull the information we want into DATA step variables.

A key to using SAS for basic web scraping is using the URL access method on a FILENAME statement, where the FILENAME statement points to a web page. Then you read that file in a DATA step using an INFILE statement signaling that the records are variable length (with maximum length 32767, which hopefully is long enough to hold each record). In the code below, LINE is a character variable with length 32767! Then we create additional variables based on parsing each line.

A simple example is presented in the referenced Zhu and Ghosh paper. If you go to webpage https://www.sas.com/en_ca/user-groups/alberta-usergroups/edmonton-archive.html (the page for the Edmonton, Canada SAS User group presentation archive), you’ll see a list of presentations made to the user group over the years. Since this list is presented in an orderly, systematic way, the HTML that produced it is probably also orderly and systematic. If you view the page source (in Chrome, right click on the body of the page and select *View page source*), you can find the HTML code that produces this list of presentations near the bottom (search for Presentations). Can we pull the paper titles, author names, and presentation dates from the page source?

[Note that the way website materials are organized for Canadian SAS user groups has changed over the years. The material in these 2019 notes matches the current web page organization, while the video reflects a previous organization. The SAS web scraping principles illustrated are the same however. The possibility of such web page changes is one of the perils of web scraping – people can change web page organization at any time, requiring changes to your web scraping code should you want to rerun it.]

A few lines from the relevant section of HTML code:

<p>Data Visualization with SAS: Matt Joyce, SAS Canada (October 2017)</p>

<p>Macro Variables from Excel for SAS Scripts: Sylvia So, Alberta Health (October 2017)</p>

Here is SAS code that can be used to parse this HTML file, creating as variables the presentation name, the author(s), and the presentation date. In this case we are finding only presentations done by someone named Matt. The function of each line of this program is provided following the code.

```
/*  1 */ FILENAME eSUG URL "https://www.sas.com/en_ca/user-groups/alberta-
usergroups/edmonton-archive.html";

/*  2 */ DATA eSUG_archive(KEEP=title author date);
/*  3 */      LENGTH Title $200 Author $100 Date $50;

/*  4 */      INFILE eSUG LENGTH=len LRECL=32767;
/*  5 */      INPUT line $VARYING32767. len;
/*  6 */      *IF FIND(line,'Matt') THEN DO;  /* equivalent */
              IF PRXMATCH('/Matt/',line) THEN DO;
/*  7 */          * PUT line= ;               /* can help during code development */
/*  8 */          title = SUBSTR(line, 4, INDEX(line,":")-4);
/*  9 */          author = SUBSTR(line, INDEX(line,":")+1, (INDEX(line,',')-
INDEX(line,":")-1));
/* 10 */          date = SUBSTR(SCAN(line,-2,'<('), 1, LENGTH(SCAN(line,-2,'<('))-1);
/* 11 */          OUTPUT;
/* 12 */      END;

         RUN;

/* 14 */ FILENAME eSUG CLEAR;
```

OK, what is each line of this SAS code doing?

[1] `FILENAME eSUG URL "https://www.sas.com/en_ca/user-groups/alberta-usergroups/edmonton-archive.html";`

Set up the eSUB fileref to point to the web page of interest (the INFILE statement below references eSUB). Using URL as the access method lets SAS know that it needs to connect to an external web server to find this file.

[2] `DATA eSUG_archive(KEEP=title author date);`

We are naming our new data set eSUG_archive, and we want that data set to contain only variables Title, Author, and Date.

[3] `LENGTH Title $200 Author $100 Date $50;`

Set up the lengths of these three character variables that we will be creating – be sure to make them long enough to hold any conceivable value.

[4] `INFILE eSUG LENGTH=len LRECL=32767;`

Tells SAS to read data from the file referenced by eSUG (this would be the web page HTML file). The option `LRECL=32767` tells SAS that the maximum length of a record is 32767 (well, this is the longest a SAS variable can be, so we hope it is long enough for each record of our HTML file – if not, some records will be truncated). The option `LENGTH=len` stores the actual length of each record in variable LEN, which is needed in our INPUT statement because we are using the VARYING informat (see [5]).

[5] `INPUT line $VARYING32767. len;`

Tells SAS to read each line of the file into the single variable LINE. `$VARYING32767.` is an informat that tells SAS that the line will be of variable length, up to 32767 characters. When using the VARYING informat, you must specify a variable containing the real length directly after, and that's what LEN is (see [4]).

[6] `IF FIND(line, "Matt") THEN DO;` or equivalently
`IF PRXMATCH("/Matt/", line) THEN DO;`

Locate lines containing *Matt* and run code (the DO group) for each such line.

[7] `* PUT line= ;`

Writing lines of interest to the SAS log can really help during code development.

[8] `title = SUBSTR(line, 4, INDEX(line,":")-4);`

Pulls off the string after <p> and before : . There would be many other ways to code this, and it takes experimentation.

[9] `author = SUBSTR(line, INDEX(line,":")+1, (INDEX(line,',')-INDEX(line,":")-1));`

This code is designed to pick up the words between the : that follows each title and the , that follows each author name. This code is definitely not perfect but generally works for the current task.

[10] `date = SUBSTR(SCAN(line,-2,'<('), 1, LENGTH(SCAN(line,-2,'<('))-1);`

In this nested sequence, SAS first scans the line for the second word from the end when using only < and (as the delimitors. The desired date substring starts at that point (position 1 of the scanned "word") and has a length of 1 less than that total substring [to get rid of the) on the end].

[11] `OUTPUT;`

For each line containing *Matt*, output a record.

[12] `END;`

End the section of code to run whenever a line contains *Matt*.

[13] `FILENAME eSUG CLEAR;`

Clean up. It's good form to release a file when you are finished with it. This is especially true when the file is a web page – web hosting services aren't always happy to have their pages crawled.

If you look at the resulting data set, the variables are in pretty good shape - but not perfect since one of the presentation titles contained a : . It's hard to be perfect when web scraping strings!

Reference:

George Zhu and Sunita Ghosh, Accessing and Extracting Data from the Internet Using SAS, http://support.sas.com/resources/papers/proceedings12/121-2012.pdf

Using Python (Beautiful Soup)

This Python example illustrates two important points:

1. With just a little bit of knowledge you can use Python to do web scraping. I am NOT a Python programmer, so if I can do it, you can do it!
2. How tenuous and sketchy web scraping is compared with downloading prepared data (someone has already compiled one or more CSV or text files) or using an API.

Before this year, my Python example scraped data from www.wunderground.com, and I used code provided in the book Visualizing Data by Nathan Yau. However, in the summer of 2018 wunderground drastically changed the layout of its historical web pages so that they are no longer scrape-able (and actually now wunderground has an API).

So I needed to find another example for Python web scraping or drop this section of the notes entirely. Well, I lucked out and discovered a good web page to play around with for Python web scraping, and it has worked out well for an example.

I have a 2006 Honda Civic Hybrid, one where the electric motor is in a supporting role and is charged during braking. It cannot run in pure electric mode like some other hybrids. It has been a great car, but if/when I need to buy another car, I'm considering a plug-in hybrid. This type of car has both electric and gasoline motors and can run in pure electric mode for a certain number of miles, and you charge it by plugging into an appropriate outlet or charging station. But if you want to go on a long trip without worrying about being able to charge up, you can run on the gasoline motor.

I found (in March 2019) a site https://www.plugincars.com/cars that lists all pure electric cars and plug-in hybrids available in the United States. Over 40 cars are listed in a systematic way, and the information of interest (to me) is embedded in the source HTML (rather than pulled from a database as the web page is built), so the web page is a good candidate for web scraping.

Setting up for our task

If you would like to give this a try, let's get started. I've gotten Python scraping to work on both a PC and a Mac. The main notes focus on Windows, with a special section at the bottom of the Python notes on getting things to work on a Mac. Another special section at the bottom concerns running Python through Anaconda (the Spyder application), and if you've already got Anaconda that's definitely the way you should go – no additional installation is necessary!

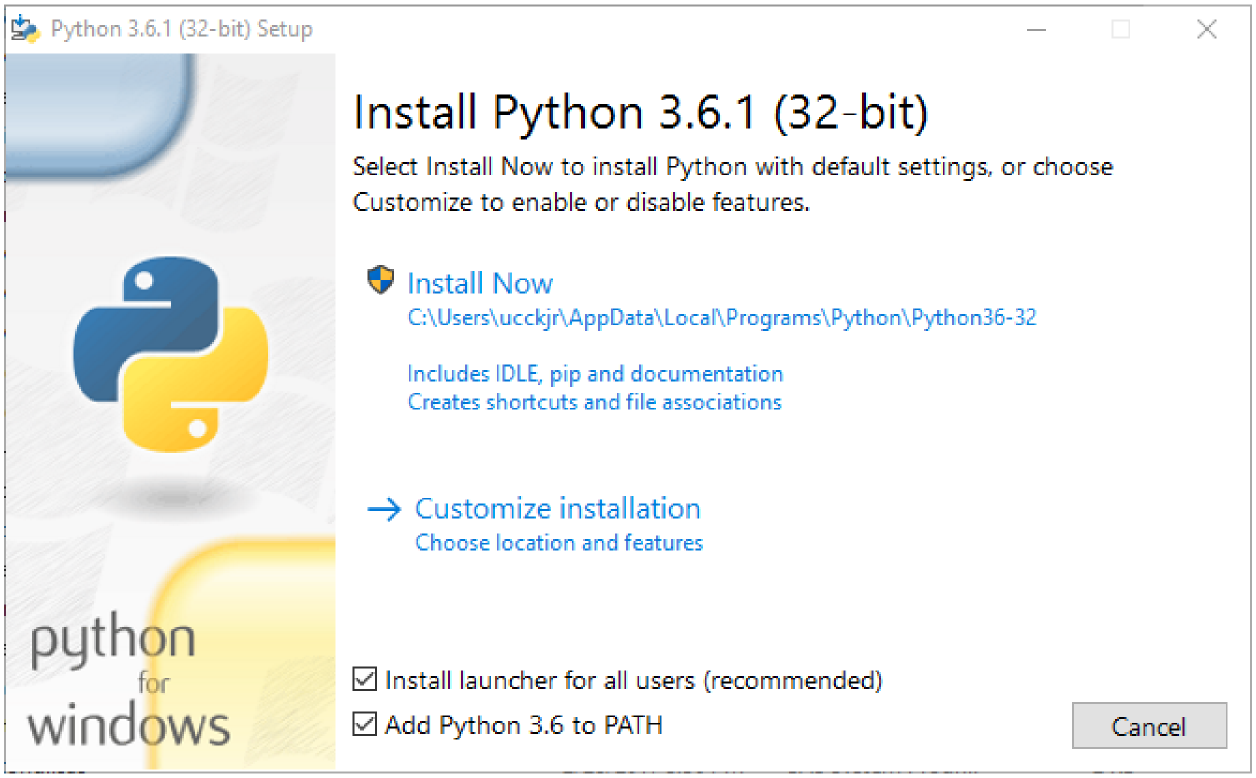
What we are aiming for in this case is a text file of information on all cars listed on plugincars.com. We will get this information by web scraping the site. Once we have this text file, we can easily create a SAS data set from it, to use for graphing or whatever.

So, how about Python? Mac OS X users, you should have Python already. To make sure, open the Terminal application and type *python* to see whether it's found.

PC users, you'll need to start by installing Python on your PC. Python is open source, so you can simply go to https://www.python.org/downloads/ and download it. Lots of releases are available, and perhaps any of them will work for you – I can't say. The version I happen to be running on my PC is 3.5.0, which does work with the release of Beautiful Soup that I have, and that's all that matters.

An important Python install note: Early during the install, watch for a checkbox offering to add the Python path to the system path variable. This checkbox is not selected by default, but you definitely want to select it. Making the Python path part of the system path variable will enable the Python command to be recognized from any directory.

Here's a sample install setup window with that checkbox selected:



Install Python 3.6.1 (32-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\ucckjr\AppData\Local\Programs\Python\Python36-32

Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

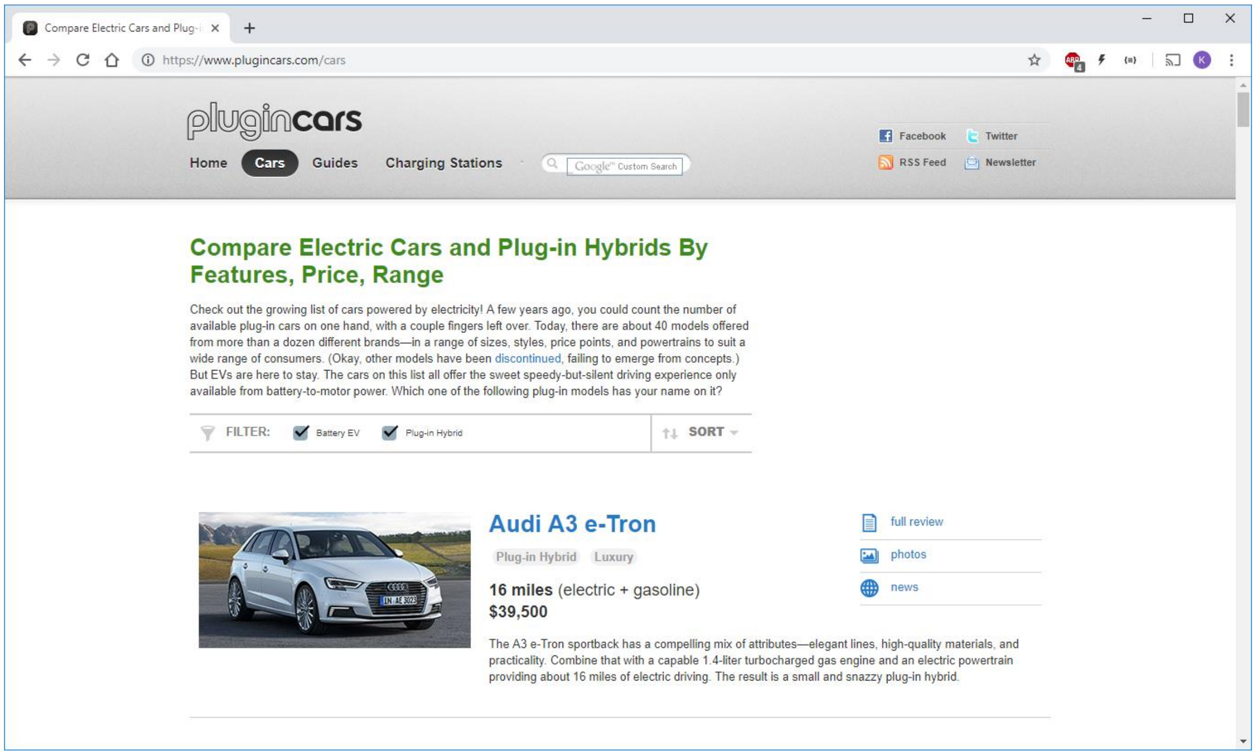
☑ Install launcher for all users (recommended)
☑ Add Python 3.6 to PATH

You'll need to download Beautiful Soup separately from Python itself. I have Beautiful Soup 4 (bs4), which you can download from https://pypi.python.org/pypi/beautifulsoup4. Assuming you have this version, put the bs4 folder in the directory where you will put your Python programs. Putting the bs4 folder in the same location as your Python programs means that you don't need to do a formal "install" of Beautiful Soup.

Let's say you have a directory C:\myPython where you will store your Python programs. Put the bs4 folder in there. Also, grab programs test-python-scrape.py and scrape-plug-in-hybrid-info.py from Sakai and store them in that directory.

Looking at the site we want to scrape

Here's a look at the site to be scraped, https://www.plugincars.com/cars. Scrolling down reveals dozens more cars, with the layout for each car paralleling what we see below for the Audi A3 e-Tron.


Compare Electric Cars and Plug-i

https://www.plugincars.com/cars

plugincars

Home | Cars | Guides | Charging Stations

Facebook | Twitter | RSS Feed | Newsletter

Compare Electric Cars and Plug-in Hybrids By Features, Price, Range

Check out the growing list of cars powered by electricity! A few years ago, you could count the number of available plug-in cars on one hand, with a couple fingers left over. Today, there are about 40 models offered from more than a dozen different brands—in a range of sizes, styles, price points, and powertrains to suit a wide range of consumers. (Okay, other models have been discontinued, failing to emerge from concepts.) But EVs are here to stay. The cars on this list all offer the sweet speedy-but-silent driving experience only available from battery-to-motor power. Which one of the following plug-in models has your name on it?

FILTER: ✓ Battery EV ✓ Plug-in Hybrid | SORT

Audi A3 e-Tron

Plug-in Hybrid | Luxury

16 miles (electric + gasoline)
$39,500

The A3 e-Tron sportback has a compelling mix of attributes—elegant lines, high-quality materials, and practicality. Combine that with a capable 1.4-liter turbocharged gas engine and an electric powertrain providing about 16 miles of electric driving. The result is a small and snazzy plug-in hybrid.

full review
photos
news

If we view the source for this page (in Chrome, right click on the page and select *View page source*), we can use Ctrl-F and search for *Audi A3 e-Tron*. From examining the source code, this Audi model and each other car seems to be embedded in a "table data" (<td> </td>) section having class="carz-text-area". Here is the section for the Audi A3 e-Tron:

```
  <td class="carz-text-area" valign="top">

        <div class="carz-text-sidebar">

          <p class="carz-sidebar-item"><a href="/audi-a1-e-tron"><img
src="/sites/all/themes/plugincars/images/carz-2014-august/icon_fullreview.png"/> full review</a></p>

          <p class="carz-sidebar-item"><a href="/audi-e-tron-photos-videos.html"><img
src="/sites/all/themes/plugincars/images/carz-2014-august/icon_photos.png"/> photos</a></p>
<p class="carz-sidebar-item"><a href="/a1-audi-e-tron/news"><img
src="/sites/all/themes/plugincars/images/carz-2014-august/icon_news.png"/> news</a></p>

        </div>

        <h3><a href="/audi-a1-e-tron">Audi A3 e-Tron</a></h3>

        <p class="type">
```

```
          <span class="field-content">Plug-in Hybrid</span>       <span class="field-
content">Luxury</span>           </p>

        <p class="highlights">

          <strong>16 miles </strong>

                (electric + gasoline)<br/>

          <strong><span class="field-content">$39,500</span></strong><br/>

        </p>

        <!-- ?php echo $fields['field_highlights_value']->content; ? -->

        <span class="blurb" style="clear: both"><div class="field-content"><p>The A3 e-Tron sportback
has a compelling mix of attributes—elegant lines, high-quality materials, and practicality. Combine that
with a capable 1.4-liter turbocharged gas engine and an electric powertrain providing about 16 miles of
electric driving. The result is a small and snazzy plug-in hybrid.</p>

</div></span>

  </td>
```

The information of interest to me is all included in this section, as highlighted in cyan above. But there is no obvious pattern to the tags of the items I want to pull out. BeautifulSoup is oriented toward traversal of an HTML "tree", but I couldn't figure out how to do that here. Fortunately, I read about BeautifulSoup's "stripped_strings" method and was able to use that to grab the information of interest (as well as extraneous info, but we will deal with that later). Once I had the values, I could print them or write them to a file. I knew how to write a Python loop from my wunderground scraping, so I was able to loop through all 47 cars and write out their information to a text file. Then I could use a SAS program to read the text file, convert it to a usable form, and produce graphs and other helpful output.

Looking at our test program

Let's start with a simple program to prove that we can read this page source and do something with it. Our test program here is test-python-scrape.py.

```
import urllib.request

page = urllib.request.urlopen("https://www.plugincars.com/cars")

from bs4 import BeautifulSoup

soup = BeautifulSoup(page, "html.parser")
```

```
carz_all = soup.find_all(class_="carz-text-area")
#print(carz_all)

for string in carz_all[2].stripped_strings:
    print(string)
```

A line-by-line look:

```
import urllib.request
```

Tells python to load the library for accessing URLs

```
page = urllib.request.urlopen("https://www.plugincars.com/cars")
```

Requests this specific URL (HTML page). All of the lines of HTML are loaded into the *page* variable.

```
from bs4 import BeautifulSoup
```

Imports the Beautiful Soup library to help us parse the HTML. bs4 is the name of a folder in the same directory as our program, and it contains our Beautiful Soup files as described above.

```
soup = BeautifulSoup(page, "html.parser")
```

Variable *soup* contains the HTML (as stored in variable *page*) as "parsed" by Beautiful Soup.

```
carz_all = soup.find_all(class_="carz-text-area")
#print(carz_all)
```

As noted earlier, the information for each car is located in a section of HTML code having the class "carz-text-area". soup.find_all finds all incidents of this class and writes the information to a data structure we are naming carz_all. To see the data structure, you could print it (the command is currently commented out).

```
for string in carz_all[2].stripped_strings:
    print(string)
```

The indexes of Python data structures start at 0, and here we are calling for carz_all[2]. That should produce information about the third car listed on the website, the BMW 330e. The stripped_string method returns all strings contained as values in the data structure, and we ask for those strings to be printed.

Here is what gets printed to our console upon running this program:

```
full review
photos
BMW 330e
```

```
Plug-in Hybrid
Sedan
14 miles
(electric + gasoline)
$44,100
If you like the styling and road manners of a BMW 3-Series, but want to push
the envelope on efficiency, then the 33e is the answer.  Commutes of less
than 14 miles can happen purely on electricity, with an official 72 MPGe
rating.  Punch the accelerator for combined power from a 2-liter turbocharged
engine and 87-hp electric motor.  That results in nearly as much torque as
the six-cylinter 340i at a lower purchase price after federal incentives.
```

This does contain the information we need, and for the BMW 330e as we had anticipated! We'll just need to do a little more to make the information useful, and that we can do with SAS.

Running our test program

Assuming that during the Python install you did ask the installer to make the Python path part of the system path variable, running your Python program is simple. You need to open a command window (on a PC) or a terminal window (on a Mac) in the directory containing your Python programs (or else use *cd* to navigate to that directory after you open the window). On a PC, a simple way to open a command window in a particular directory is to shift-right-click on the directory name in the Windows Explorer. Select *Open command window here* (or *Open PowerShell window here*) from the resulting list. When the command window opens, you'll be able to tell whether you are in the appropriate directory. If you are, simply type *python* followed by the name of the program (*python test-python-scrape.py*) and hit enter to run it. You should get the results shown above.

If you do, YAY! You have just been successful in web scraping with Python.

Preparing and running our real program

Compared to our test program, in our real program we are interested in information on all cars, and we want to write that information to a file rather than printing it. Program scrape-plug-in-hybrid-info.py does this. It's actually only a little bit more complicated than our test program.

```
import urllib.request

page = urllib.request.urlopen("https://www.plugincars.com/cars")

from bs4 import BeautifulSoup

soup = BeautifulSoup(page, "html.parser")
```

```
 8
 9 carz_all = soup.find_all(class_="carz-text-area")
10 #print(carz_all)
11
12 #for string in carz_all[2].stripped_strings:
13 #    print(string)
14
15 len(carz_all)
16 #47
17
18 #proof of concept
19 #for m in range(0,47):
20 #    for string in carz_all[m].stripped_strings:
21 #        print(string)
22
23 f = open('evinfo.txt', 'w')
24 for m in range(0,47):
25     for string in carz_all[m].stripped_strings:
26         f.write(string + '\n')
27 f.close()
```

Any line beginning with # is a comment. Comments here are lines from the test program, left in for reference, or lines that I ran during program development as a proof of concept (for example, could I loop through all cars using range (0,47)).

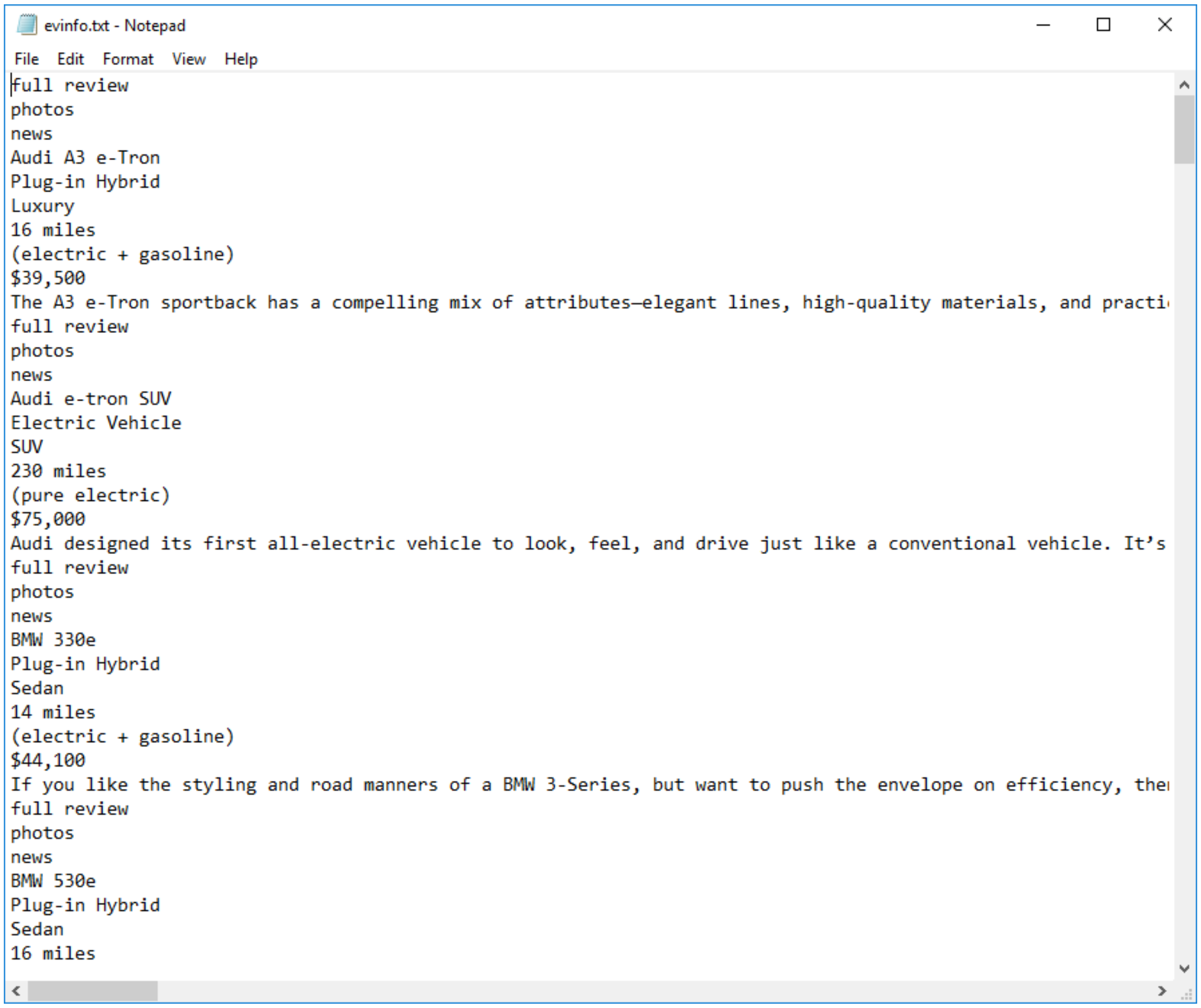
New here are lines 15 and 23-27. Line 15 shows using the LEN function to get the size of a Python data structure. 47 is returned, so our loop needs to index from 0 (the Python starting point) to 47.

Line 23 opens up a file named evinfo.txt for write access. This file will be written to the directory from which you are running Python.

Lines 24-26 loop through all cars in the carz-all data structure, pulling out all strings and writing each string to evinfo.txt. \n requests that each string be put on a new line.

Line 27 closes the file.

And that's it! If the program succeeds and you open up evinfo.txt, here's what you'll see:


evinfo.txt - Notepad

File Edit Format View Help

```
full review
photos
news
Audi A3 e-Tron
Plug-in Hybrid
Luxury
16 miles
(electric + gasoline)
$39,500
The A3 e-Tron sportback has a compelling mix of attributes—elegant lines, high-quality materials, and practi
full review
photos
news
Audi e-tron SUV
Electric Vehicle
SUV
230 miles
(pure electric)
$75,000
Audi designed its first all-electric vehicle to look, feel, and drive just like a conventional vehicle. It's
full review
photos
news
BMW 330e
Plug-in Hybrid
Sedan
14 miles
(electric + gasoline)
$44,100
If you like the styling and road manners of a BMW 3-Series, but want to push the envelope on efficiency, the
full review
photos
news
BMW 530e
Plug-in Hybrid
Sedan
16 miles
```

Using SAS to make the desired data set

This text file contains more data about each car than I need, but I can get rid of extra information when making my SAS data set.

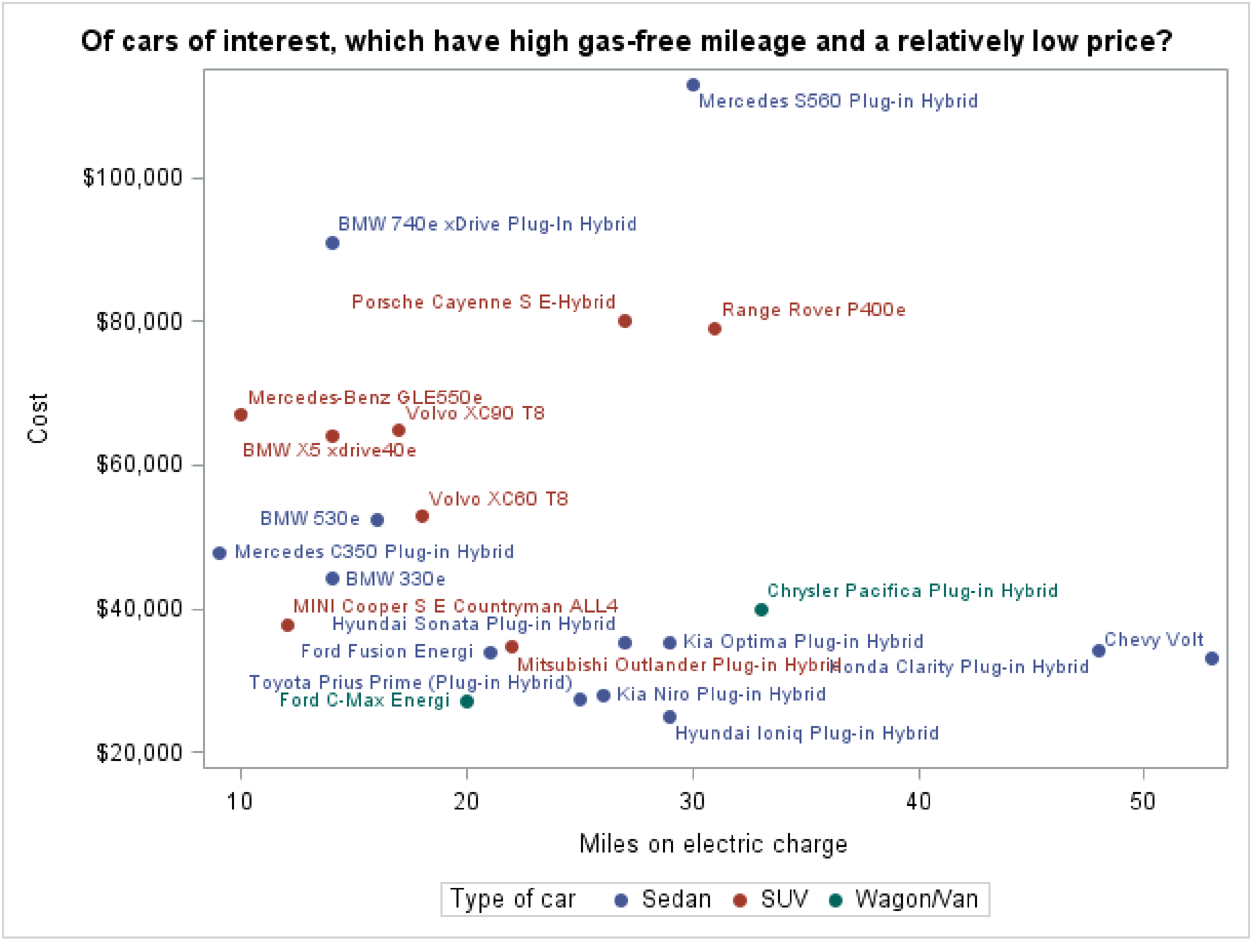
The needed SAS program is kind of sketchy and depends on there being exactly 10 rows per car. Yuck! It turned out that only one car – the Honda Clarity Electric – did not have ten rows in evinfo.txt, missing price information on the website (which turned out to be because it is only for lease, not purchase). So I had to add a bogus row for that car/item in order to be able to transpose the data set reliably. Also, I had to add a car counter and then a row counter for purposes of transposition. So it's a good thing that the scraped data was as regular as it was! I think I was lucky.

Anyway, here's my SAS code to transform the text file into a SAS data set of 47 observations, one per car, ready for my analysis as I try to figure out what car I might want to buy next.

```
/* data scraped from https://www.plugincars.com/cars */

filename in 'P:\Sakai\Sakai 2019\Course units\08 Web scraping\evinfo.txt';
data readin;
    infile in length=len lrecl=2000;
    input line $varying1000. len;

    * prepare to insert line 169 with price info for Honda Clarity Electric;
    nobs1 = _n_;
    nobs=nobs1;
    if nobs1>=169 then nobs=nobs1+1;
run;
data fix;
    nobs=169;
    line='TBD';
run;
data all;
    set readin fix;
    by nobs;
run;

* keep only rows with car name, ev type, car type, miles per charge, fuel
types, and price;
data keep;
    set all;
    if mod(nobs,10) in (4,5,6,7,8,9);
run;

* generate indexes to help with organized transposition into one row per car;
data generate_type;
    do car=1 to 47;
        do j=1 to 6;
            output;
        end;
    end;
run;

* add indexes to main data - yikes, merge with no BY! but rows match one-to-
one I AM SURE;
data addtype;
    merge generate_type keep;
    drop nobs1;
run;

* now collapse to one row per car;
data collapse;
```

```
    set addtype;
    by car;

    length car_name $40 ev_type $20 car_type $20 miles $20 fuel $30 coststr $20;

    retain car_name ev_type car_type miles fuel coststr;

    if first.car then do;
        car_name=' '; ev_type=' '; car_type=' '; miles=' '; fuel=' '; coststr=' ';
    end;

    if j=1 then car_name=line;
    if j=2 then ev_type =line;
    if j=3 then car_type=line;
    if j=4 then miles   =line;
    if j=5 then fuel    =line;
    if j=6 then coststr =line;

    if last.car then output;

    keep car car_name ev_type car_type miles fuel coststr;
run;

* make needed numeric variables from strings;
data fixup;
    set collapse;

    n_miles = input(scan(miles,1),best.);

    if coststr='TBD' then cost=.;
    else cost = input(coststr,dollar10.);
    format cost dollar10.;
run;

title 'Of cars of interest, which have high gas-free mileage and a relatively
low price?';
proc sgplot data=fixup;
    where fuel='(electric + gasoline)' and ^missing(cost) and car_type ^in
('Luxury','Coupe');
    label cost='Cost'
          n_miles='Miles on electric charge'
          car_type='Type of car';
    scatter x=n_miles y=cost / group=car_type
                               markerattrs=(symbol=CircleFilled)
                               datalabel=car_name ;
run;
```

Here is the resulting plot. It looks like the Chevy Volt could be the car for me (except that GM recently discontinued the Volt ☹).



Of course, by the time I am ready to buy a car in a few years, not only will the list of available cars and their information have changed, but the web page's HTML source format might have changed! So maybe I should go ahead and buy a new car this year (not!).

Special notes for Mac users: Depending on the age of your Mac, the version of Python on your Mac could be too old for Beautiful Soup 4. That's fine, older versions of Beautiful Soup are available and work great. To find out which version of Python is on your Mac, in the terminal application enter the command *python --version* (that's two dashes before version). What version you have determines what version of Beautiful Soup will work for you (per information on the Beautiful Soup website). For example, my old iMac (before it died) had Python 2.7.10, so I needed to download Beautiful Soup-3.2.1. I put the Beautiful Soup-3.2.1 folder into the same folder as my Python programs and renamed it BeautifulSoup. Then I changed lines of our testing program as follows to make it work:

```
import urllib2
page = urllib2.urlopen("https://www.plugincars.com/cars")

from BeautifulSoup import BeautifulSoup

soup = BeautifulSoup(page)
```

Special note for Anaconda users: The Spyder application available through Anaconda Navigator is actually a Python IDE (Integrated Development Environment). Yay! And Beautiful Soup is already available. So any Python code that I provide can be pasted into the Spyder editor (left side of the IDE) and run from there (either line by line or in its entirety using the appropriate icon, though line by line doesn't work so well with FOR loops), with results showing up in the console on the right side of the IDE. The console will show you where the output file has been written – the code provides a file name, but what folder has that file ended up in? It will probably be in a folder named something like C:\Users\<onyen> or C:\Users\<onyen>\.spyder-py3 or else in the folder from which you opened scrape-plug-in-hybrid-info.py .

Summary

You might have noticed that these four ways of web scraping have different strengths, meaning that it's nice if you know more than one possibility when confronted with a new web scraping task.

This summary conceptually divides web scraping between reading tables, which already have a rectangular structure, vs. pulling individual pieces of data from a web page using parsing techniques (imposing structure based on a pattern: starting with finding a token or signal of a nearby piece of data, then locating that piece of data relative to the signal).

Using the techniques presented here (which admittedly are only the tip of the iceberg), it appears that

1. R is useful for reading tables of data.
2. JMP is another handy way to read HTML tables.
3. On the parsing front, SAS is particularly useful when data to be scraped appears on the same line of HTML code as its signal since that simplifies the parsing task. In this case, you can use familiar SAS DATA step tools to do the parsing work. However, the SAS DATA step and INPUT statement are very flexible, so having everything on one line – both the signal and the related data - is not an absolute limitation. Regular expressions could help you to do more complex parsing.
4. Also on the parsing front, Python allows you to key on the structure of a file, allowing you to succeed even when the piece of data of interest is not on the same line as the signal for that piece of data. For better or worse, you'll need to learn something about a new language to succeed with this approach.