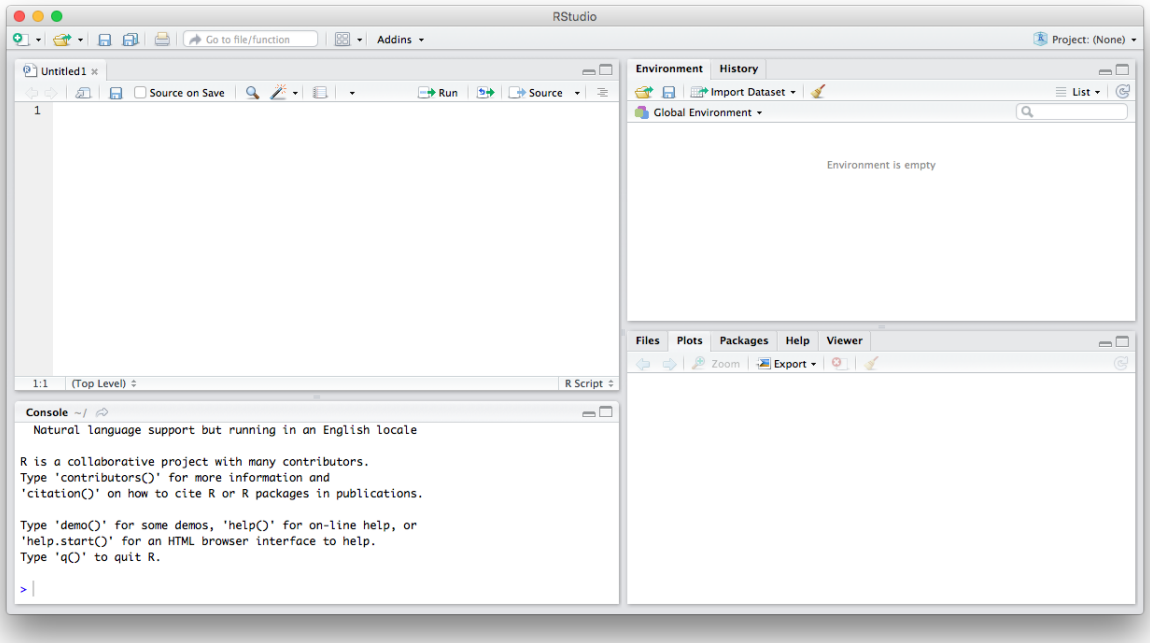
Scraping HTML Tables Tutorial

Sports-Reference and its sub-sites make it very easy to take data from their site. You simply go to the table you want, click "Share & More", and then click "Download as CSV (for Excel)". However, very few fun data science problems can be solved with only one data table. What happens if you need an entire season's worth of box scores ? Two seasons ? Ten ? Nobody is going to sit there and manually download each one. This tutorial is to show you step-by-step how to get a load of data at once, with minimal programming (and no experience needed) !

1. Download Required Software

There are only two things required for this. Both are free ! The first is R, a programming language devoted to statistics. You should be able to search "Download R" in your search engine and find the installation file that corresponds to your machine.

The second thing you need to download is R Studio, which is just an easy way to see your code. It is especially great for beginners, but I use it too. Again, acquiring it should be as easy as a Google search. Once you have everything installed, and you open R Studio, it should look something like this :



The top left box is where we will primarily be working. It is where we write our code so that we can save it for later, just like a Word document. The bottom left box is called the console, and

that is where our code is actually executed. If we type a command there, it will run. If we type it in the top-left, it will just sit there, until we hit the run button in the bar across the top.

The top right box shows our variables. Variables in code are just as they are in mathematics; they're names that represent something else, like a number, a vector, or a data frame. Naturally, we have none yet, but it will begin to fill up. The bottom right corner shows any graphics we create or documents if we ask for help about a certain command. We probably will not use it.

2. Install Packages and Setup

There are certain commands that simply come with R. For example, if we were to type max(c(1,2,3,4)), the result would be 4. (c(...) is the syntax for writing a vector, or a list, in R). However, certain commands require that we first install the package, or library, in which they are contained. There are three packages we need, and two steps we need to take in order to be able to use the functions inside of them. (To be totally honest, we may even be able to accomplish our goal without three packages, but I'm going to explain to you the way I am comfortable with. If any of you have easier ways, please feel free to share!) The following are the names of the packages :

rvest
XML
RCurl

Installing the packages is a one-time deal. If you are doing a second project a year from now where you need to scrape more data tables, you do not need to do this step again (unless you have changed computers). Because of this, we will type commands in the console, since we do not need to save the command for next time. The syntax is as follows : install.packages("<name of package>"). So, basically, we will type the following three lines, hitting Enter after each one :

```
install.packages("rvest")
install.packages("XML")
install.packages("RCurl")
```

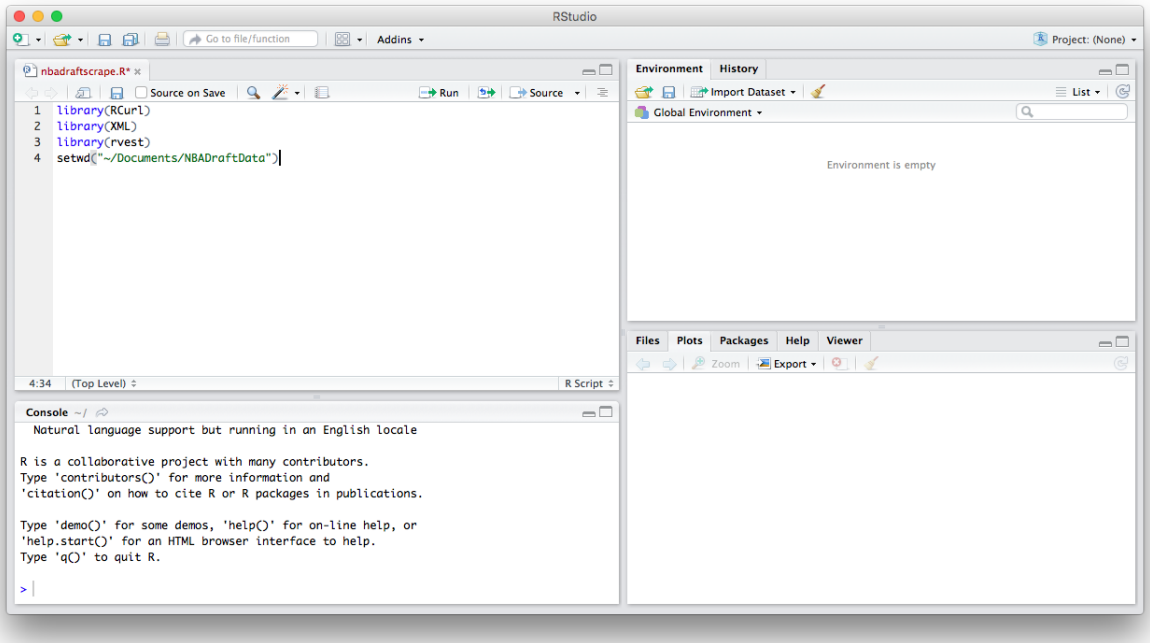
Now we have reached the part where we should start typing in the upper left box, in case we want to reuse our code for later. The library() command tells the software which libraries we are going to need for our code, so it's good to take care of that first. It's very similar to the previous syntax :

```
library(rvest)
library(XML)
library(RCurl)
```

There is one last thing for setup that isn't required, but it is a good habit, and it makes it easy to keep track of things. We our going to set our working directory. That means we are going to pick a folder where all of our tables and our code are going to be stored. I made a folder within my Documents folder called "NBADraftData" to store mine, but you can name yours anything you want and put it anywhere you want, as long as you can put the exact directory (path of folders) in the command. As you can see below, the syntax is :

```
setwd(" /Documents/NBADraftData")
```

Except you, of course, would put your own directory and names within the quotes. Each folder should be separated by a /, and the folders get more and more specific from left to right. The just represents the home directory, which is the default. Finally, I am going to save my file as "nbadraftscrape.R", and it will be saved in that folder.

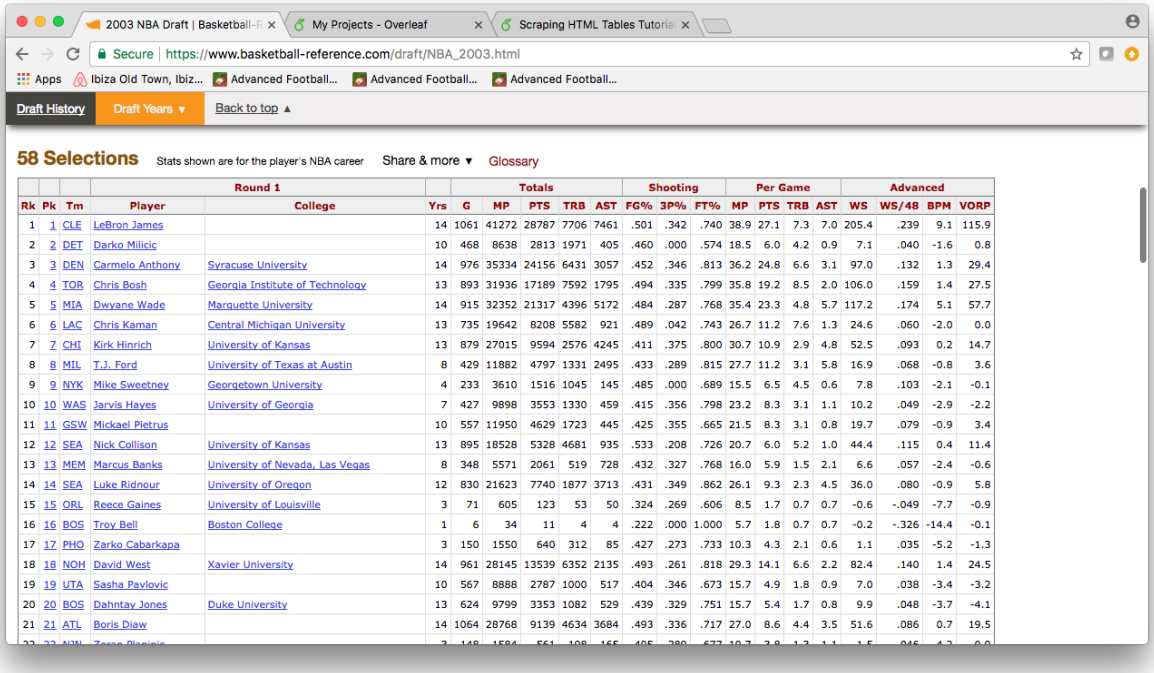


3. Getting URLs

This is probably the hardest part of this entire tutorial. Our code has to know where the tables are coming from, so it has to know all the URL's. There are two different situations we can have. Either the URL's can be generated by a simple sequence function, or we first have to scrape the URL's in order to scrape the data. It depends on the complexity of the problem.

URL Generation Through Sequences

If you navigate to the professional basketball side of sports-reference, then to "Draft", then click some random year (2003 is shown below), and finally scroll down a bit, you'll see something like this.



			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	CLE	LeBron James		14	1061	41272	28787	7706	7461	.501	.342	.740	38.9	27.1	7.3	7.0	205.4	.239	9.1	115.9
2	2	DET	Darko Milicic		10	468	8638	2813	1971	405	.460	.000	.574	18.5	6.0	4.2	0.9	7.1	.040	-1.6	0.8
3	3	DEN	Carmelo Anthony	Syracuse University	14	976	35334	24156	6431	3057	.452	.346	.813	36.2	24.8	6.6	3.1	97.0	.132	1.3	29.4
4	4	TOR	Chris Bosh	Georgia Institute of Technology	13	893	31936	17189	7592	1795	.494	.335	.799	35.8	19.2	8.5	2.0	106.0	.159	1.4	27.5
5	5	MIA	Dwyane Wade	Marquette University	14	915	32352	21317	4396	5172	.484	.287	.768	35.4	23.3	4.8	5.7	117.2	.174	5.1	57.7
6	6	LAC	Chris Kaman	Central Michigan University	13	735	19642	8208	5582	921	.489	.042	.743	26.7	11.2	7.6	1.3	24.6	.060	-2.0	0.0
7	7	CHI	Kirk Hinrich	University of Kansas	13	879	27015	9594	2576	4245	.411	.375	.800	30.7	10.9	2.9	4.8	52.5	.093	0.2	14.7
8	8	MIL	T.J. Ford	University of Texas at Austin	8	429	11882	4797	1331	2495	.433	.289	.815	27.7	11.2	3.1	5.8	16.9	.068	-0.8	3.6
9	9	NYK	Mike Sweetney	Georgetown University	4	233	3610	1516	1045	145	.485	.000	.689	15.5	6.5	4.5	0.6	7.8	.103	-2.1	-0.1
10	10	WAS	Jarvis Hayes	University of Georgia	7	427	9898	3553	1330	459	.415	.356	.798	23.2	8.3	3.1	1.1	10.2	.049	-2.9	-2.2
11	11	GSW	Mickael Pietrus		10	557	11950	4629	1723	445	.425	.355	.665	21.5	8.3	3.1	0.8	19.7	.079	-0.9	3.4
12	12	SEA	Nick Collison	University of Kansas	13	895	18528	5328	4681	935	.533	.208	.726	20.7	6.0	5.2	1.0	44.4	.115	0.4	11.4
13	13	MEM	Marcus Banks	University of Nevada, Las Vegas	8	348	5571	2061	519	728	.432	.327	.768	16.0	5.9	1.5	2.1	6.6	.057	-2.4	-0.6
14	14	SEA	Luke Ridnour	University of Oregon	12	830	21623	7740	1877	3713	.431	.349	.862	26.1	9.3	2.3	4.5	36.0	.080	-0.9	5.8
15	15	ORL	Reece Gaines	University of Louisville	3	71	605	123	53	50	.324	.269	.606	8.5	1.7	0.7	0.7	-0.6	-.049	-7.7	-0.9
16	16	BOS	Troy Bell	Boston College	1	6	34	11	4	4	.222	.000	1.000	5.7	1.8	0.7	0.7	-0.2	-.326	-14.4	-0.1
17	17	PHO	Zarko Cabarkapa		3	150	1550	640	312	85	.427	.273	.733	10.3	4.3	2.1	0.6	1.1	.035	-5.2	-1.3
18	18	NOH	David West	Xavier University	14	961	28145	13539	6352	2135	.493	.261	.818	29.3	14.1	6.6	2.2	82.4	.140	1.4	24.5
19	19	UTA	Sasha Pavlovic		10	567	8888	2787	1000	517	.404	.346	.673	15.7	4.9	1.8	0.9	7.0	.038	-3.4	-3.2
20	20	BOS	Dahntay Jones	Duke University	13	624	9799	3353	1082	529	.439	.329	.751	15.7	5.4	1.7	0.8	9.9	.048	-3.7	-4.1
21	21	ATL	Boris Diaw		14	1064	28768	9139	4634	3684	.493	.336	.717	27.0	8.6	4.4	3.5	51.6	.086	0.7	19.5

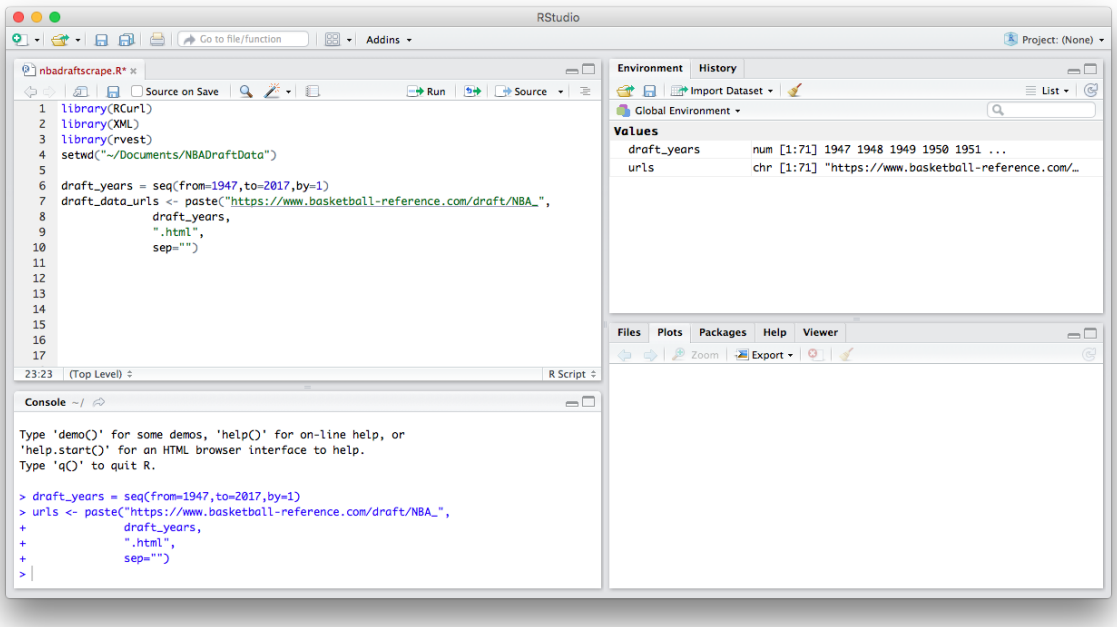
The important part of this image is the URL. Every draft has the same URL structure, and the only thing that changes is year toward the end. Therefore, all we have to do is make a list (or rather, vector) of URLs where the date is changed.

One very easy way to do this is an R command called seq(). You simply give it the beginning (*from*) and end (*to*) of the sequence you want to create, as well as the *step* (e.g., the step would be 2 if you wanted every year.) If your step size is one, you do not need to include that part because that is the default. These three things are the three arguments, or things that go inside the parentheses.

seq(from=1947,to=2017,by=1), OR

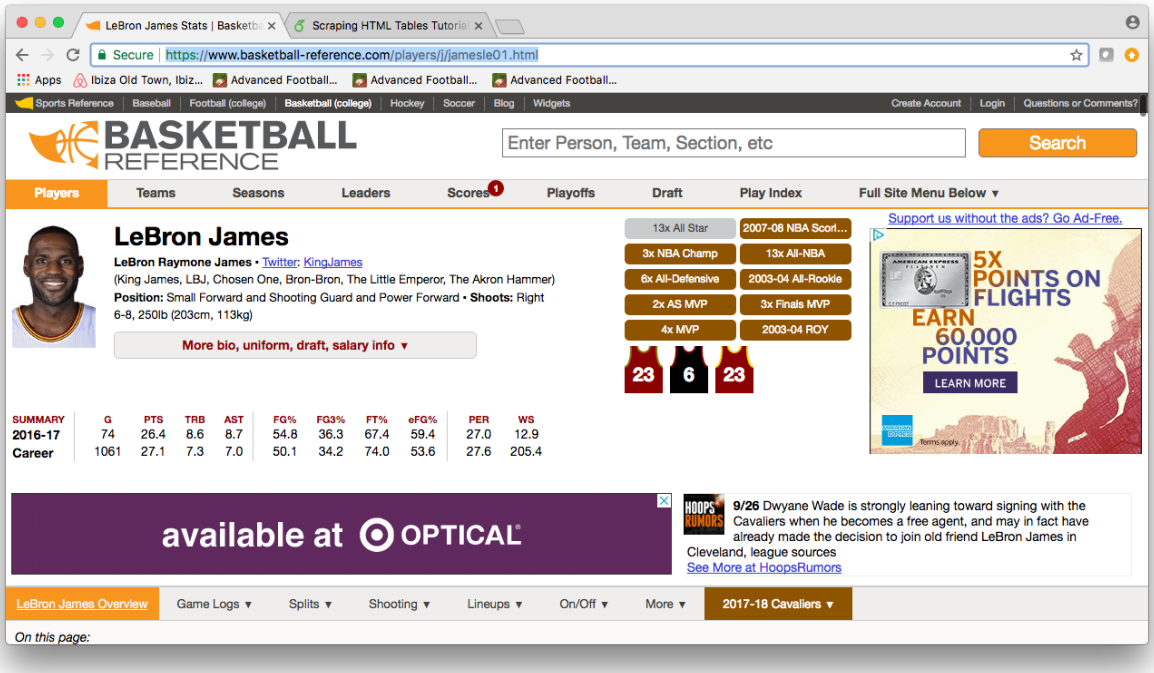
seq(1947,2017) because this is the default order, and the default step is already one.

I'll save the result of seq() in an object called draft_years. It is a list of the years where the draft took place. Then, I'll want to combine it with the rest of the URL using a function called paste() which simply combines strings. paste() takes any number of strings that you want to combine and separates them with whatever you want - the default is a space. Since we want there to be no separation at all, we will make sure to add sep="" inside the parentheses, after all of the things we want to paste together. The code is shown below.



URL Generation Through Scraping

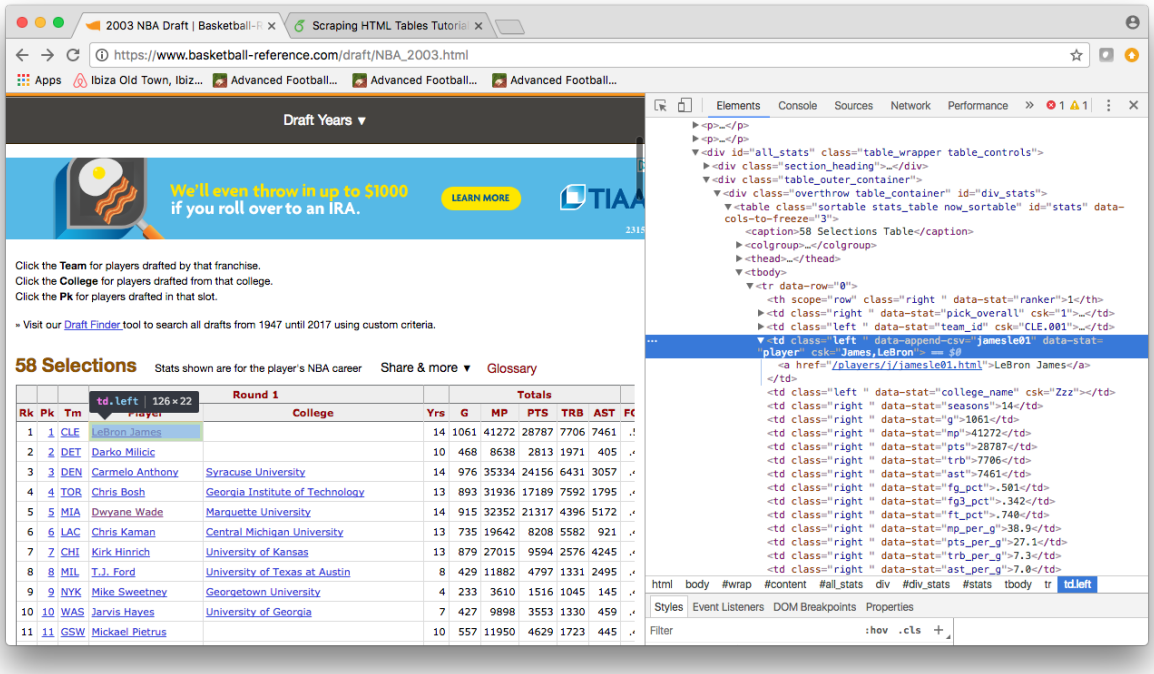
What happens if I'm not interested in this table, but I'm interested in what it links to ? Maybe I want the statistics of all the players that were drafted in 2003, instead. Below you can see the statistics page for Lebron James.



Unfortunately, making changes to this URL is not as intuitive. It seems that each player has his

own code name, where Lebron's is jamesle01. In this case, the best thing to do would be to scrape the URL's first from the draft table, then using those URL's, scrape the individual player data tables. As you can see, you can link to any of the drafted players' pages by clicking their names. Therefore, all of their URL's have to be stored on that page somewhere. You can find them with the following steps :

Right click somewhere on the page, preferably on a row of the data.
Select "Inspect." This is the HTML for the current webpage.
Find the link you're looking for in the HTML. You may have to hit the arrows to the left of the code to show all of the HTML. Below, you can see where Lebron's link is.



As you can see, the URL itself is several layers deep into the HTML. Starting at the part that begins with "<table ...", where the ellipsis ... acts as a replacement for the remainder of the attributes between the <>. the layers are as follows :

<table ...>. This is the main table on the page.
<tbody> This is the table body - after headers, and such.
<tr data-row="0"> This is the first table row. It is very common to start counting at 0 instead of 1.
<td ... data-stat="player"...> td stands for table data, or the column. There are a few other <td>s before the one for which data-stat="player", and these merely represent the couple of columns that occur before the player's name, but we aren't concerned about those.
<a href="/players/j/jamesle01.html"> Here is where the URL we need is located. If you append this to "www.basketball-reference.com" then you will reach Lebron's page.

This structure is the same for every player in the table. We will come back to it. First, we have to get the HTML and save it. We can take the data directly from the site, but it is good practice to save it as a .html on your computer. The function to do so is getURL(), and you put your URL (with quotes) in the parentheses. You also must save it as some object - mine is called mydata.

mydata <- getURL("your_url_goes_here")

Then, we will save it. This is done using the writeLines() function. The two arguments that go in the parentheses are your data object (mydata) and the name of the file that you want it to be saved as. Remember which directory you are working in - that is where it will be saved, unless you specify in the file name a subdirectory. It is also important to close files after you are done with them.

```
writeLines(mydata,"NBA_Draft_03.html")
close(file("NBA_Draft_03.html"))
```

The next step is to parse the file and store it in some variable; I called mine mydoc. Parsing basically analyzes the HTML to determine its structure, similarly to what we did above manually to determine where the URL was.

```
mydoc <- htmlParse("NBA_Draft_03.html")
```

Finally we have reached the part where we must use the information above to create the XML Path to our URL. An XML path is very similar to your file path, where folders are separated by slashes. Instead, the different "layers" as I called them (technically HTML commands) are separated by slashes.

We started with the table command, but that wasn't the start of the document. There were several layers before that. However, we don't have to manually write out them all. If we write a double slash, "//", it indicates that any sort of commands can come before it. Thus, the start of our path will be "//table[@id='stats']". There are a couple of tables, so we have to clarify the unique id of the table we want. This is the syntax for it. The next part was the body, then the row and several different columns. We want the data from every row, so the rows and the first couple columns don't need to be specified. Therefore we can rope them into another "//" until we reach what we really care about - the column for "player". Thus, we so far have "//table[@id='stats']/tbody//td[@data-stat='player']". Finally, we must tell XML that we are looking for looking for the "a" command that follows it. Thus, our final path is :

```
"//table[@id='stats']/tbody//td[@data-stat='player']/a"
```

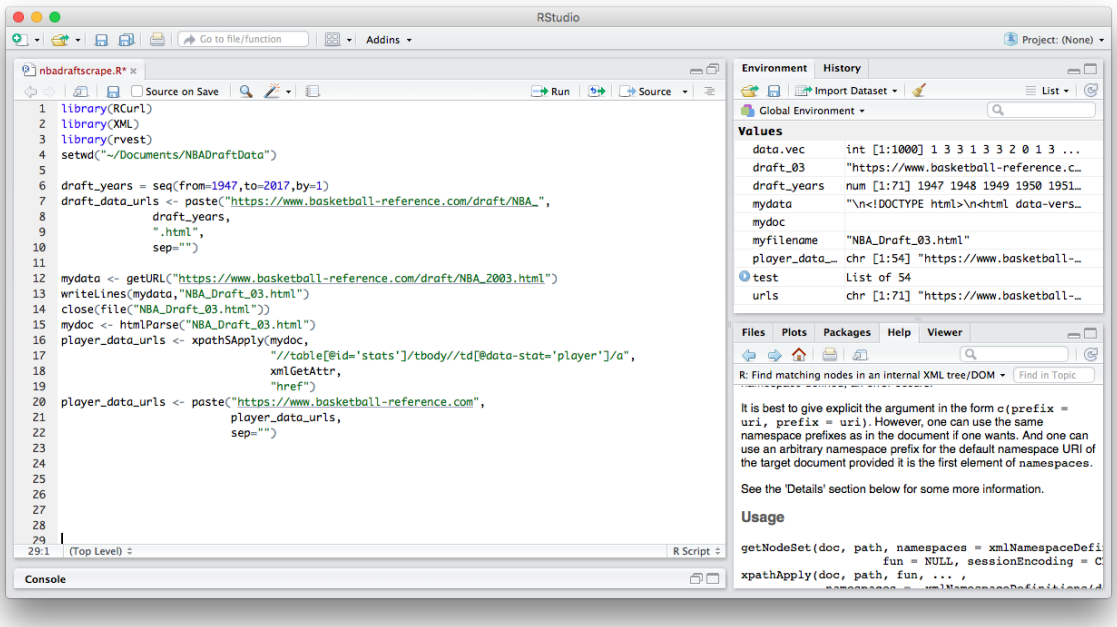
This will serve as one of our arguments to the xpathSApply function. The first argument will be our parsed document, then the path. Then, we must tell it what we want to extract. In our case, we want the href attribute. An attribute is what modifies, or makes specifications in, a command. Thus it appears within the <> after the command itself. For example, the href attribute specifies that "/players/j/jamesle01.html" is the URL that this section of the document must link to. In short, the third argument is xmlGetAttr. Finally, we tell it which attribute - href.

```
player_data_urls <- xpathSApply(mydoc, "//table[@id='stats']/tbody//td[@data-stat='player']/a",
 xmlGetAttr, "href")
```

Running this should get those URL segments. However, we still need to append them to our root URL, "www.basketball-reference.com". We can come back to our paste function for that.

```
paste("https ://www.basketball-reference.com",player_data_urls,sep="")
```

The final product is shown below.

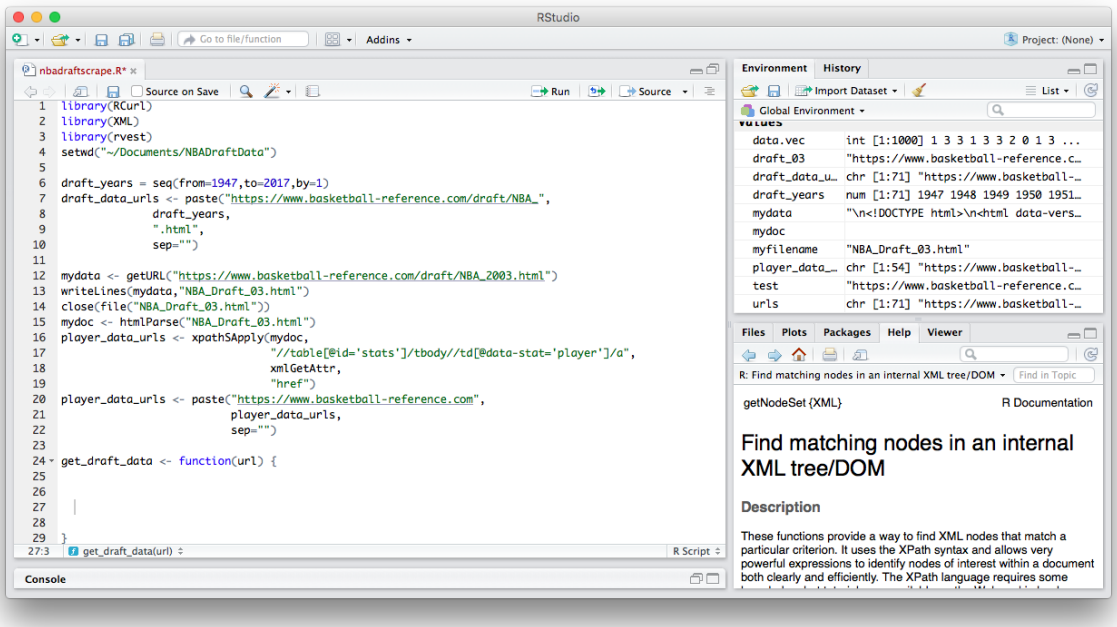


4. Scraping HTML Tables

For this portion, I'm going to use the draft URLs rather than the player statistics URLs.

Again, the first thing we must do is get the data from the link and save it as an HTML file. However, we must go through 71 years of draft data, so 71 URLs. That'd be extremely tedious to do by hand, so we can create our own function that takes URL as an argument, and at the end, we can simply apply that function to the entire URL vector. For now, however, let's write a function that will work for one URL.

The syntax for a function is just like naming a variable and can be seen below. The code that we want our function to execute will be placed inside the braces.

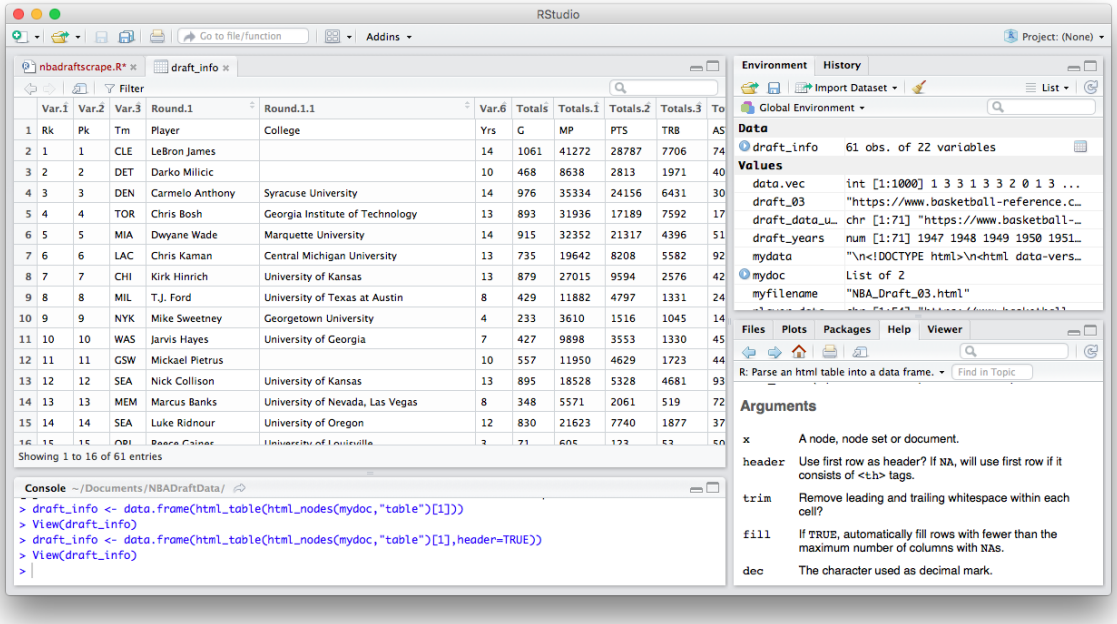


First, we use getURL() to get the HTML code from the website. Then, we want to save it to a file. However, we will have to write it in such a way that each file name is unique for when we apply the function the entire set of URLs. We will use the year because that is what differentiates each one. Therefore, we will have to extract the year from the URL string. This can be done with the strsplit() function, which splits a string into a list of parts at a given character. As a reminder, our URL is "https ://www.basketball-reference.com/draft/NBA_2003.html", so we want the last part of that.

mydata <- getURL(https ://www.basketball-reference.com/draft/NBA_2003.html)
myfilename <- unlist(strsplit(url,"/"))[5]. This picks the fifth element of the list, which is NBA_2003.html (or whatever year we're on).
writeLines(mydata,myfilename)
close(file(myfilename))

Instead of the parsing function we used before, we will use read_html() which does essentially the same thing. We save that in a variable called mydoc. Then, we will use a function called html_nodes() that says we want to extract from mydoc only the "table" nodes. This returns a list of all the tables on the page - there is only one on this page, so we want tables[1], and we would like to convert it into a data frame, which is the easiest thing to work with in R.

mydoc <- read_html(myfilename)
tables <- html_nodes(mydoc,"table")
draft_info <- data.frame(html_table(tables[1])) Below, we display the resultant draft_info!



5. Fixing Up the Data

It looks pretty good, except for the first row should actually be the headers. Your code likely will have to deal with other formatting issues, but I'll show you how I took care of those. We also have other issues about formatting, but we will take care of them another time.

colnames(draft_info) <- draft_info[1,] This is assigning the column names of draft_info to be the values in row one.

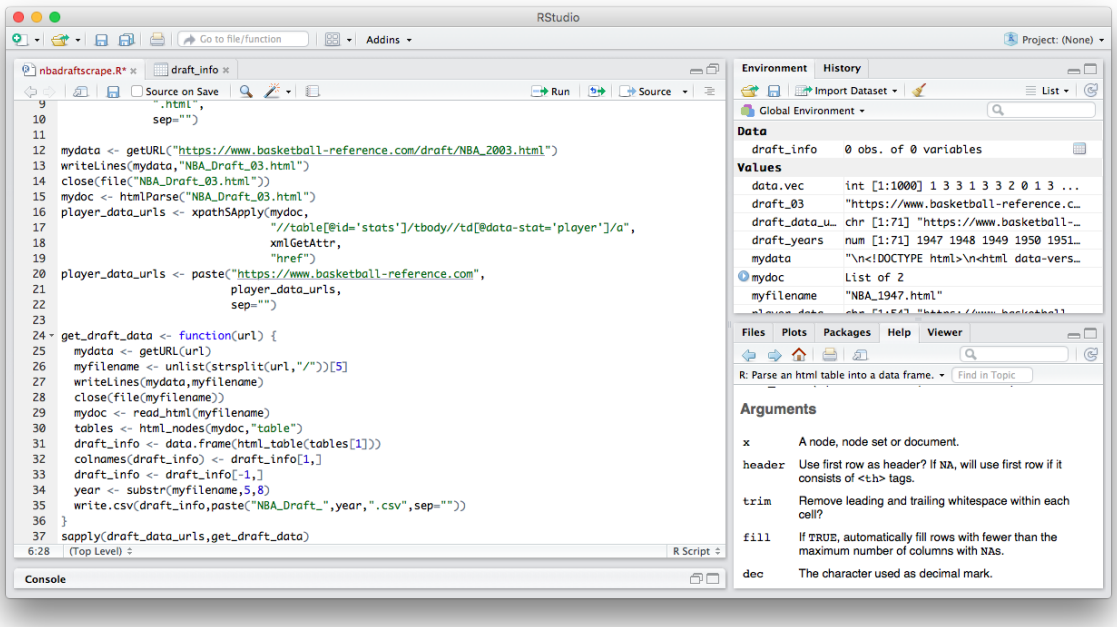
draft_info <- draft_info[-1,] This will get rid of the first row.

year <- substr(myfilename,5,8) This is a substring function that takes a portion of the string from position 5 to position 8.

6. Save As CSV

It's important to save the data frame each time we make it, so that will be the last thing we do in the function. There is a command called write.csv() which will take the first argument and save it under a file name given in the second argument.

Finally, using sapply, we will apply the function get_draft_data to every url in draft_data_urls.


RStudio
nbadraftscrape.R*
draft_info
Source on Save
Run
Source
".html",
sep="")
mydata <- getURL("https://www.basketball-reference.com/draft/NBA_2003.html")
writeLines(mydata,"NBA_Draft_03.html")
close(file("NBA_Draft_03.html"))
mydoc <- htmlParse("NBA_Draft_03.html")
player_data_urls <- xpathSApply(mydoc,
"//table[@id='stats']/tbody//td[@data-stat='player']/a",
xmlGetAttr,
"href")
player_data_urls <- paste("https://www.basketball-reference.com",
player_data_urls,
sep="")
get_draft_data <- function(url) {
mydata <- getURL(url)
myfilename <- unlist(strsplit(url,"/"))[5]
writeLines(mydata,myfilename)
close(file(myfilename))
mydoc <- read_html(myfilename)
tables <- html_nodes(mydoc,"table")
draft_info <- data.frame(html_table(tables[1]))
colnames(draft_info) <- draft_info[1,]
draft_info <- draft_info[-1,]
year <- substr(myfilename,5,8)
write.csv(draft_info,paste("NBA_Draft_",year,".csv",sep=""))
}
sapply(draft_data_urls,get_draft_data)
6:28
(Top Level)
R Script
Console
Environment
History
Import Dataset
List
Global Environment
Data
draft_info
0 obs. of 0 variables
Values
data.vec
int [1:1000] 1 3 3 1 3 3 2 0 1 3 ...
draft_03
"https://www.basketball-reference.c...
draft_data_u...
chr [1:71] "https://www.basketball-...
draft_years
num [1:71] 1947 1948 1949 1950 1951...
mydata
"\n<!DOCTYPE html>\n<html data-vers...
mydoc
List of 2
myfilename
"NBA_1947.html"
Files
Plots
Packages
Help
Viewer
R: Parse an html table into a data frame.
Find in Topic
Arguments
x
A node, node set or document.
header
Use first row as header? If NA, will use first row if it consists of <th> tags.
trim
Remove leading and trailing whitespace within each cell?
fill
If TRUE, automatically fill rows with fewer than the maximum number of columns with NAs.
dec
The character used as decimal mark.