An Introduction to Using APIs to Obtain Data

API stands for Application Programming Interface. As with everything else, you can get a comprehensive look at APIs at Wikipedia (https://en.wikipedia.org/wiki/Application_programming_interface). For purposes of this course, what I say should be good enough!

Basically, APIs enable software packages or applications to communicate with each other. One piece of software says, if you provide me with these pieces of information about who you are and what you want, I'll pass the information that you want back to you. This can help a person weave different software applications together rather than trying to rely on any one piece of software to meet all needs.

For purposes of BIOS 669, we will simply think of and use APIs as another way of obtaining data – we will provide the pieces of information needed on particular websites to ask them to return data of interest to us. Also, we will only use relatively simple APIs that do not require accounts and special tokens being passed back and forth.

The notes and videos will provide some examples of using APIs to obtain data, and the related assignment will have you use APIs to do this independently. Each example will also include converting the returned data to useful SAS data sets.

APIs use structured data

APIs that serve up data do so by providing data in a structured fashion. For a while, XML was the most common data structure, but these days the JSON file structure seems to be preferred. Both XML and JSON involve nested lists of name – value pairs.

Here is a very simple XML file (looks like structured HTML, doesn't it?):

```
<CATALOG>
        <BOOK>
                <TITLE>Harry Potter and the Philosopher's Stone</TITLE>
                <AUTHOR>J. K. Rowling</AUTHOR>
                <YEAR>1997</YEAR>
        </BOOK>
        <BOOK>
                <TITLE>Harry Potter and the Chamber of Secrets</TITLE>
                <AUTHOR>J. K. Rowling</AUTHOR>
                <YEAR>1998</YEAR>
        </BOOK>
</CATALOG>
```

And here is the same data in a JSON file:

```
{
   "catalog": {
        "book":
                {
                        "title": "Harry Potter and the Philosopher’s Stone",
                        "author": "J. K. Rowling",
                         "year": "1997"
                        },
            "book":
                         {
                        "title": "Harry Potter and the Chamber of Secrets",
                        "author": "J. K. Rowling",
                        "year": "1998"
                        }
             }
}
```

Another way to structure this data in a JSON file would be

```
{
   "author": "J. K. Rowling",
            "books": [
                         {
                         "title": "Harry Potter and the Philosopher’s Stone",
                        "year": "1997"
                        },
                        {
                        "title": "Harry Potter and the Chamber of Secrets",
                         "year": "1998"
                         }
                          ]
}
```

Extending the above, another JSON structure would be

```
{ "catalog": [
   {
      "author": "J. K. Rowling",
             "books": [
                        {
                        "title": "Harry Potter and the Philosopher’s Stone",
                        "year": "1997"
                        },
                        {
                        "title": "Harry Potter and the Chamber of Secrets",
                        "year": "1998"
                        }
                        ]
   },
   {
      "author": "Suzanne Collins",
            "books": [
                        {
                        "title": "The Hunger Games",
                        "year": "2008"
                        },
                         {
                        "title": "Catching Fire",
                        "year": "2009"
                        },
                        {
                        "title": "Mockingjay",
                        "year": "2010"
                        }
                        ]
   }
             ]
}
```

Here is part of what a weather JSON file might look like (for the current conditions at a specific site). In fact, this is a slight variation of what I am seeing at the openweathermap API for Carrboro, NC at 11:40 AM on 2/11/2019.

```
{
"weather": [
  {
   "id": 4459343,
   "name": "Carrboro",
   "coord": {
    "lat": 35.9101,
    "lon": -79.0753
   },
   "main": {
    "temp": 40.24,
    "pressure": 1022,
    "humidity": 88,
    "temp_min": 37.4,
    "temp_max": 42.98
   },
   "dt": 1549901220,
   "wind": {
    "speed": 3.87,
    "deg": 66.0114
   },
   "sys": {
    "country": "US"
   },
   "rain": {
    "1h": 1.4
```

```
      },
      "snow": null,
      "clouds": {
        "all": 90
      }
  }
        ]
}
```

You can probably imagine that it would be possible to convert such structured lists into the rectangular form of a SAS data set (or an R data frame), but it might not be so easy to write that conversion code yourself especially since file structures can vary. Fortunately, SAS and other languages provide helpful tools. In this course we will focus on SAS tools for making data sets from JSON files.

The SAS tools we will use are PROC HTTP, the JSON libname engine, and JSON maps – and the SAS DATA step and PROC SQL as well, of course.

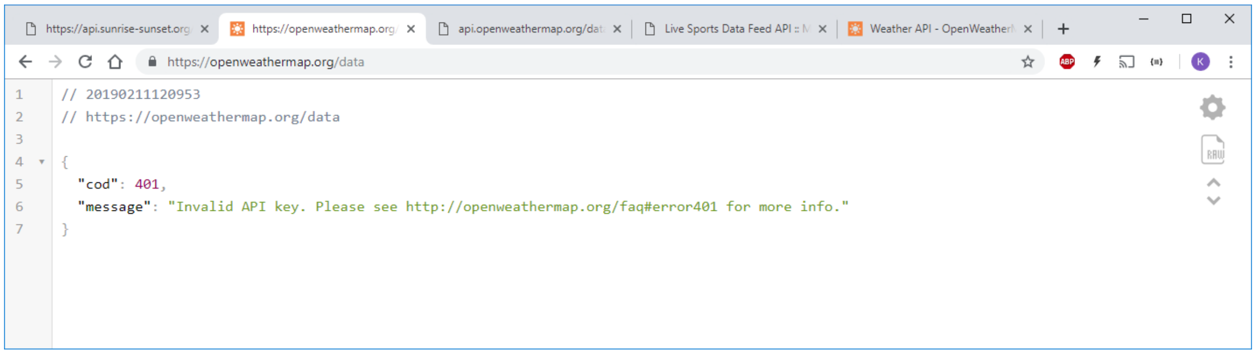
APIs often require keys and sometimes payment

As you can imagine, data APIs require time and effort to build, compile, and maintain. Some people do this work as a labor of love about a certain subject and make their API freely open to all, but more and more APIs now require API users to register for an access key and possibly even pay a subscription fee. The reasoning is that, as you will see, getting data from an API enables you to do something with that data, and maybe you are even using that data in a website for which you are charging! So it makes sense that the data compiler would be charging for use.

And sometimes sites have levels of access. For example, check out https://www.mysportsfeeds.com/. An individual fan can sign up for free access, but you'll see a button "For Sites and Apps" that provides non-commercial and commercial access for different levels of fees. Presumably, commercial sites might be pulling data from the mysportsfeed API to build their own websites or support betting operations.

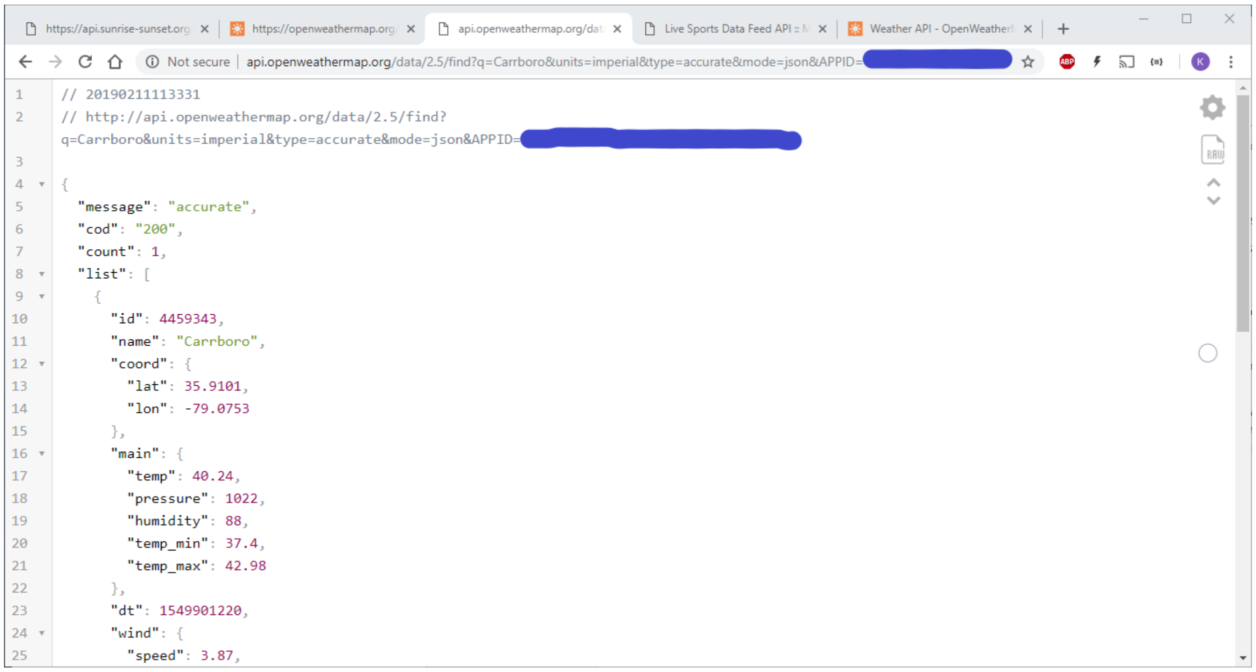
For this BIOS 669 introduction to APIs, we won't be using any APIs that require fees or even access keys. But let me show you how an access key works, using http://openweathermap.org as an example.

For a long time, the openweathermap API was entirely open. Now, you need to sign up for a free account to have access to current weather or 5 day forecasts. If you try to access data without supplying a key, you see this:



```
// 20190211120953
// https://openweathermap.org/data

{
  "cod": 401,
  "message": "Invalid API key. Please see http://openweathermap.org/faq#error401 for more info."
}
```

I do have a key (I requested it once and now can re-use it whenever I want), and when I supply that key (masked below) along with some other needed information about WHAT I want to see, I get this:



api.openweathermap.org/data/2.5/find?q=Carrboro&units=imperial&type=accurate&mode=json&APPID=[redacted]

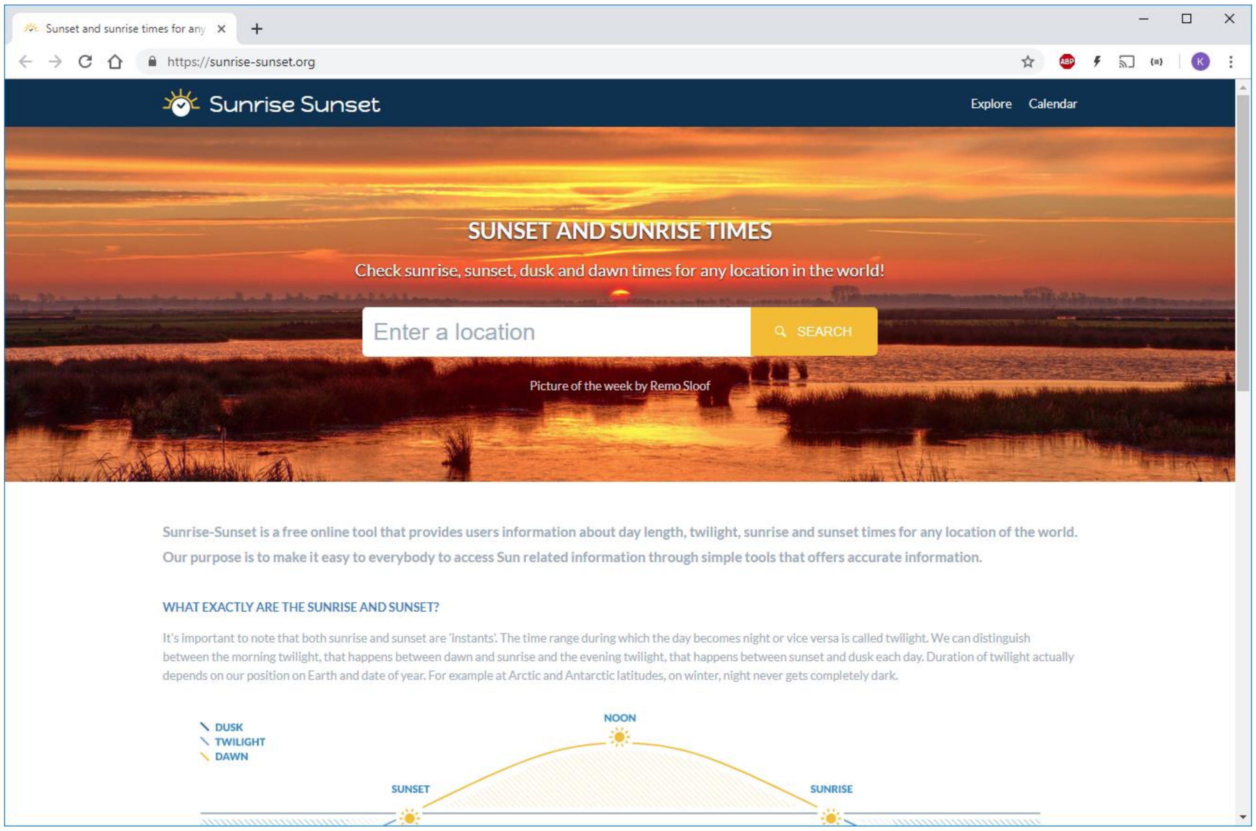
```
// 20190211113331
// http://api.openweathermap.org/data/2.5/find?
q=Carrboro&units=imperial&type=accurate&mode=json&APPID=[redacted]

{
  "message": "accurate",
  "cod": "200",
  "count": 1,
  "list": [
    {
      "id": 4459343,
      "name": "Carrboro",
      "coord": {
        "lat": 35.9101,
        "lon": -79.0753
      },
      "main": {
        "temp": 40.24,
        "pressure": 1022,
        "humidity": 88,
        "temp_min": 37.4,
        "temp_max": 42.98
      },
      "dt": 1549901220,
      "wind": {
        "speed": 3.87,
```

Note that some APIs require multiple keys / tokens / what are called "secrets" for access to data. Such authentication requirements are related to data security, of course. For example, the Twitter API requires you to obtain and use multiple keys / tokens for successful access to even your own account as data.

PROC HTTP

The SAS System's PROC HTTP "issues Hypertext Transfer Protocol (HTTP) requests". What does this mean? It means that you can use PROC HTTP to request data from a web server. Just as you enter a URL in a web browser for the web page you want to see, you supply PROC HTTP with a URL of the data you are interested in. And just as a web page URL often includes parameters to provide details about what you are interested in, usually the URL passed to PROC HTTP will include parameters.

For example, http://sunrise-sunset.org allows us to enter a location to find out today's sunrise and sunset times for that place.



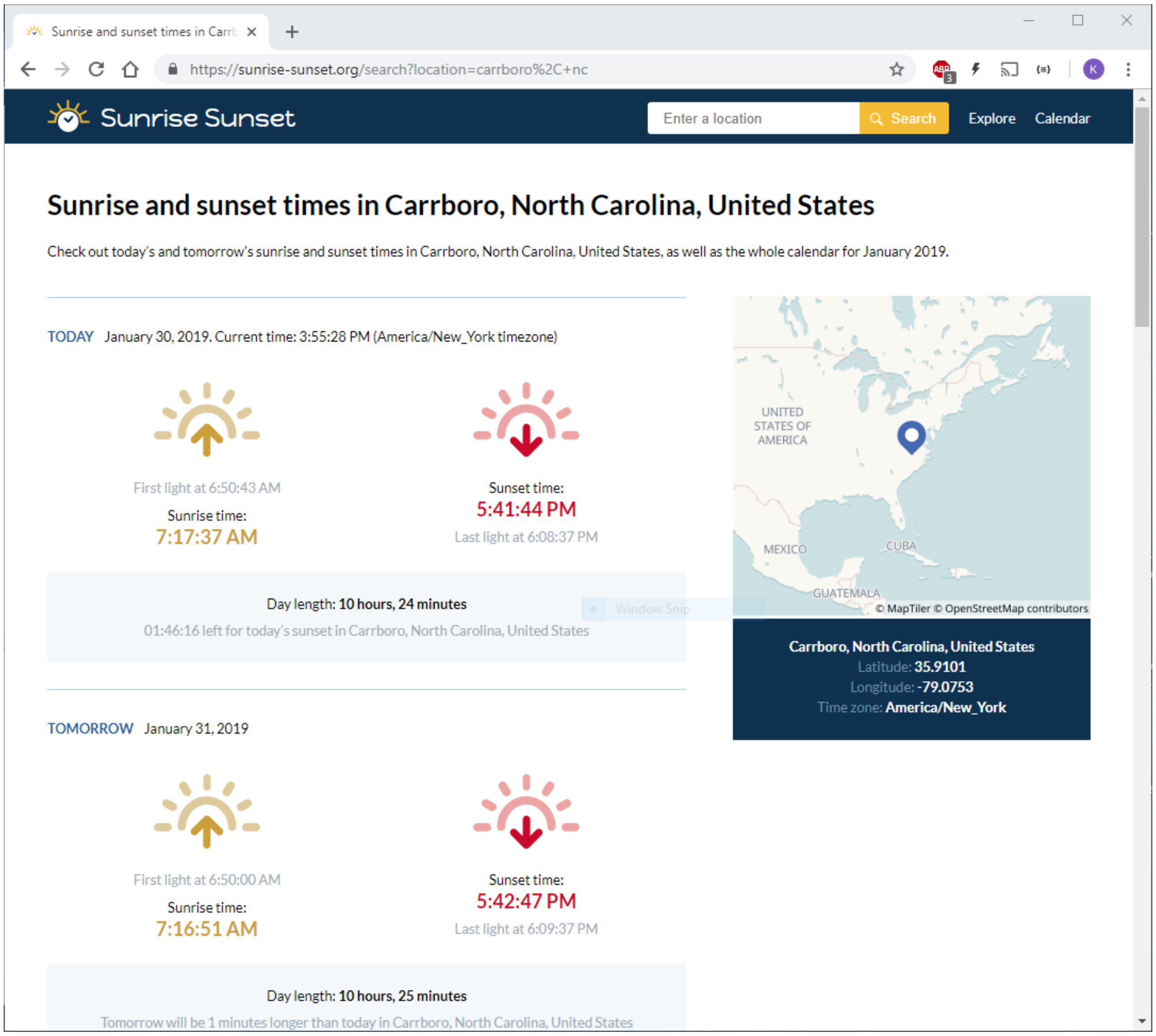
Sunset and sunrise times for any
https://sunrise-sunset.org
Sunrise Sunset
Explore
Calendar
SUNSET AND SUNRISE TIMES
Check sunrise, sunset, dusk and dawn times for any location in the world!
Enter a location
SEARCH
Picture of the week by Remo Sloof
DUSK
TWILIGHT
DAWN
NOON
SUNSET
SUNRISE

Sunrise-Sunset is a free online tool that provides users information about day length, twilight, sunrise and sunset times for any location of the world. Our purpose is to make it easy to everybody to access Sun related information through simple tools that offers accurate information.

WHAT EXACTLY ARE THE SUNRISE AND SUNSET?

It's important to note that both sunrise and sunset are 'instants'. The time range during which the day becomes night or vice versa is called twilight. We can distinguish between the morning twilight, that happens between dawn and sunrise and the evening twilight, that happens between sunset and dusk each day. Duration of twilight actually depends on our position on Earth and date of year. For example at Arctic and Antarctic latitudes, on winter, night never gets completely dark.

Well, there is no data here, and even after I enter a location, I don't see structured data.


Sunrise and sunset times in Carr
https://sunrise-sunset.org/search?location=carrboro%2C+nc
Sunrise Sunset
Enter a location
Search
Explore
Calendar

Sunrise and sunset times in Carrboro, North Carolina, United States

Check out today's and tomorrow's sunrise and sunset times in Carrboro, North Carolina, United States, as well as the whole calendar for January 2019.

TODAY January 30, 2019. Current time: 3:55:28 PM (America/New_York timezone)

First light at 6:50:43 AM
Sunrise time:
7:17:37 AM

Sunset time:
5:41:44 PM
Last light at 6:08:37 PM

Day length: 10 hours, 24 minutes
01:46:16 left for today's sunset in Carrboro, North Carolina, United States

TOMORROW January 31, 2019

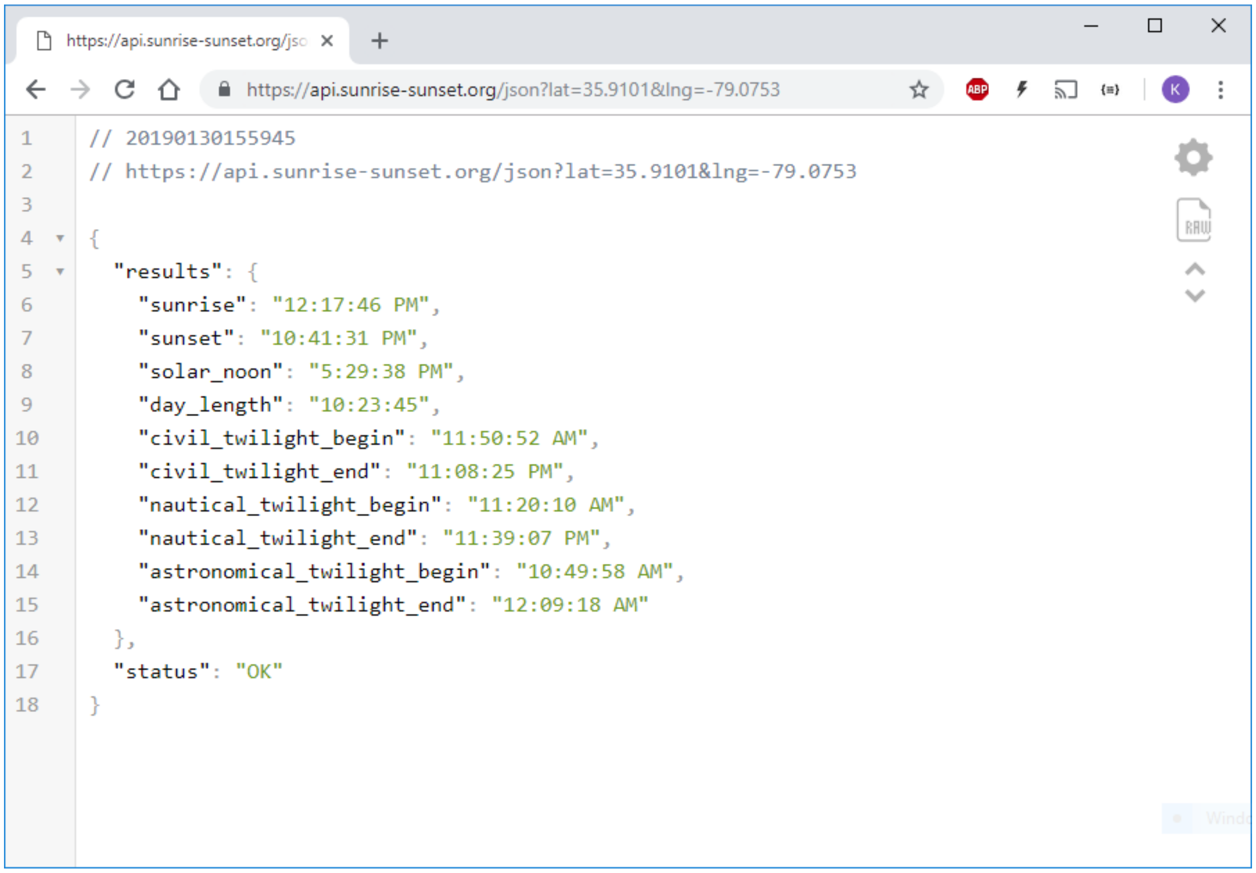
First light at 6:50:00 AM
Sunrise time:
7:16:51 AM

Sunset time:
5:42:47 PM
Last light at 6:09:37 PM

Day length: 10 hours, 25 minutes
Tomorrow will be 1 minutes longer than today in Carrboro, North Carolina, United States

Carrboro, North Carolina, United States
Latitude: 35.9101
Longitude: -79.0753
Time zone: America/New_York

But if I enter a variation of the URL above - appending api. before sunrise-sunset.org and passing information about what form of a file I want (json) and for what latitude and longitude – the URL becomes https://api.sunrise-sunset.org/json?lat=35.9101&lng=-79.0753 and I see


https://api.sunrise-sunset.org/json?lat=35.9101&lng=-79.0753

```
// 20190130155945
// https://api.sunrise-sunset.org/json?lat=35.9101&lng=-79.0753

{
  "results": {
    "sunrise": "12:17:46 PM",
    "sunset": "10:41:31 PM",
    "solar_noon": "5:29:38 PM",
    "day_length": "10:23:45",
    "civil_twilight_begin": "11:50:52 AM",
    "civil_twilight_end": "11:08:25 PM",
    "nautical_twilight_begin": "11:20:10 AM",
    "nautical_twilight_end": "11:39:07 PM",
    "astronomical_twilight_begin": "10:49:58 AM",
    "astronomical_twilight_end": "12:09:18 AM"
  },
  "status": "OK"
}
```

Just as my browser understands https://api.sunrise-sunset.org/json?lat=35.9101&lng=-79.0753 and shows me data, I can send that same URL string to PROC HTTP and have data returned. The syntax is

```
FILENAME carrbor1 TEMP;

PROC HTTP
     URL="https://api.sunrise-sunset.org/json?lat=35.9101&lng=-79.0753"
     METHOD="GET"
     OUT=carrbor1;
RUN;

LIBNAME sunrise JSON FILEREF=carrbor1;
```

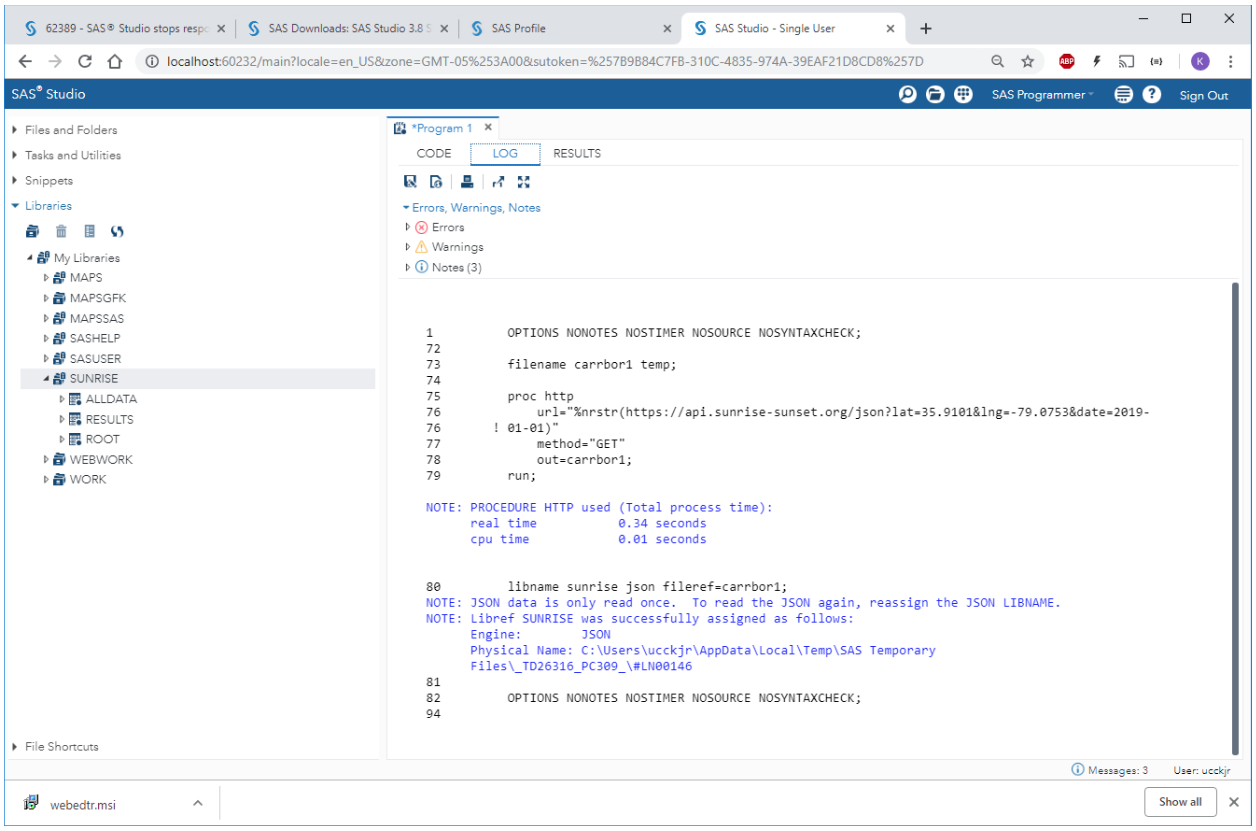
Looking more closely at this code:

- Method “GET” is the most common type of server request and signals that all information needed by the server is contained in the URL string. All of our requests in BIOS 669 will use GET.
- “carrbor1” in the code above is a file reference but not to any particular file or location. This fileref appears in three locations.
    - The initial FILENAME statement with the TEMP option asks SAS to set up a temporary file location with the fileref carrbor1.

  - Then, PROC HTTP uses that temporary file location (specified with OUT=carrbor1) to store the results of the request to the web server.
  - Finally, the LIBNAME statement asks SAS to interpret what is in the carrbor1 temporary file as a JSON file.
- In the LIBNAME statement, sunrise is our selected libref. SAS interprets a keyword appearing in a LIBNAME statement immediately after the libref as what's called a "libname engine". So here, the libname engine is JSON. That tells SAS to interpret the contents of the temporary carrbor1 file as JSON.

So after running this code, we can look at what is in the sunrise SAS library to see the data pulled down from the web as SAS has interpreted the JSON. Yay! And of course, once we have data in a SAS library, we can run DATA steps, PROC SQL, and other SAS procs on that data just as with the data sets in any other SAS library.

In fact, here is a SAS Studio session after running the code above.



```
1          OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
72
73         filename carrbor1 temp;
74
75         proc http
76             url="%nrstr(https://api.sunrise-sunset.org/json?lat=35.9101&lng=-79.0753&date=2019-
76       ! 01-01)"
77             method="GET"
78             out=carrbor1;
79         run;

NOTE: PROCEDURE HTTP used (Total process time):
      real time           0.34 seconds
      cpu time            0.01 seconds


80         libname sunrise json fileref=carrbor1;
NOTE: JSON data is only read once.  To read the JSON again, reassign the JSON LIBNAME.
NOTE: Libref SUNRISE was successfully assigned as follows:
      Engine:        JSON
      Physical Name: C:\Users\ucckjr\AppData\Local\Temp\SAS Temporary
      Files\_TD26316_PC309_\#LN00146
81
82         OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
94
```

Note that there are three data sets in the sunrise library. We will take a closer look at them in a later section of these API notes.

What if I need a key even to see data in a web browser, as with the openweather API? Well, we can send the API credentials in our URL string. For example, I have successfully used this code, which includes my openweather API key, to pull openweather data into a SAS session:

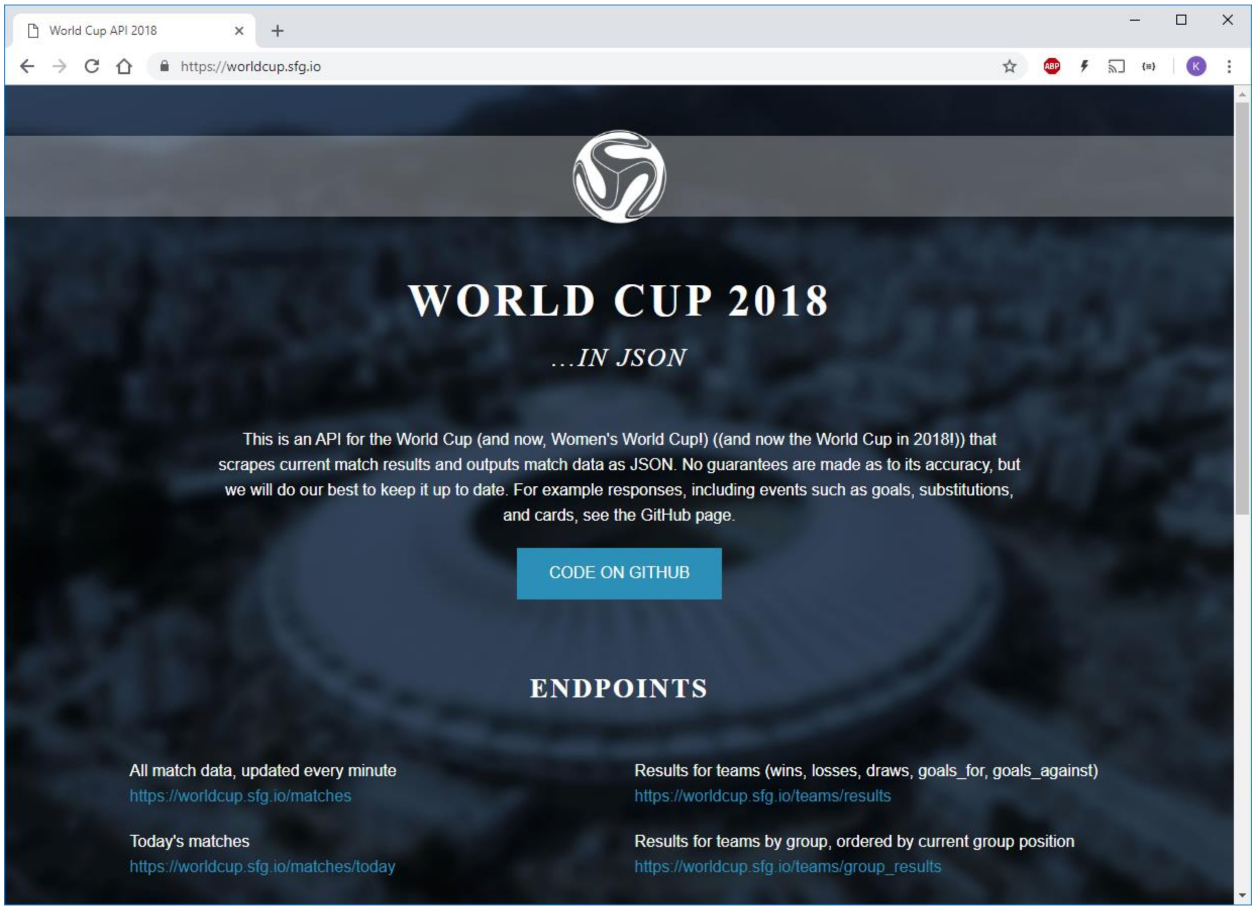
```
filename opentest temp;

proc http

url="%nrstr(http://api.openweathermap.org/data/2.5/find?q=Carrboro&uni
ts=imperial&type=accurate&mode=json&APPID=xxxxxxxxxxxxxxxxxxxxxxxxxx)"
    method="GET"
    out=opentest;
run;

libname open json fileref=opentest;
```

In the remaining sections of these notes, and in the exercises, we will work with PROC HTTP and the JSON engine to get data concerning world cup soccer, sunrise and sunset times, air quality, and Game of Thrones.

World Cup Soccer API Example

- Using the JSON engine in a LIBNAME statement

This API example uses a world cup soccer site, https://worldcup.sfg.io/ .



WORLD CUP 2018

…IN JSON

This is an API for the World Cup (and now, Women's World Cup!) ((and now the World Cup in 2018!)) that scrapes current match results and outputs match data as JSON. No guarantees are made as to its accuracy, but we will do our best to keep it up to date. For example responses, including events such as goals, substitutions, and cards, see the GitHub page.

ENDPOINTS

All match data, updated every minute
https://worldcup.sfg.io/matches

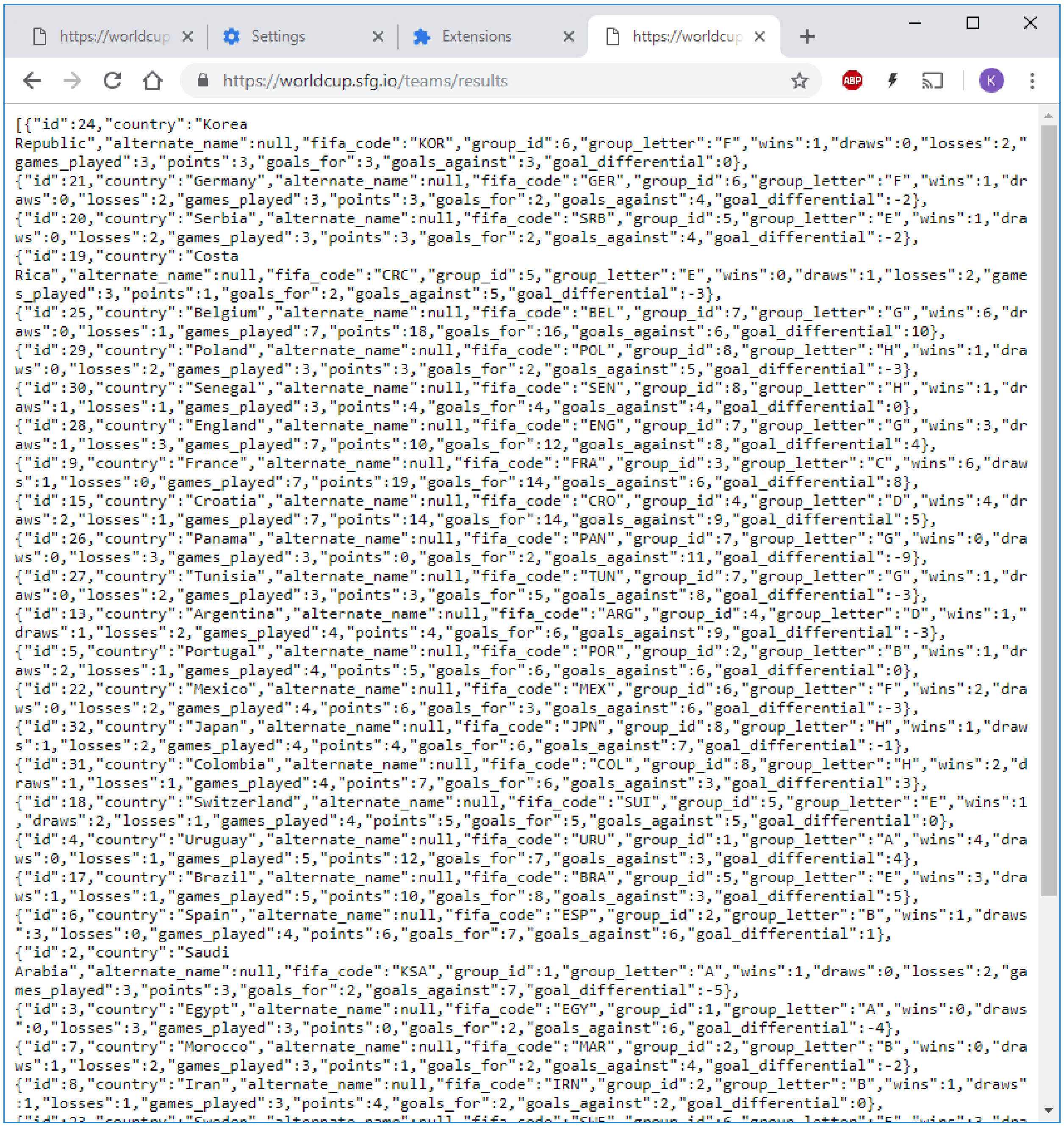
Today's matches
https://worldcup.sfg.io/matches/today

Results for teams (wins, losses, draws, goals_for, goals_against)
https://worldcup.sfg.io/teams/results

Results for teams by group, ordered by current group position
https://worldcup.sfg.io/teams/group_results

The soccer lovers who maintain this site make the most current world cup soccer match data available to the rest of us in the form of JSON files. The available data always comes from the most recent world cup, whether men's or women's. Since the men's world cup was in summer 2018, the data currently available is from that competition. When the women's world cup starts in June 2019, the men's world cup data will be replaced by women's match data.

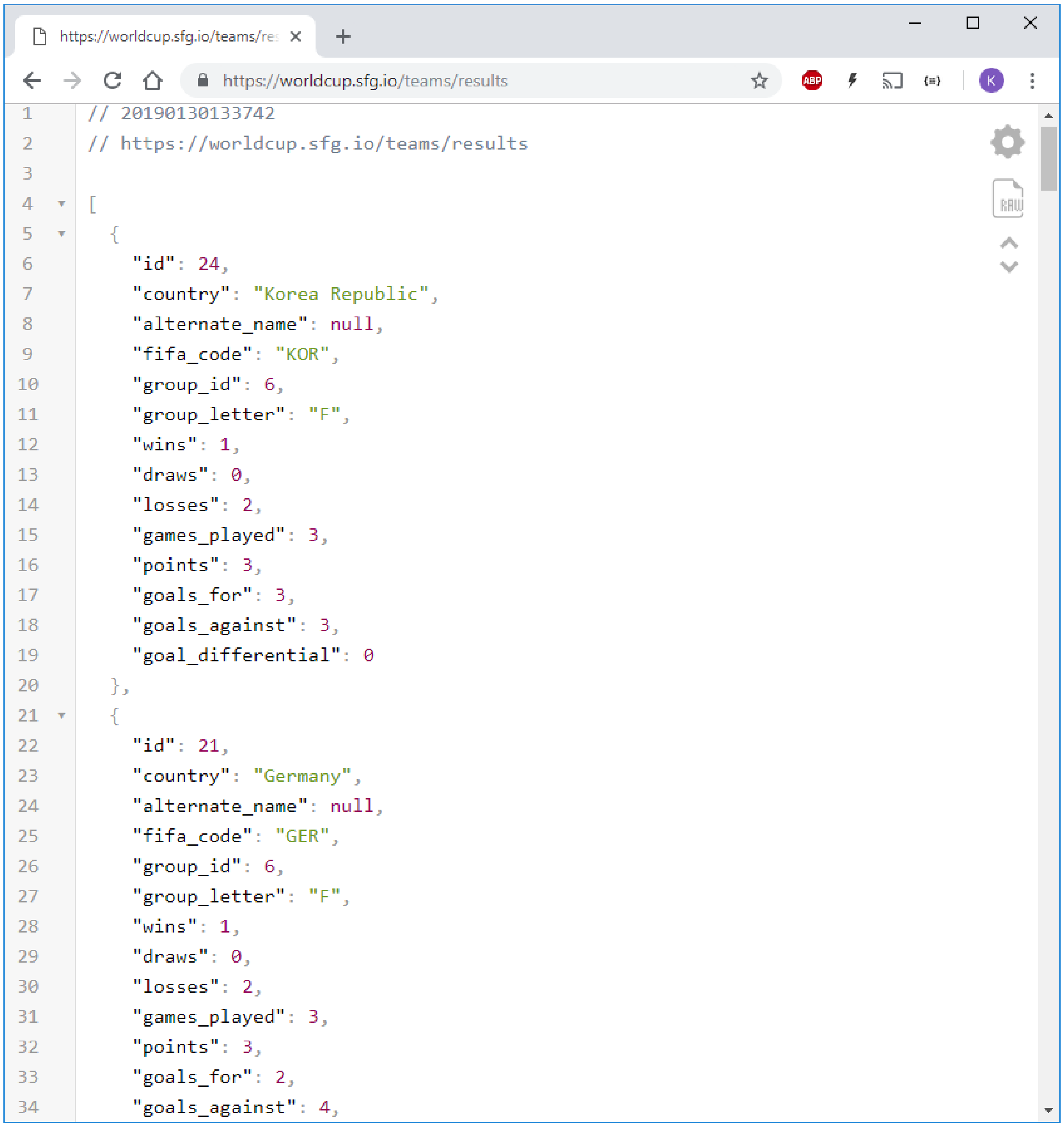
Different links on the page go to different types and levels of data. For example, if we click on https://worldcup.sfg.io/teams/results and we are using Chrome, we might see something like this:


https://worldcup.sfg.io/teams/results

```
[{"id":24,"country":"Korea
Republic","alternate_name":null,"fifa_code":"KOR","group_id":6,"group_letter":"F","wins":1,"draws":0,"losses":2,"
games_played":3,"points":3,"goals_for":3,"goals_against":3,"goal_differential":0},
{"id":21,"country":"Germany","alternate_name":null,"fifa_code":"GER","group_id":6,"group_letter":"F","wins":1,"dr
aws":0,"losses":2,"games_played":3,"points":3,"goals_for":2,"goals_against":4,"goal_differential":-2},
{"id":20,"country":"Serbia","alternate_name":null,"fifa_code":"SRB","group_id":5,"group_letter":"E","wins":1,"dra
ws":0,"losses":2,"games_played":3,"points":3,"goals_for":2,"goals_against":4,"goal_differential":-2},
{"id":19,"country":"Costa
Rica","alternate_name":null,"fifa_code":"CRC","group_id":5,"group_letter":"E","wins":0,"draws":1,"losses":2,"game
s_played":3,"points":1,"goals_for":2,"goals_against":5,"goal_differential":-3},
{"id":25,"country":"Belgium","alternate_name":null,"fifa_code":"BEL","group_id":7,"group_letter":"G","wins":6,"dr
aws":0,"losses":1,"games_played":7,"points":18,"goals_for":16,"goals_against":6,"goal_differential":10},
{"id":29,"country":"Poland","alternate_name":null,"fifa_code":"POL","group_id":8,"group_letter":"H","wins":1,"dra
ws":0,"losses":2,"games_played":3,"points":3,"goals_for":2,"goals_against":5,"goal_differential":-3},
{"id":30,"country":"Senegal","alternate_name":null,"fifa_code":"SEN","group_id":8,"group_letter":"H","wins":1,"dr
aws":1,"losses":1,"games_played":3,"points":4,"goals_for":4,"goals_against":4,"goal_differential":0},
{"id":28,"country":"England","alternate_name":null,"fifa_code":"ENG","group_id":7,"group_letter":"G","wins":3,"dr
aws":1,"losses":3,"games_played":7,"points":10,"goals_for":12,"goals_against":8,"goal_differential":4},
{"id":9,"country":"France","alternate_name":null,"fifa_code":"FRA","group_id":3,"group_letter":"C","wins":6,"draw
s":1,"losses":0,"games_played":7,"points":19,"goals_for":14,"goals_against":6,"goal_differential":8},
{"id":15,"country":"Croatia","alternate_name":null,"fifa_code":"CRO","group_id":4,"group_letter":"D","wins":4,"dr
aws":2,"losses":1,"games_played":7,"points":14,"goals_for":14,"goals_against":9,"goal_differential":5},
{"id":26,"country":"Panama","alternate_name":null,"fifa_code":"PAN","group_id":7,"group_letter":"G","wins":0,"dra
ws":0,"losses":3,"games_played":3,"points":0,"goals_for":2,"goals_against":11,"goal_differential":-9},
{"id":27,"country":"Tunisia","alternate_name":null,"fifa_code":"TUN","group_id":7,"group_letter":"G","wins":1,"dr
aws":0,"losses":2,"games_played":3,"points":3,"goals_for":5,"goals_against":8,"goal_differential":-3},
{"id":13,"country":"Argentina","alternate_name":null,"fifa_code":"ARG","group_id":4,"group_letter":"D","wins":1,"
draws":1,"losses":2,"games_played":4,"points":4,"goals_for":6,"goals_against":9,"goal_differential":-3},
{"id":5,"country":"Portugal","alternate_name":null,"fifa_code":"POR","group_id":2,"group_letter":"B","wins":1,"dr
aws":2,"losses":1,"games_played":4,"points":5,"goals_for":6,"goals_against":6,"goal_differential":0},
{"id":22,"country":"Mexico","alternate_name":null,"fifa_code":"MEX","group_id":6,"group_letter":"F","wins":2,"dra
ws":0,"losses":2,"games_played":4,"points":6,"goals_for":3,"goals_against":6,"goal_differential":-3},
{"id":32,"country":"Japan","alternate_name":null,"fifa_code":"JPN","group_id":8,"group_letter":"H","wins":1,"draw
s":1,"losses":2,"games_played":4,"points":4,"goals_for":6,"goals_against":7,"goal_differential":-1},
{"id":31,"country":"Colombia","alternate_name":null,"fifa_code":"COL","group_id":8,"group_letter":"H","wins":2,"d
raws":1,"losses":1,"games_played":4,"points":7,"goals_for":6,"goals_against":3,"goal_differential":3},
{"id":18,"country":"Switzerland","alternate_name":null,"fifa_code":"SUI","group_id":5,"group_letter":"E","wins":1
,"draws":2,"losses":1,"games_played":4,"points":5,"goals_for":5,"goals_against":5,"goal_differential":0},
{"id":4,"country":"Uruguay","alternate_name":null,"fifa_code":"URU","group_id":1,"group_letter":"A","wins":4,"dra
ws":0,"losses":1,"games_played":5,"points":12,"goals_for":7,"goals_against":3,"goal_differential":4},
{"id":17,"country":"Brazil","alternate_name":null,"fifa_code":"BRA","group_id":5,"group_letter":"E","wins":3,"dra
ws":1,"losses":1,"games_played":5,"points":10,"goals_for":8,"goals_against":3,"goal_differential":5},
{"id":6,"country":"Spain","alternate_name":null,"fifa_code":"ESP","group_id":2,"group_letter":"B","wins":1,"draws
":3,"losses":0,"games_played":4,"points":6,"goals_for":7,"goals_against":6,"goal_differential":1},
{"id":2,"country":"Saudi
Arabia","alternate_name":null,"fifa_code":"KSA","group_id":1,"group_letter":"A","wins":1,"draws":0,"losses":2,"ga
mes_played":3,"points":3,"goals_for":2,"goals_against":7,"goal_differential":-5},
{"id":3,"country":"Egypt","alternate_name":null,"fifa_code":"EGY","group_id":1,"group_letter":"A","wins":0,"draws
":0,"losses":3,"games_played":3,"points":0,"goals_for":2,"goals_against":6,"goal_differential":-4},
{"id":7,"country":"Morocco","alternate_name":null,"fifa_code":"MAR","group_id":2,"group_letter":"B","wins":0,"dra
ws":1,"losses":2,"games_played":3,"points":1,"goals_for":2,"goals_against":4,"goal_differential":-2},
{"id":8,"country":"Iran","alternate_name":null,"fifa_code":"IRN","group_id":2,"group_letter":"B","wins":1,"draws"
:1,"losses":1,"games_played":3,"points":4,"goals_for":2,"goals_against":2,"goal_differential":0},
{"id":23,"country":"Sweden","alternate_name":null,"fifa_code":"SWE","group_id":6,"group_letter":"F","wins":3,"dra
```

Yikes, it looks like there might be some kind of structure here, but what is it?

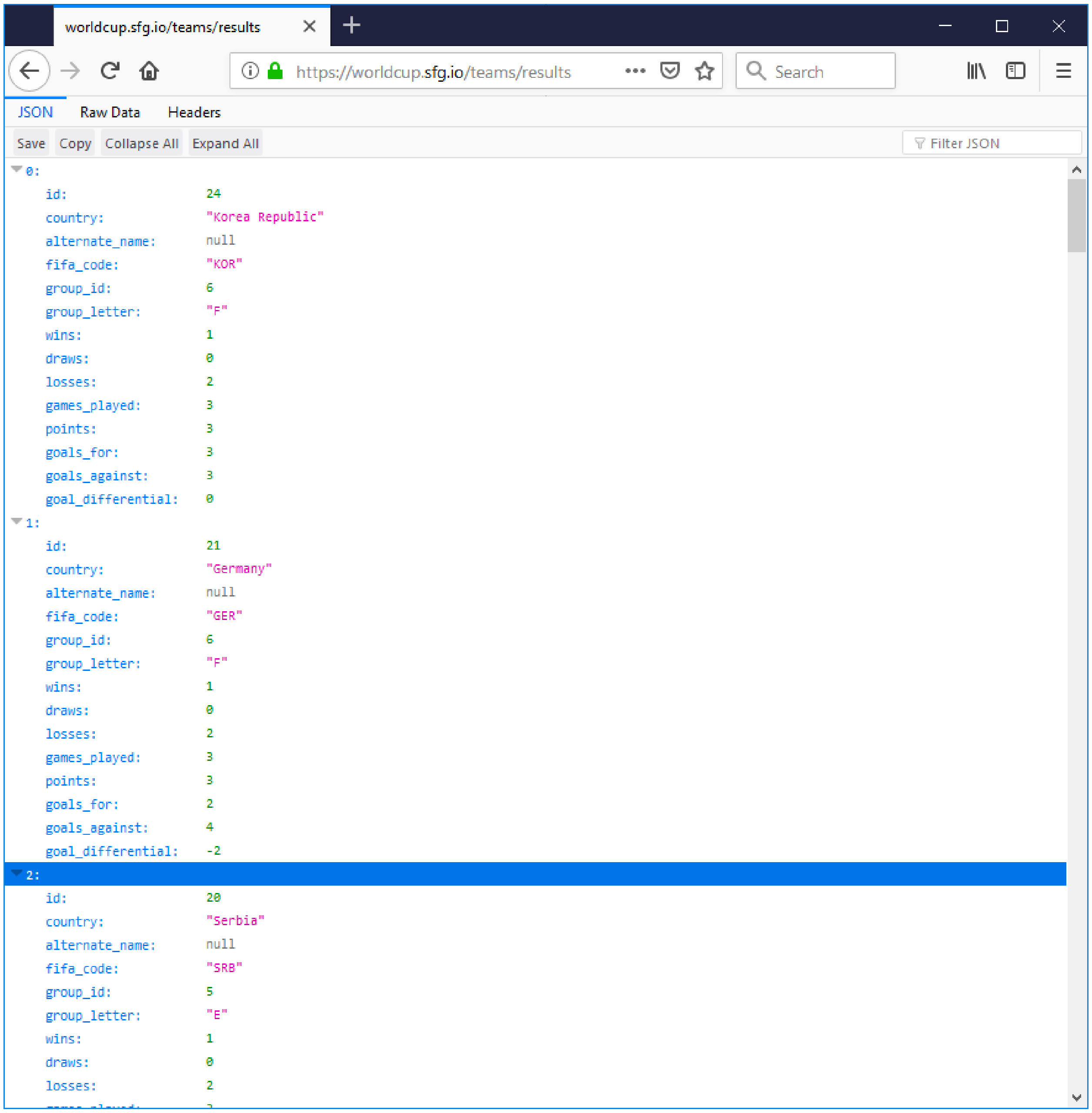
Fortunately, you can install and enable a Chrome extension for reading JSON files, and then you'll see something like this (using an extension call JSON Viewer):


https://worldcup.sfg.io/teams/results

```
// 20190130133742
// https://worldcup.sfg.io/teams/results

[
  {
    "id": 24,
    "country": "Korea Republic",
    "alternate_name": null,
    "fifa_code": "KOR",
    "group_id": 6,
    "group_letter": "F",
    "wins": 1,
    "draws": 0,
    "losses": 2,
    "games_played": 3,
    "points": 3,
    "goals_for": 3,
    "goals_against": 3,
    "goal_differential": 0
  },
  {
    "id": 21,
    "country": "Germany",
    "alternate_name": null,
    "fifa_code": "GER",
    "group_id": 6,
    "group_letter": "F",
    "wins": 1,
    "draws": 0,
    "losses": 2,
    "games_played": 3,
    "points": 3,
    "goals_for": 2,
    "goals_against": 4,
```

OK, that makes the structure much clearer.

I think Firefox has a built-in JSON viewer, and on Firefox https://worldcup.sfg.io/teams/results looks like this:



worldcup.sfg.io/teams/results
https://worldcup.sfg.io/teams/results
Search
JSON
Raw Data
Headers
Save
Copy
Collapse All
Expand All
Filter JSON

```
0:
    id:                 24
    country:            "Korea Republic"
    alternate_name:     null
    fifa_code:          "KOR"
    group_id:           6
    group_letter:       "F"
    wins:               1
    draws:              0
    losses:             2
    games_played:       3
    points:             3
    goals_for:          3
    goals_against:      3
    goal_differential:  0
1:
    id:                 21
    country:            "Germany"
    alternate_name:     null
    fifa_code:          "GER"
    group_id:           6
    group_letter:       "F"
    wins:               1
    draws:              0
    losses:             2
    games_played:       3
    points:             3
    goals_for:          2
    goals_against:      4
    goal_differential:  -2
2:
    id:                 20
    country:            "Serbia"
    alternate_name:     null
    fifa_code:          "SRB"
    group_id:           5
    group_letter:       "E"
    wins:               1
    draws:              0
    losses:             2
```

OK, we've now seen that the data has structure, but how do we read it? In SAS, we can take advantage of SAS's JSON LIBNAME engine (documentation at https://documentation.sas.com/?docsetId=lestmtsglobal&docsetTarget=n1jfdetszx99ban1rl4zll6tej7j.htm&docsetVersion=9.4&locale=en; also see https://blogs.sas.com/content/sasdummy/2016/12/02/json-libname-engine-sas/).

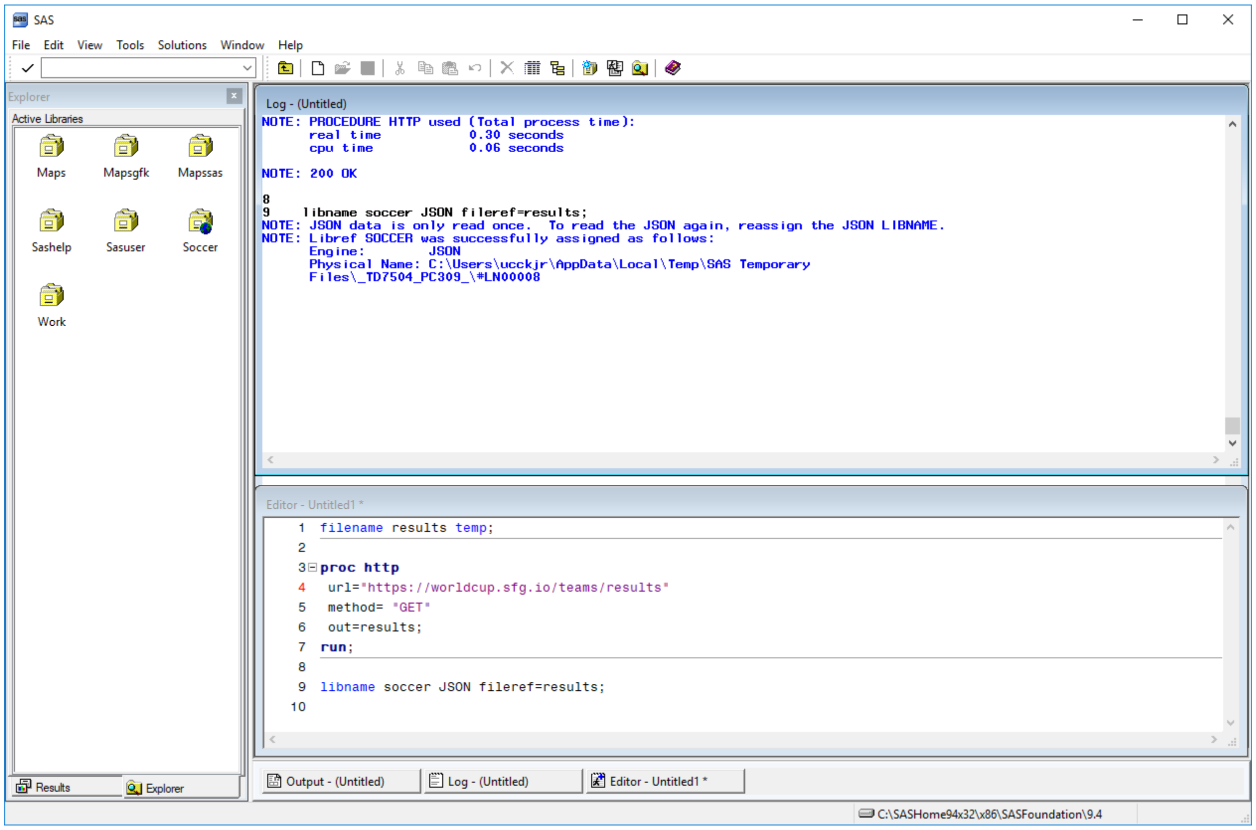
JSON files basically consist of nested lists of name: value pairs, and they enable exchange of information between people, organizations, and applications (note: JSON files have kind of supplanted XML files, which are older and fulfill a similar role). Any modern language provides some facilities for reading JSON files, and SAS is no different.

Code for reading the team results data with SAS:

```
FILENAME results TEMP;

PROC HTTP
      URL="https://worldcup.sfg.io/teams/results"
      METHOD="GET"
      OUT=results;
RUN;

LIBNAME soccer JSON FILEREF=results;
```

Note that “results” appears three times in this code (well, four times if we count its appearance in the URL). These three occurrences need to match: we establish a temporary file reference, our PROC HTTP code writes to that file, and our LIBNAME statement also references the file.

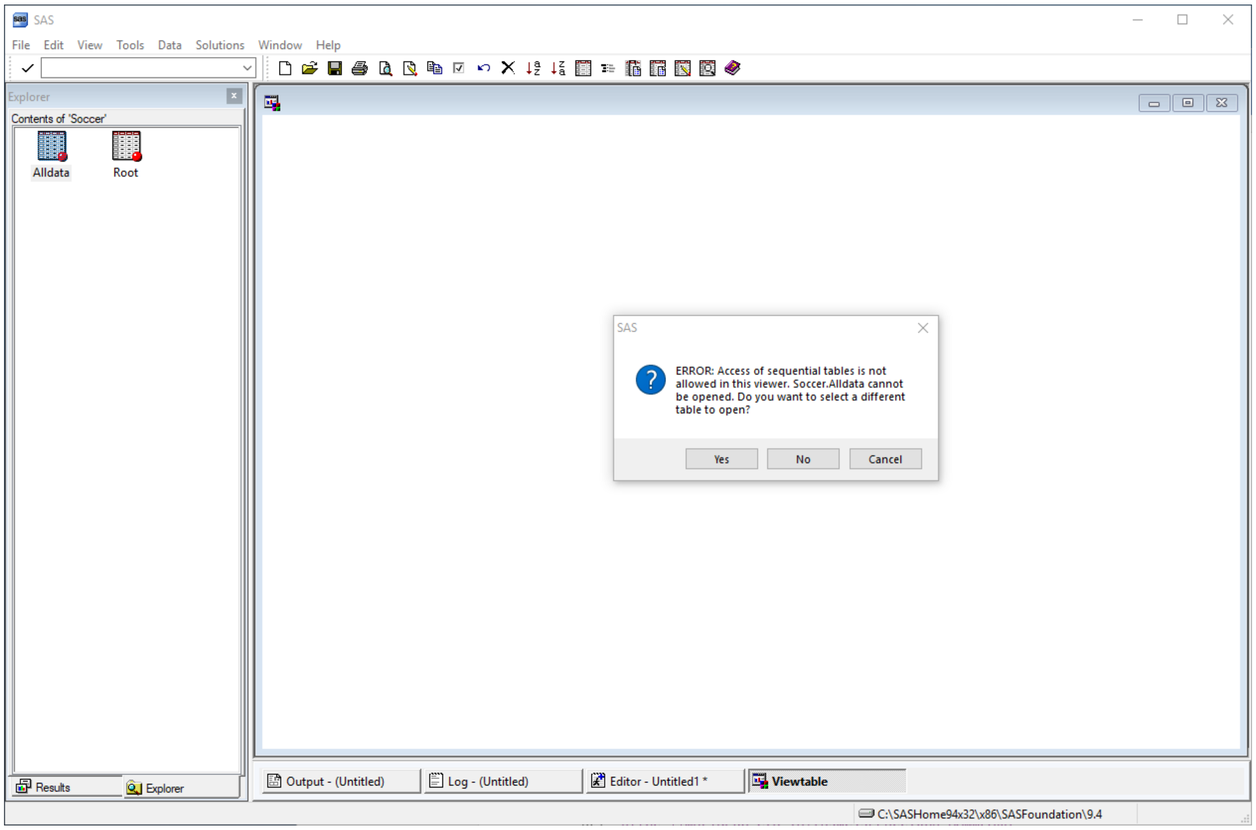
The subsequent LIBNAME statement asks SAS to interpret the data in the file as if it were JSON and to let us access the data through libref *soccer*. If we now go to SAS, will we see a data library named soccer? We do!



```
NOTE: PROCEDURE HTTP used (Total process time):
      real time           0.30 seconds
      cpu time            0.06 seconds

NOTE: 200 OK

8
9    libname soccer JSON fileref=results;
NOTE: JSON data is only read once.  To read the JSON again, reassign the JSON LIBNAME.
NOTE: Libref SOCCER was successfully assigned as follows:
      Engine:        JSON
      Physical Name: C:\Users\ucckjr\AppData\Local\Temp\SAS Temporary
      Files\_TD7504_PC309_\#LN00008
```

```
filename results temp;

proc http
 url="https://worldcup.sfg.io/teams/results"
 method= "GET"
 out=results;
run;

libname soccer JSON fileref=results;
```

If we double-click on Soccer in the Active Libraries window, we see two data sets, Alldata and Root. Yay! But what do these look like? If we double-click on Alldata, what happens? Sadness! We get this disappointing message:



ERROR: Access of sequential tables is not allowed in this viewer. Soccer.Alldata cannot be opened. Do you want to select a different table to open?

So we back out of this aborted attempt to view the data. Note that SAS will allow us to right click on Alldata and Root and select View Columns, but that's not quite as helpful as actually seeing the data. Fortunately, we can use PROC PRINT. Here are the first 10 observations of soccer.alldata and soccer.root:

Alldata

Obs	*P*	*P1*	*V*	*Value*
1	1	id	1	24
2	1	country	1	Korea Republic
3	1	alternate_name	1	null
4	1	fifa_code	1	KOR
5	1	group_id	1	6
6	1	group_letter	1	F
7	1	wins	1	1
8	1	draws	1	0
9	1	losses	1	2
10	1	games_played	1	3

Root

Obs	*ordinal_root*	*id*	*country*	*alternate_name*	*fifa_code*	*group_id*	*group_letter*	*wins*
1	1	24	Korea Republic	.	KOR	6	F	1
2	2	21	Germany	.	GER	6	F	1
3	3	20	Serbia	.	SRB	5	E	1
4	4	19	Costa Rica	.	CRC	5	E	0
5	5	25	Belgium	.	BEL	7	G	6
6	6	29	Poland	.	POL	8	H	1
7	7	30	Senegal	.	SEN	8	H	1
8	8	28	England	.	ENG	7	G	3
9	9	9	France	.	FRA	3	C	6
10	10	15	Croatia	.	CRO	4	D	4

Obs	*draws*	*losses*	*games_played*	*points*	*goals_for*	*goals_against*	*goal_differential*
1	0	2	3	3	3	3	0
2	0	2	3	3	2	4	-2
3	0	2	3	3	2	4	-2
4	1	2	3	1	2	5	-3
5	0	1	7	18	16	6	10
6	0	2	3	3	2	5	-3
7	1	1	3	4	4	4	0
8	1	3	7	10	12	8	4
9	1	0	7	19	14	6	8
10	2	1	7	14	14	9	5

The data format in Alldata doesn't look very helpful, but Root looks like exactly what we need – and all we had to do was use the JSON libname engine!

Root looks like a useable data set, and in fact it is. We can run additional DATA and PROC steps on it to answer our questions. For example, maybe we are interested in what teams played the most games and what teams scored the most points:

```
proc sql;
    title "What team scored the most points in the 2018 Men's World Cup?";
    select country, wins, points, goals_for, goals_against, goal_differential
        from soccer.root
        having points=max(points)
        ;

    title "What team or teams played the most games in the 2018 Men's World 
Cup?";
    select country, games_played, wins, points, goals_for, goals_against, 
goal_differential
        from soccer.root
        having games_played=max(games_played)
        ;
```

What team scored the most points in the 2018 Men's World Cup?

country	wins	points	goals_for	goals_against	goal_differential
France	6	19	14	6	8

What team or teams played the most games in the 2018 Men's World Cup?

country	games_played	wins	points	goals_for	goals_against	goal_differential
Belgium	7	6	18	16	6	10
England	7	3	10	12	8	4
France	7	6	19	14	6	8
Croatia	7	4	14	14	9	5

These results make sense because France was the overall winner, beating Croatia in the championship game, with Belgium and England playing in the game for third place. (Note: I don't think it's necessarily the case that the team scoring the most points overall would win the tournament, but it was true in this world cup.)

As an aside, you don't necessarily need to run PROC HTTP to use this soccer data. You could manually copy the data from the web to a local file and read that local file with the JSON libname engine to do your analyses. I did so, copying the results to a file on my local network and naming the file local_results.json. I established a fileref to that file with a FILENAME statement, used a LIBNAME statement with the JSON engine to read that fileref, and was able to access the data through that libname just as above.

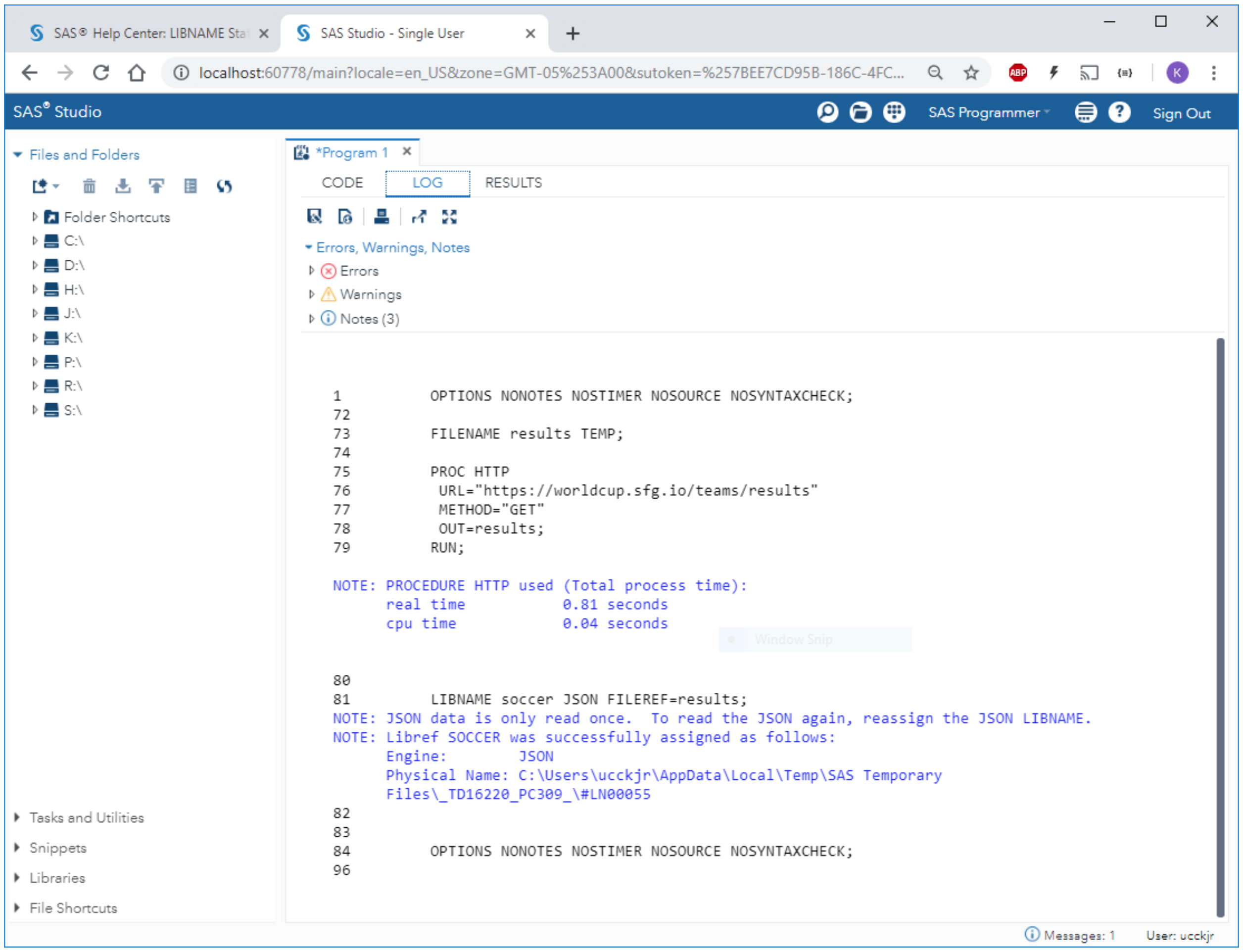
```
FILENAME localres 'P:\BIOS 669 2019\API materials\soccer\local_results.json';
LIBNAME localsoc JSON FILEREF=localres;

PROC PRINT DATA=localsoc.root(OBS=10); TITLE 'local root'; RUN; TITLE;
```

An advantage of reading the data with the URL option to PROC HTTP rather than downloading the data to a static file means that you can read the page IN REAL TIME AS IT IS UPDATED MATCH BY MATCH and thus periodically update the statistics, graphs, or whatever it is that you are generating based on the data. An application that reads data using an API can be updated in almost real time.

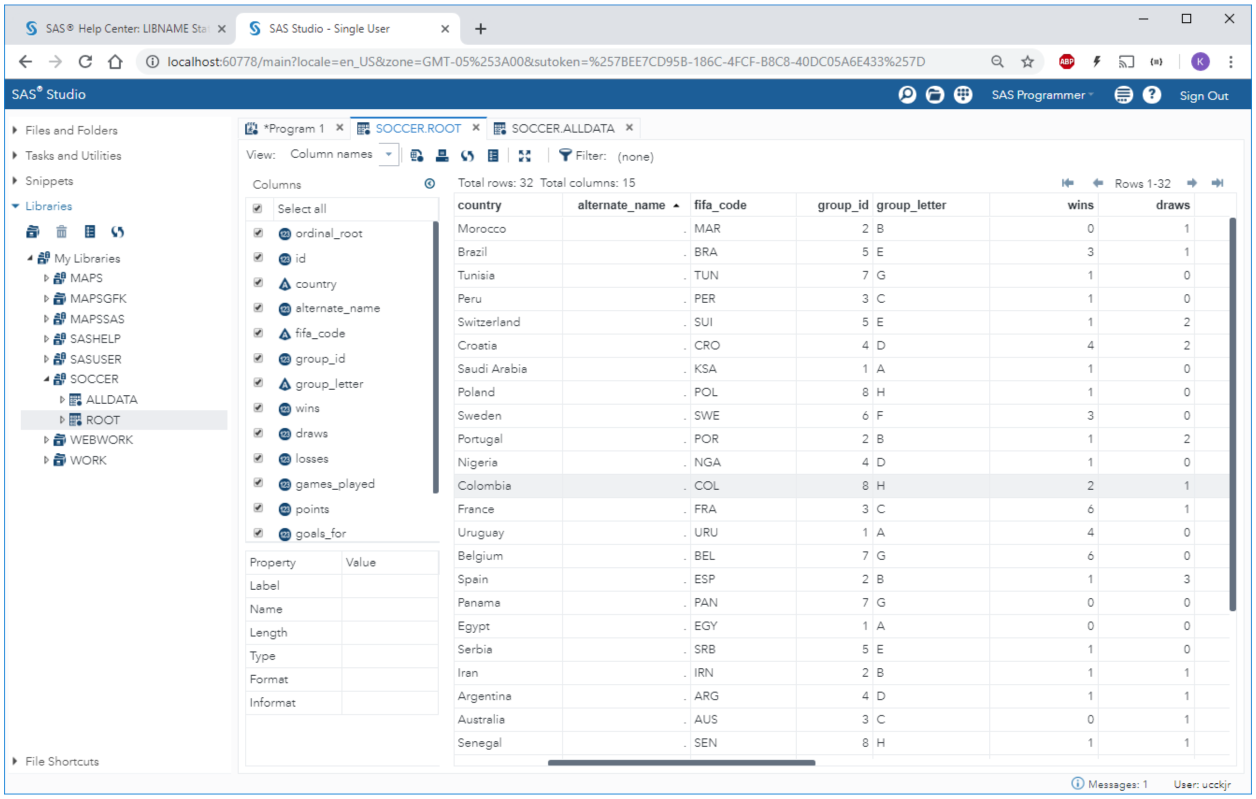
I originally learned about this world cup soccer API at a 2017 Data Matters course at NCSU. The instructors shared this R code for a sequence similar to the above: getting the team results data and analyzing it for the team scoring the most points. I've run the code in January 2019, when the data on the site are from the 2018 world cup.

```
> library("httr")
> library("jsonlite")
> url <- "https:/worldcup.sfg.io/teams/results"
> result <- GET(url)
> result_data <- result$content
> result_data <- rawToChar(result_data)
> fifa_data <- fromJSON(result_data)
> winner_data <- fifa_data[which.max(fifa_data$points),]
> winner_data
  id country alternate_name fifa_code group_id group_letter wins draws losses games_playe
points
9  9  France              NA        FRA        3            C     6     1      0
19
  goals_for goals_against goal_differential
9        14            6                 8
> report <- paste("2018 Men's FIFA World Cup Champion:", winner_data$country, sep=" ")
> print(report)
[1] "2018 Men's FIFA World Cup Champion: France"
```

News flash: SAS Studio / UE out-perform the old-fashioned SAS windowing environment in terms of usability of data in a library created using the JSON engine. Check out the sequence below in which world cup soccer data is read and then I am able to view the root data set directly rather than having to use PROC PRINT.



```
1          OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
72
73         FILENAME results TEMP;
74
75         PROC HTTP
76          URL="https://worldcup.sfg.io/teams/results"
77          METHOD="GET"
78          OUT=results;
79         RUN;

NOTE: PROCEDURE HTTP used (Total process time):
      real time           0.81 seconds
      cpu time            0.04 seconds


80
81         LIBNAME soccer JSON FILEREF=results;
NOTE: JSON data is only read once.  To read the JSON again, reassign the JSON LIBNAME.
NOTE: Libref SOCCER was successfully assigned as follows:
      Engine:        JSON
      Physical Name: C:\Users\ucckjr\AppData\Local\Temp\SAS Temporary
      Files\_TD16220_PC309_\#LN00055
82
83
84         OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
96
```


Total rows: 32 Total columns: 15

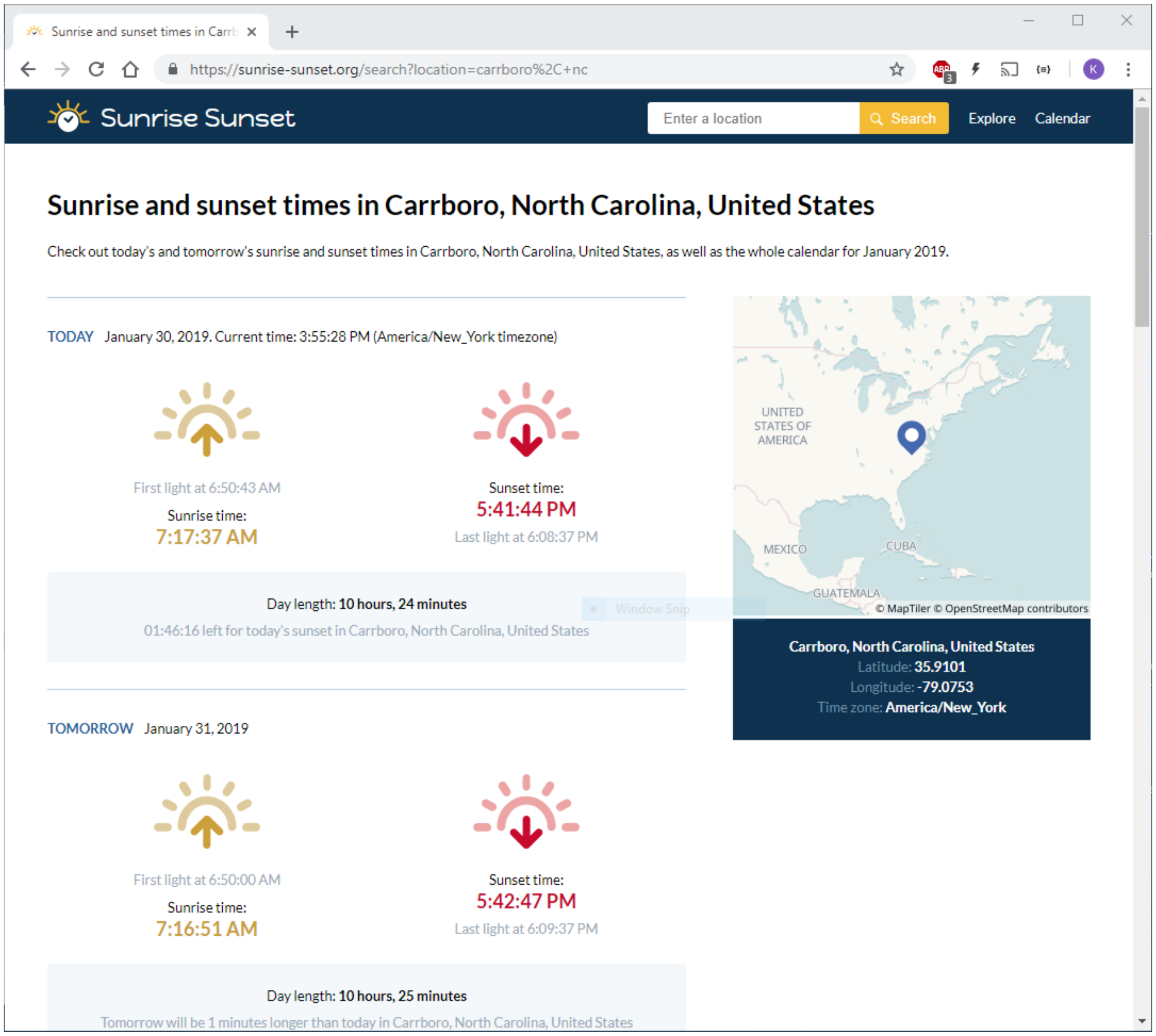
Rows 1-32

country	alternate_name	fifa_code	group_id	group_letter	wins	draws
Morocco	.	MAR	2	B	0	1
Brazil	.	BRA	5	E	3	1
Tunisia	.	TUN	7	G	1	0
Peru	.	PER	3	C	1	0
Switzerland	.	SUI	5	E	1	2
Croatia	.	CRO	4	D	4	2
Saudi Arabia	.	KSA	1	A	1	0
Poland	.	POL	8	H	1	0
Sweden	.	SWE	6	F	3	0
Portugal	.	POR	2	B	1	2
Nigeria	.	NGA	4	D	1	0
Colombia	.	COL	8	H	2	1
France	.	FRA	3	C	6	1
Uruguay	.	URU	1	A	4	0
Belgium	.	BEL	7	G	6	0
Spain	.	ESP	2	B	1	3
Panama	.	PAN	7	G	0	0
Egypt	.	EGY	1	A	0	0
Serbia	.	SRB	5	E	1	0
Iran	.	IRN	2	B	1	1
Argentina	.	ARG	4	D	1	1
Australia	.	AUS	3	C	0	1
Senegal	.	SEN	8	H	1	1

Sunrise / Sunset API Example

- Using %NRSTR for complicated URLs
- Working with date and time values

A fun free API lets you get sunrise and sunset times for any day, any location in the world. The site is https://sunrise-sunset.org . If I go to that site and enter the location Carrboro, NC, I see something like this:



Sunrise and sunset times in Carrboro, North Carolina, United States

Check out today's and tomorrow's sunrise and sunset times in Carrboro, North Carolina, United States, as well as the whole calendar for January 2019.

TODAY January 30, 2019. Current time: 3:55:28 PM (America/New_York timezone)

First light at 6:50:43 AM
Sunrise time:
7:17:37 AM

Sunset time:
5:41:44 PM
Last light at 6:08:37 PM

Day length: 10 hours, 24 minutes
01:46:16 left for today's sunset in Carrboro, North Carolina, United States

Carrboro, North Carolina, United States
Latitude: 35.9101
Longitude: -79.0753
Time zone: America/New_York

TOMORROW January 31, 2019

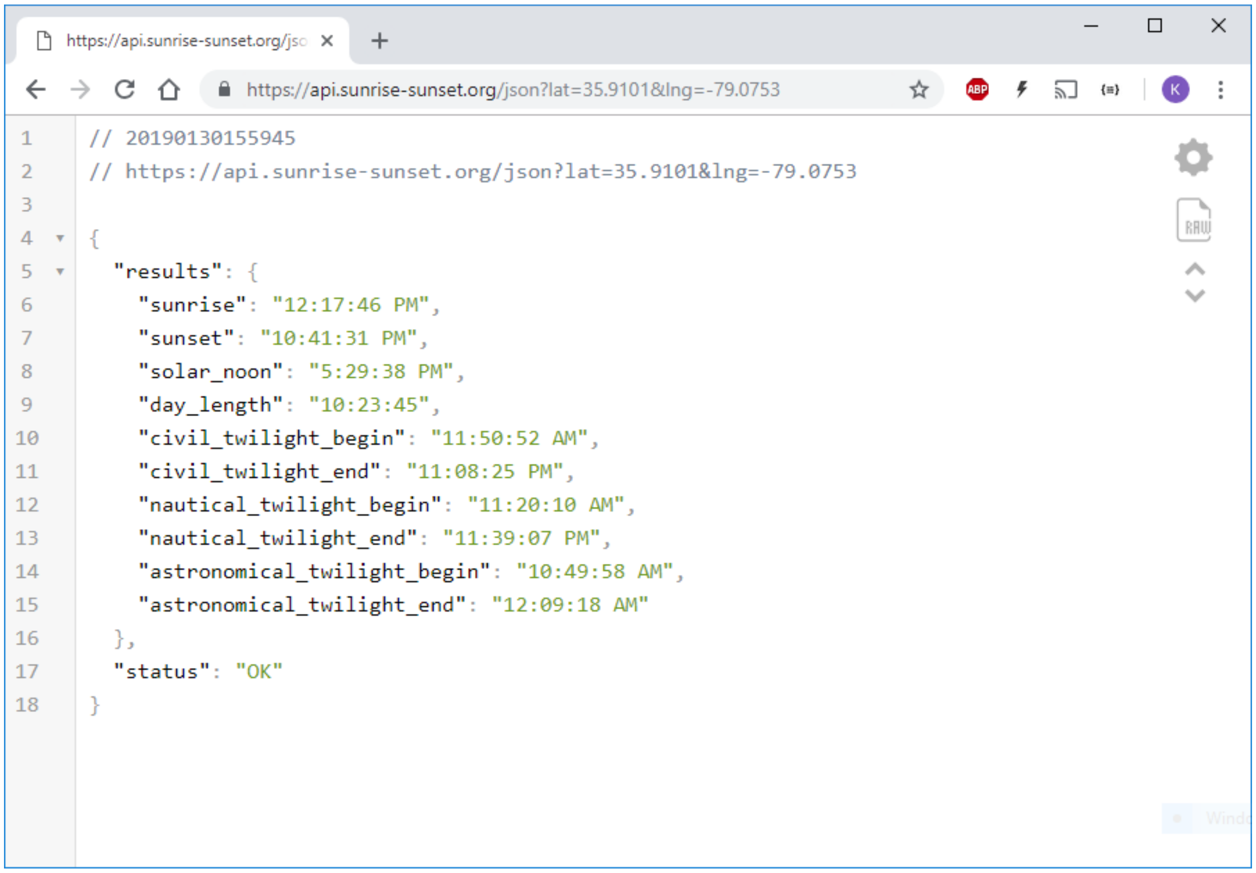
First light at 6:50:00 AM
Sunrise time:
7:16:51 AM

Sunset time:
5:42:47 PM
Last light at 6:09:37 PM

Day length: 10 hours, 25 minutes
Tomorrow will be 1 minutes longer than today in Carrboro, North Carolina, United States

Sunrise and sunset times in Carrb
https://sunrise-sunset.org/search?location=carrboro%2C+nc
Sunrise Sunset
Enter a location
Search
Explore
Calendar
UNITED STATES OF AMERICA
MEXICO
CUBA
GUATEMALA
© MapTiler © OpenStreetMap contributors

Notice that Carrboro's latitude and longitude are provided here, and of course there are many other ways to find out the latitude and longitude of a location. Once I have that information, look what happens when I feed the latitude and longitude of Carrboro to a slight variation of the original web address, https://api.sunrise-sunset.org/json?lat=35.9101&lng=-79.0753 :



```
// 20190130155945
// https://api.sunrise-sunset.org/json?lat=35.9101&lng=-79.0753

{
  "results": {
    "sunrise": "12:17:46 PM",
    "sunset": "10:41:31 PM",
    "solar_noon": "5:29:38 PM",
    "day_length": "10:23:45",
    "civil_twilight_begin": "11:50:52 AM",
    "civil_twilight_end": "11:08:25 PM",
    "nautical_twilight_begin": "11:20:10 AM",
    "nautical_twilight_end": "11:39:07 PM",
    "astronomical_twilight_begin": "10:49:58 AM",
    "astronomical_twilight_end": "12:09:18 AM"
  },
  "status": "OK"
}
```

It's JSON again!

So we can read this sunrise/sunset information just as we did the world cup soccer data, but with a twist or two. Mainly, we need to specify some parameters as part of the URL: the latitude and longitude of the place we are interested in.

If we use code just like we used for the soccer data, that would suggest this:

```
FILENAME carrboro TEMP;

PROC HTTP
    URL="https://api.sunrise-sunset.org/json?lat=35.9101&lng=-79.0753"
    METHOD="GET"
    OUT=carrboro;
RUN;

LIBNAME sunrise JSON FILEREF=carrboro;
```

When we run this code, PROC HTTP seems to work – we do get refreshed data – but this message also appears:

```
PROC HTTP
    URL="https://api.sunrise-sunset.org/json?lat=35.9101&lng=-79.0753"
WARNING: Apparent symbolic reference LNG not resolved.
    METHOD="GET"
    OUT=carrboro;
RUN;
```

That is, SAS is trying to figure out what &lng means as if it were a macro variable! But we are using it here as part of our URL to pass information, and using & as a separator is absolutely the needed URL syntax here – there is no substitute for it. So how can we ask SAS not to treat &lng as a macro variable reference? The magic %NRSTR macro quoting function to the rescue!

What does %NRSTR do? At compilation time, it "masks" a series of special characters such as + - ; , # and most importantly % and &. "Masking" means not interpreting those characters as meaningful to SAS – just let them pass through like any other characters. Thus, with %NRSTR the & in &lng is NOT interpreted as signaling a macro variable, which is exactly the behavior we need.

To write the above code with %NRSTR, we wrap the URL value in a %NRSTR( ) call, and then put double quotes around the macro function call, as shown below.

```
FILENAME carrbor1 TEMP;

PROC HTTP
    URL="%NRSTR(https://api.sunrise-sunset.org/json?lat=35.9101&lng=-79.0753)"
    METHOD="GET"
    OUT=carrbor1;
RUN;

LIBNAME sunrise JSON FILEREF=carrbor1;
```

When we run this code, we get no complaints in the SAS log. So what do we find in the SUNRISE library? We find three data sets named Alldata, Results, and Root. If we check out these data sets by PROC PRINTing them, we find that Root contains nothing of interest, Alldata contains the labels in one column and values in another (not very useful), and Results contains one row of data with well-named variables.

Results

ordinal_root	*ordinal_results*	*sunrise*	*sunset*	*solar_noon*	*day_length*	*civil_twilight_begin*	*civil_twilight_end*
1	1	12:17:01 PM	10:42:33 PM	5:29:47 PM	10:25:32	11:50:09 AM	11:09:25 PM

nautical_twilight_end	*astronomical_twilight_begin*	*astronomical_twilight_end*
11:40:04 PM	10:49:21 AM	12:10:13 AM

Wait, can this be right? I'm working on this document in late January, and the listing above says that sunrise is at 12:17 PM. If we check the documentation for the API at https://sunrise-sunset.org/api, we find that times are always returned in UTC and do not include summer time adjustments. UTC means Coordinated Universal Time, and compared with Chapel Hill's time zone of Eastern Standard Time, UTC is 5 hours ahead. So if we subtract 5 hours from 12:17 PM (just after noon), we get 7:17 AM, which does seem like what I've been observing this week.

The API documentation also notes that you can send a date to the API as well as latitude and longitude, so that you can get sunrise, sunset, etc. for anywhere on earth on any date. If you don't send a date, then the times returned are for the current date, which is what we see above.

Let's make up a problem for ourselves. Say we want to find out the time of sunrise in Carrboro on the first day of the month for every month in 2019. We'll just make a data set of those times, but you can imagine some creative person using these times to make a graphical representation of Carrboro sunrises over the course of a year – just the type of thing that APIs are for (someone retrieves information and uses it in a creative way).

We will start by requesting the sunrise time for January 1, 2019, and converting that to the correct Eastern Standard Time. Then we will use that code as the basis for a macro to request the sunrise time for the first day of every month in 2019. We'll call the macro 12 times and stack up the results to get the list we want. Note that this work won't adjust for Eastern Daylight Time in the warmer months (spring forward in the spring, fall back in the fall), though that would be easy enough to do if the dates to and from EDT weren't shifted around every year by the powers-that-be.

This code for January works great; notice that we specify the date in the format yyyy-mm-dd as directed on the API website.

```
FILENAME carrbor1 TEMP;

PROC HTTP
    URL="%nrstr(https://api.sunrise-sunset.org/json?lat=35.9101&lng=-
79.0753&date=2019-01-01)"
    METHOD="GET"
    OUT=carrbor1;
RUN;

LIBNAME sunrise JSON FILEREF=carrbor1;
```

The Results data set from this run is

Carrboro - January - Results

ordinal_root	*ordinal_results*	*sunrise*	*sunset*	*solar_noon*	*day_length*	*civil_twilight_begin*	*civil_twilight_end*
1	1	12:26:40 PM	10:13:29 PM	5:20:05 PM	09:46:49	11:58:22 AM	10:41:47 PM

nautical_twilight_end	*astronomical_twilight_begin*	*astronomical_twilight_end*
11:13:43 PM	10:55:19 AM	11:44:50 PM

This seems right – note the very early sunrise (7:27 AM), very early sunset (5:13 PM) , and short day length (less than 10 hours).

My conversion code, which aims to show sunrise in Eastern Standard Time:

```
data january;
    set sunrise.results(keep=sunrise);

    * the API returns all values as strings;
    * start by pulling off the pieces useful to us: the time, the hour, and
whether AM or PM;
    sunrise_time_part=substr(sunrise,1,9);
    sunrise_hour= input(substr(sunrise,1,2),best2.);
    ampm=substr(sunrise,10);

    * now convert the time to a SAS time value (seconds since midnight);
    sunrise_time = input(sunrise_time_part, time9.);

    * but that time value might not be correct depending on whether time
returned by the API was tagged AM or PM;
    * converting to military time (0 - 24 with no AM or PM) is our safest
bet);
    * military time examples:  0-1 military is midnight to 1 AM, 12-13
military is noon to 1 PM;
    * so if the API time is PM and between 1:00 and 11:59, we need to add 12
hours to get to military time;
    * since SAS times are in seconds, we need to add 12 hours worth of
seconds, which is 12*60*60;
    if ampm='PM' and 1<=sunrise_hour<12 then
        sunrise_time_military = sunrise_time + 12*60*60;
    else sunrise_time_military= sunrise_time;

    * OK, once we have the API-provided time in military form, we simply need
to subtract 5 hours to get EST;
    sunrise_time_est=sunrise_time_military - 5*60*60;

    format sunrise_time_est time9.;
run;
```

```
title 'January';
proc print data=january;
run;
```

The printed sunrise_time_est value is 7:26:40, which we know is AM (when the sun always rises!).

Here's my code for a macro that we can call for each month. We'll call it 12 times and then stack up the resulting one-record data sets and print to see Carrboro sunrise times for the first day of each month in 2019. I will talk through this macro in the video for this example. The main new thing here is how I'm constructing the URL value to send to PROC HTTP, with the complication of needing to use %NRSTR and incorporating a value passed in as a macro parameter (the month number).

The macro parameters:

m is a number for the month that I can plug in where helpful – numbers less than 10 have a leading 0 to match the API's expected date string

name is the month name to use in titles, as a data set name, and as a variable value – passed in unquoted for maximum flexibility

```
%macro whichmonth(m= ,name=);

filename carrb&m temp;

proc http
    url="%nrstr(https://api.sunrise-sunset.org/json?lat=35.9101&lng=-
79.0753&date=2019-)&m.%nrstr(-01)"
    method="GET"
    out=carrb&m;
run;

libname sunrise json fileref=carrb&m;

title "Carrboro - &name raw results";
proc print data=sunrise.results;
run;

data &name;
    set sunrise.results(keep=sunrise);

    length month $10;
    month="&name";
    sunrise_time_part=substr(sunrise,1,9);
    sunrise_hour= input(substr(sunrise,1,2),best2.);
    ampm=substr(sunrise,10);

    sunrise_time = input(sunrise_time_part, time9.);

    if ampm='PM' and 1<=sunrise_hour<12 then
        sunrise_time_military = sunrise_time + 12*60*60;
    else sunrise_time_military= sunrise_time;

    sunrise_time_est=sunrise_time_military - 5*60*60;

    format sunrise_time_est time9.;
run;

title "&name";
proc print data=&name;
run;

%mend;

%whichmonth(m=01,name=January)
%whichmonth(m=02,name=February)
%whichmonth(m=03,name=March)
%whichmonth(m=04,name=April)
%whichmonth(m=05,name=May)
%whichmonth(m=06,name=June)
%whichmonth(m=07,name=July)
%whichmonth(m=08,name=August)
%whichmonth(m=09,name=September)
%whichmonth(m=10,name=October)
%whichmonth(m=11,name=November)
%whichmonth(m=12,name=December)


data stack;
    set
        January  (keep=month sunrise_time_est)
        February (keep=month sunrise_time_est)
        March    (keep=month sunrise_time_est)
        April    (keep=month sunrise_time_est)
        May      (keep=month sunrise_time_est)
        June     (keep=month sunrise_time_est)
        July     (keep=month sunrise_time_est)
        August   (keep=month sunrise_time_est)
        September(keep=month sunrise_time_est)
        October  (keep=month sunrise_time_est)
        November (keep=month sunrise_time_est)
        December (keep=month sunrise_time_est)
        ;
run;
```

First day of the month sunrise times, Carrboro, 2019

Month	*Sunrise time (EST)*
January	7:26:40
February	7:16:13
March	6:45:36
April	6:01:49
May	5:23:13
June	5:00:54
July	5:03:41
August	5:24:13
September	5:48:30
October	6:11:39
November	6:39:06
December	7:08:28

Let's look more closely at how the macro constructs the URL string for PROC HTTP. Here's the URL code from the macro.

```
    url="%nrstr(https://api.sunrise-sunset.org/json?lat=35.9101&lng=-
79.0753&date=2019-)&m.%nrstr(-01)"
```

First, notice that the entire string is in double quotes. This means that any macro variables will resolve. Are there any macro variables that need to be resolved here? YES – the month value &m that we are passing in as a macro parameter.

Second, notice that %NRSTR is used twice, at the beginning of the string and at the end. The parentheses for the first %NRSTR close right before macro variable &m. If the &m had been inside the parentheses, that would have been a signal to SAS that &m is not a macro variable (just like &lng and

&date are not macro variables, so we make sure to enclose them in %NRSTR). But since we want &m to be interpreted as a macro variable, we must put it outside the parentheses.

Third, notice that we write &m as &m. to make sure that SAS knows this is the end of the macro variable name. This might not be necessary here, but it doesn't hurt (in any case, SAS will not include the . in the constructed URL string).

Fourth, we end the string with another call to %NRSTR and have it enclose the ending part of the date value. This ensures that SAS interprets -01 as just a string of characters.

```
"%nrstr(https://api.sunrise-sunset.org/json?lat=35.9101&lng=-
79.0753&date=2019-)&m.%nrstr(-01)"
```

So if we call the macro with m=02, the constructed string will be

```
https://api.sunrise-sunset.org/json?lat=35.9101&lng=-79.0753&date=2019-02-01
```

How can you test whether this is a good URL? Simply bring up a browser and paste in this string as an address! You should see some JSON lines supplied by the sunrise-sunset API.

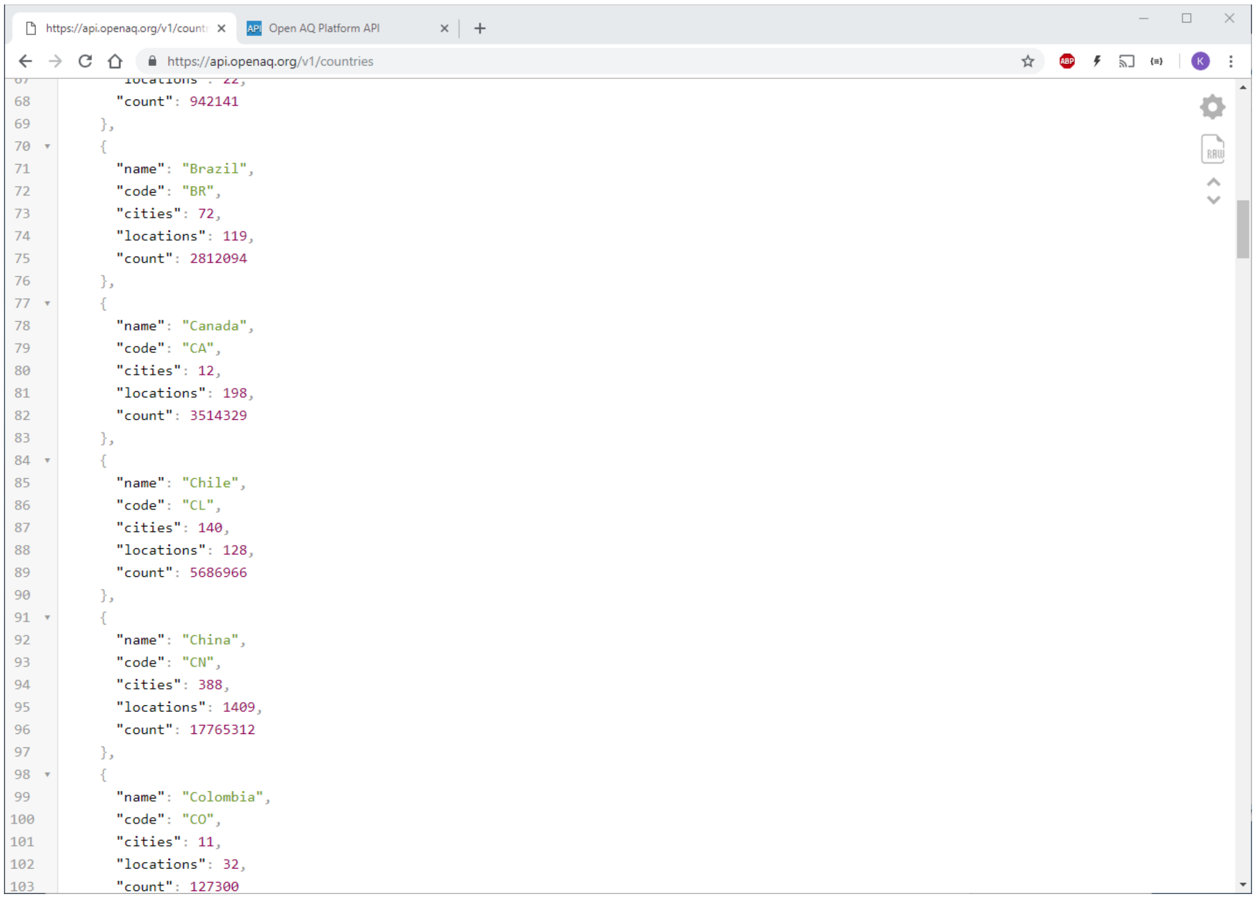
Air Quality API Example

- Making and using a custom JSON map
- More on working with date and time values

Website https://api.openaq.org, documented at https://docs.openaq.org, supplies hour-by-hour air quality data for the last 90 days for hundreds of locations around the world. Wow! Air quality measures vary from location to location, but they typically include levels of noxious gases such as ozone as well as measures of particulate matter of one or more sizes. As with the other APIs we have looked at, data is made available in JSON form.

Say our goal is to grab particulate matter data for a particular location over the last 90 days and graph that data. Somewhat randomly, I've selected Vancouver, Canada (on the west coast not too far north of Seattle, Washington).

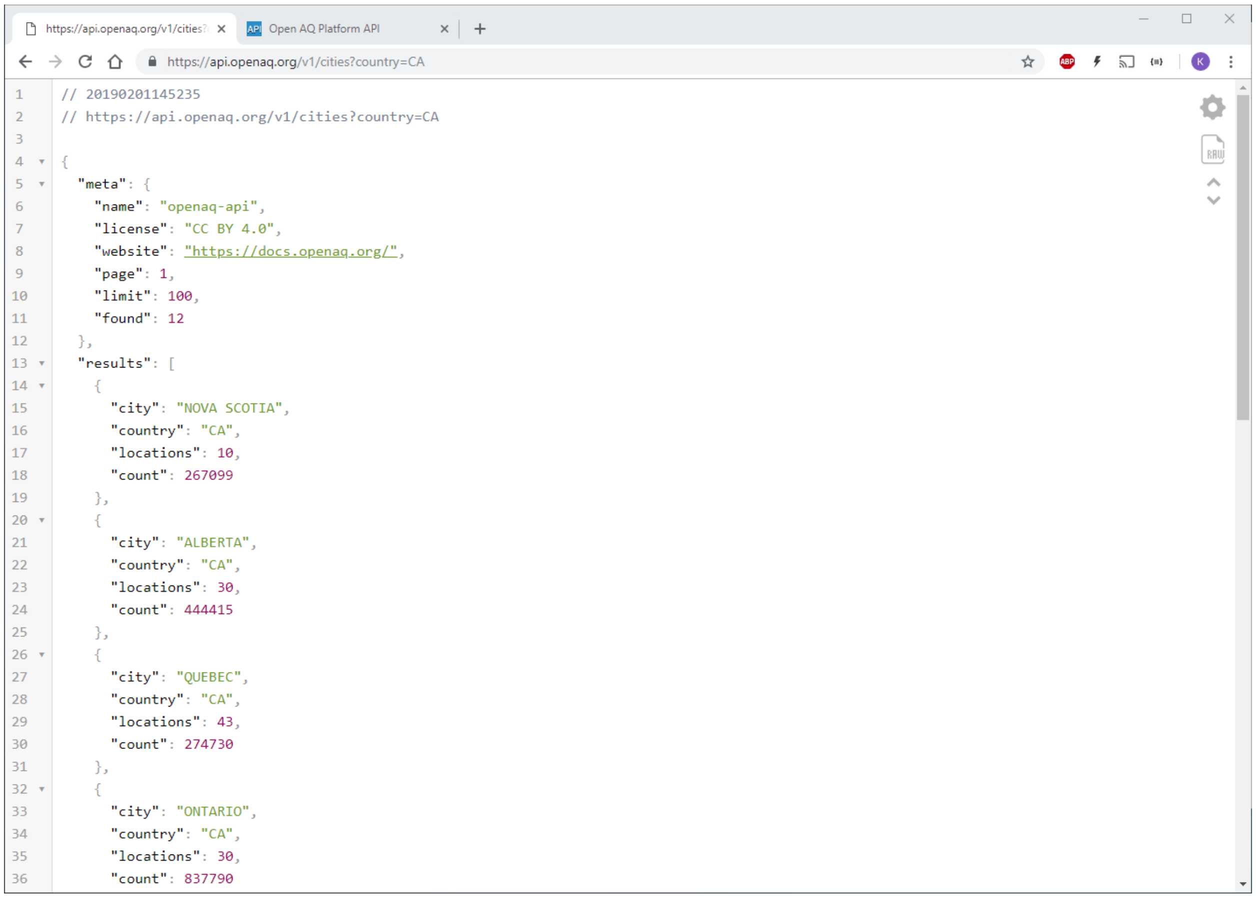
From the API documentation, I learn that I need to specify location information in the form of a country code, a city, and a location in that city. First, what is the country code for Canada? Well, the API doc tells me that https://api.openaq.ort/v1/countries shows basic country info, so I can bring that up in my browser and scroll down until I get to the section on Canada.


https://api.openaq.org/v1/countr
Open AQ Platform API
https://api.openaq.org/v1/countries

```
      "locations": 22,
      "count": 942141
    },
    {
      "name": "Brazil",
      "code": "BR",
      "cities": 72,
      "locations": 119,
      "count": 2812094
    },
    {
      "name": "Canada",
      "code": "CA",
      "cities": 12,
      "locations": 198,
      "count": 3514329
    },
    {
      "name": "Chile",
      "code": "CL",
      "cities": 140,
      "locations": 128,
      "count": 5686966
    },
    {
      "name": "China",
      "code": "CN",
      "cities": 388,
      "locations": 1409,
      "count": 17765312
    },
    {
      "name": "Colombia",
      "code": "CO",
      "cities": 11,
      "locations": 32,
      "count": 127300
```

So Canada has the code CA, with data from 12 cities and 198 locations. Hopefully our target of Vancouver is in that collection.

The documentation also mentions a cities location, so a natural place to look next is https://api.openaq.org/v1/cities. Since cities are embedded in countries, I can add information about the country I'm interested in. For this I use the *country=* parameter following a ? separator along with the CA code for Canada: https://api.openaq.org/v1/cities?country=CA.

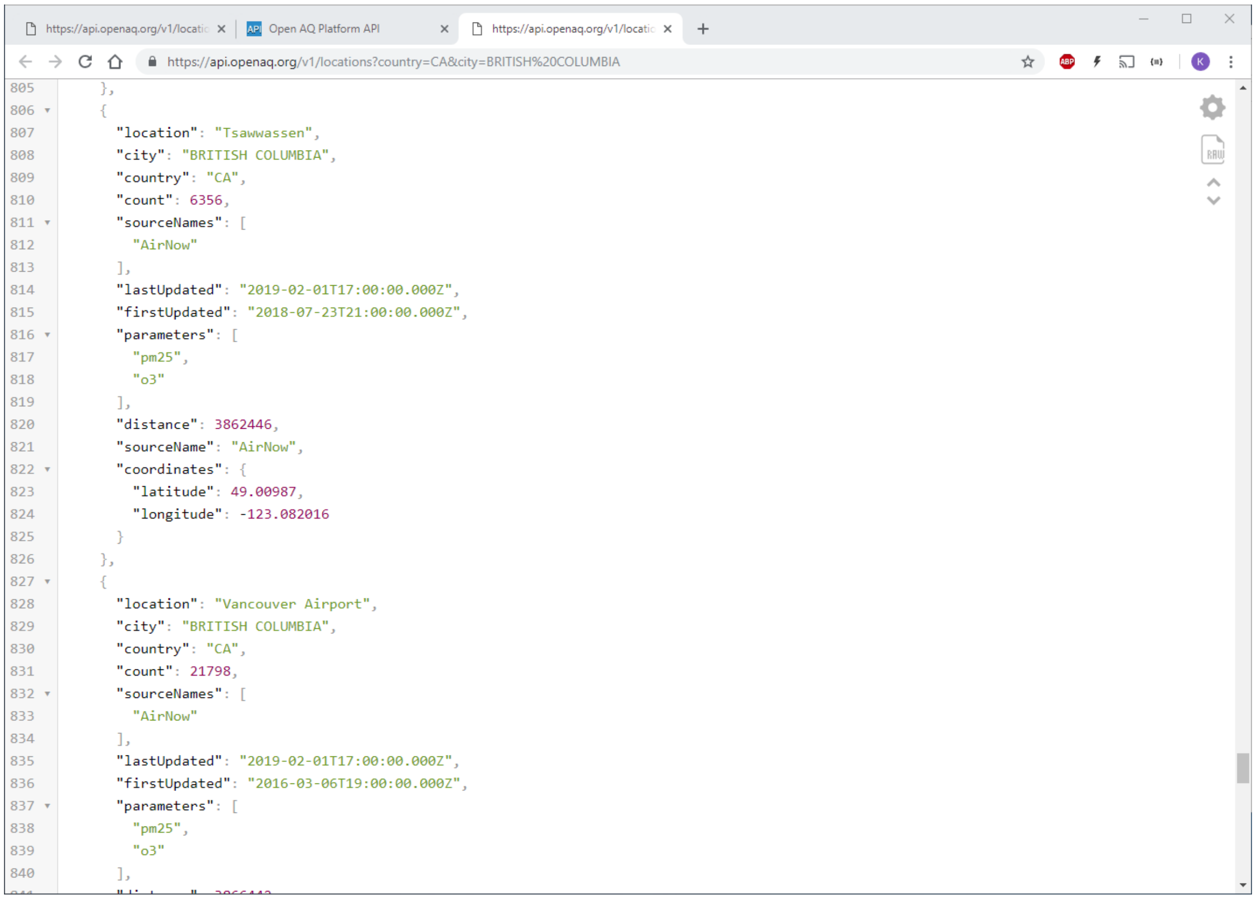


https://api.openaq.org/v1/cities?country=CA

```
// 20190201145235
// https://api.openaq.org/v1/cities?country=CA

{
  "meta": {
    "name": "openaq-api",
    "license": "CC BY 4.0",
    "website": "https://docs.openaq.org/",
    "page": 1,
    "limit": 100,
    "found": 12
  },
  "results": [
    {
      "city": "NOVA SCOTIA",
      "country": "CA",
      "locations": 10,
      "count": 267099
    },
    {
      "city": "ALBERTA",
      "country": "CA",
      "locations": 30,
      "count": 444415
    },
    {
      "city": "QUEBEC",
      "country": "CA",
      "locations": 43,
      "count": 274730
    },
    {
      "city": "ONTARIO",
      "country": "CA",
      "locations": 30,
      "count": 837790
```

Hmm, what this interface is calling cities for Canada is actually Canadian provinces such as Nova Scotia and Quebec. That's OK, I know that Vancouver is in British Columbia, so I can use that value as my "city".

Finally we are ready to use locations data mentioned in the API doc. Is there a location for Vancouver in country=CA, city=British Columbia? I look at https://api.openaq.org/v1/locations?country=CA&city=BRITISH COLUMBIA, which gets translated to https://api.openaq.org/v1/locations?country=CA&city=BRITISH%20COLUMBIA (that is, the blank is filled in with %20). The cities JSON says that British Columbia has 45 locations, and I scroll down until I see a location name that includes Vancouver. It's the Vancouver Airport, and that's fine with me. The information shown below suggests that measurements available at this site are for o3 and pm25 (respectively, ozone and particular matter with a diameter of less than 2.5 micrometers).


https://api.openaq.org/v1/locatio
Open AQ Platform API
https://api.openaq.org/v1/locatio
https://api.openaq.org/v1/locations?country=CA&city=BRITISH%20COLUMBIA

```
    },
    {
      "location": "Tsawwassen",
      "city": "BRITISH COLUMBIA",
      "country": "CA",
      "count": 6356,
      "sourceNames": [
        "AirNow"
      ],
      "lastUpdated": "2019-02-01T17:00:00.000Z",
      "firstUpdated": "2018-07-23T21:00:00.000Z",
      "parameters": [
        "pm25",
        "o3"
      ],
      "distance": 3862446,
      "sourceName": "AirNow",
      "coordinates": {
        "latitude": 49.00987,
        "longitude": -123.082016
      }
    },
    {
      "location": "Vancouver Airport",
      "city": "BRITISH COLUMBIA",
      "country": "CA",
      "count": 21798,
      "sourceNames": [
        "AirNow"
      ],
      "lastUpdated": "2019-02-01T17:00:00.000Z",
      "firstUpdated": "2016-03-06T19:00:00.000Z",
      "parameters": [
        "pm25",
        "o3"
      ],
```

So now finally we should be able to request some measurement data! We have all the information we need to use PROC HTTP to send a request to the API for Vancouver Airport measurements.

```
FILENAME aq1 TEMP;

PROC HTTP

URL="%NRSTR(https://api.openaq.org/v1/measurements?country=CA&city=BRITISH COLUMBIA&location=Vancouver Airport)"
    METHOD="GET"
    OUT=aq1;

RUN;

LIBNAME aq JSON FILEREF=aq1;
```

Looking at the aq library after running this code, we see many data sets: Alldata, Meta, Results, Results_coordinates, and Results_date. We use PROC PRINT to examine each one (or can look at them directly if using SAS Studio). What we need for our graph are measurements and measurement dates, and those seem to be separated into Results and Results_date! Is there a way to easily combine the measurement and measurement date information as we pull the data from the API? Why yes there is, using something called a **JSON map**.

As described in https://documentation.sas.com/?docsetId=lestmtsglobal&docsetTarget=n1jfdetszx99ban1rl4zll6tej7j.htm&docsetVersion=9.4&locale=en, a JSON map is a file that describes the data sets in a library made with a JSON engine. Our library aq is such a library. The cool thing is that we can ask SAS to automatically generate a map, and then we can modify a copy of the map and use our modified map to have SAS construct a data set in the form we want rather than what SAS did by default. Wow! Let's see how to do this.

To make the initial map, we submit a simplified version of the code above and add the MAP=<storage location for map> and AUTOMAP=CREATE options to the LIBNAME statement as the JSON engine is applied.

```
FILENAME aq1 TEMP;

PROC HTTP
     URL="%nrstr(https://api.openaq.org/v1/measurements)"
     METHOD="GET"
     OUT=aq1;
RUN;

LIBNAME in JSON FILEREF=aq1 MAP="C:\my aq data\measurements.user.map"
AUTOMAP=CREATE;
```

You can find this map stored on Sakai, and you can also submit the code yourself to make a copy. The map itself is in the form of a hierarchical list of nested name value pairs. Here's the top part of the map file:

```
{
 "DATASETS": [
  {
   "DSNAME": "meta",
   "TABLEPATH": "/root/meta",
   "VARIABLES": [
    {
     "NAME": "ordinal_root",
     "TYPE": "ORDINAL",
     "PATH": "/root"
    },
    {
     "NAME": "ordinal_meta",
     "TYPE": "ORDINAL",
     "PATH": "/root/meta"
    },
    {
     "NAME": "name",
     "TYPE": "CHARACTER",
     "PATH": "/root/meta/name",
     "CURRENT_LENGTH": 10
    },
    {
     "NAME": "license",
     "TYPE": "CHARACTER",
     "PATH": "/root/meta/license",
     "CURRENT_LENGTH": 9
    },
    {
     "NAME": "website",
     "TYPE": "CHARACTER",
     "PATH": "/root/meta/website",
     "CURRENT_LENGTH": 24
    },
```

Here is our final map (posted as measurements.user_meas.map):

```
{
  "DATASETS": [
    {
      "DSNAME": "pm",
      "TABLEPATH": "/root/results",
      "VARIABLES": [
        {
          "NAME": "ordinal_root",
          "TYPE": "ORDINAL",
          "PATH": "/root"
        },
        {
          "NAME": "ordinal_results",
          "TYPE": "ORDINAL",
          "PATH": "/root/results"
        },
        {
          "NAME": "location",
          "TYPE": "CHARACTER",
          "PATH": "/root/results/location",
          "CURRENT_LENGTH": 29
        },
        {
          "NAME": "parameter",
          "TYPE": "CHARACTER",
          "PATH": "/root/results/parameter",
          "CURRENT_LENGTH": 4
        },
        {
          "NAME": "value",
          "TYPE": "NUMERIC",
          "PATH": "/root/results/value"
        },
        {
          "NAME": "unit",
          "TYPE": "CHARACTER",
          "PATH": "/root/results/unit",
          "CURRENT_LENGTH": 7
        },
        {
          "NAME": "country",
          "TYPE": "CHARACTER",
          "PATH": "/root/results/country",
          "CURRENT_LENGTH": 2
        },
        {
          "NAME": "city",
          "TYPE": "CHARACTER",
          "PATH": "/root/results/city",
          "CURRENT_LENGTH": 11
        },
        {
          "NAME": "ordinal_date",
          "TYPE": "ORDINAL",
          "PATH": "/root/results/date"
        },
        {
          "NAME": "utc",
          "TYPE": "CHARACTER",
          "PATH": "/root/results/date/utc",
          "CURRENT_LENGTH": 24
        },
        {
          "NAME": "local",
          "TYPE": "CHARACTER",
          "PATH": "/root/results/date/local",
          "CURRENT_LENGTH": 25
        }
      ]
    }
  ]
}
```

How did we get from the original map to the one above, and how did we know that the second structure shown above is what we wanted?

My process was as follows (after practicing making a user map using an exercise provided in https://documentation.sas.com/?docsetId=lestmtsglobal&docsetTarget=n1jfdetszx99ban1rl4zll6tej7j.htm&docsetVersion=9.4&locale=en). Looking at the measurements.user.map file in parallel with the five data sets in the aq library, I could see a one-to-one correspondence between sections of the map and the data sets. Four of the five data sets were meta, results, results_coordinates, and results_date, and the map contained four DSNAME sections "meta", "results", "results_date", and "results_coordinates". The VARIABLES lists for each DSNAME in the map file corresponded to the variables in each data set. The idea of a user-constructed map is that you can figure out what variables you want by looking at the data and then in your map define the DSNAME you want, incorporating VARIABLES from various parts of the original map.

Looking at the data sets, I decided that I wanted variables ordinal_root, ordinal_results, location, parameter, value, unit, country, and city from results, and I wanted variables ordinal_date, utc, and local from results_date (this is more than I really needed, but that's OK). I wanted to call my constructed data set meas.

My next step was to make a copy of measurements.user.map. I called my copy measurements.user_meas.map, incorporating the name of the data set I was going to construct. I opened measurements.user_meas.map in my favorite text editor, TextPad. Any text editor will work as long as it shows the structure of the list. Notepad does not, but Notepad ++ does (both are free).

Editing sequence:

- Replace first DSNAME value "meta" with my chosen data set name, "meas"
- Change first TABLEPATH value from "/root/meta" to "/root/results" since that's where most variables will come from.
- We don't want any variables from meta, so delete the entire first "VARIABLES" section. Also delete the subsequent DSNAME and TABLEPATH lines.
- Select the sections for variables "ordinal_date", "utc", and "local" from the DSNAME "results_date" section. Paste those sections after the "city" section in the upper VARIABLES list. Now add a comma after the curly brace (}) that ends the "city" section.
- Now we just need to make sure all of the { } and [ ] are closed appropriately. Select everything below the first] through the next to the last], and delete.
- Everything should look lined up now, and you've defined a single data set MEAS that is to contain 11 variables.

OK, now let's test our map.

```
%LET jsonloc=P:\BIOS 669 2019  logistics\API materials\air quality;

FILENAME aq1 TEMP;

PROC HTTP
```

```
URL="%NRSTR(https://api.openaq.org/v1/measurements?country=CA&city=BRITISH
COLUMBIA&location=Vancouver Airport&limit=10000)"
    METHOD="GET"
    OUT=aq1;
RUN;

FILEMANE testmap "&jsonloc\measurements.user_meas.map";

LIBNAME in JSON FILEREF=aq1 MAP=testmap;

PROC PRINT DATA=in.meas; RUN;
```

Our map works great! Here are the first few records of meas (minus a few variables to fit on the page):

location	*parameter*	*value*	*unit*	*country*	*city*	*ordinal_date*	*local*
Vancouver Airport	o3	0.017	ppm	CA	BRITISH COLUMBIA	1	2019-02-04T08:00:00-08:00
Vancouver Airport	pm25	3.500	Âµg/mÂ³	CA	BRITISH COLUMBIA	2	2019-02-04T08:00:00-08:00
Vancouver Airport	o3	0.017	ppm	CA	BRITISH COLUMBIA	3	2019-02-04T07:00:00-08:00
Vancouver Airport	pm25	2.500	Âµg/mÂ³	CA	BRITISH COLUMBIA	4	2019-02-04T07:00:00-08:00
Vancouver Airport	o3	0.023	ppm	CA	BRITISH COLUMBIA	5	2019-02-04T06:00:00-08:00

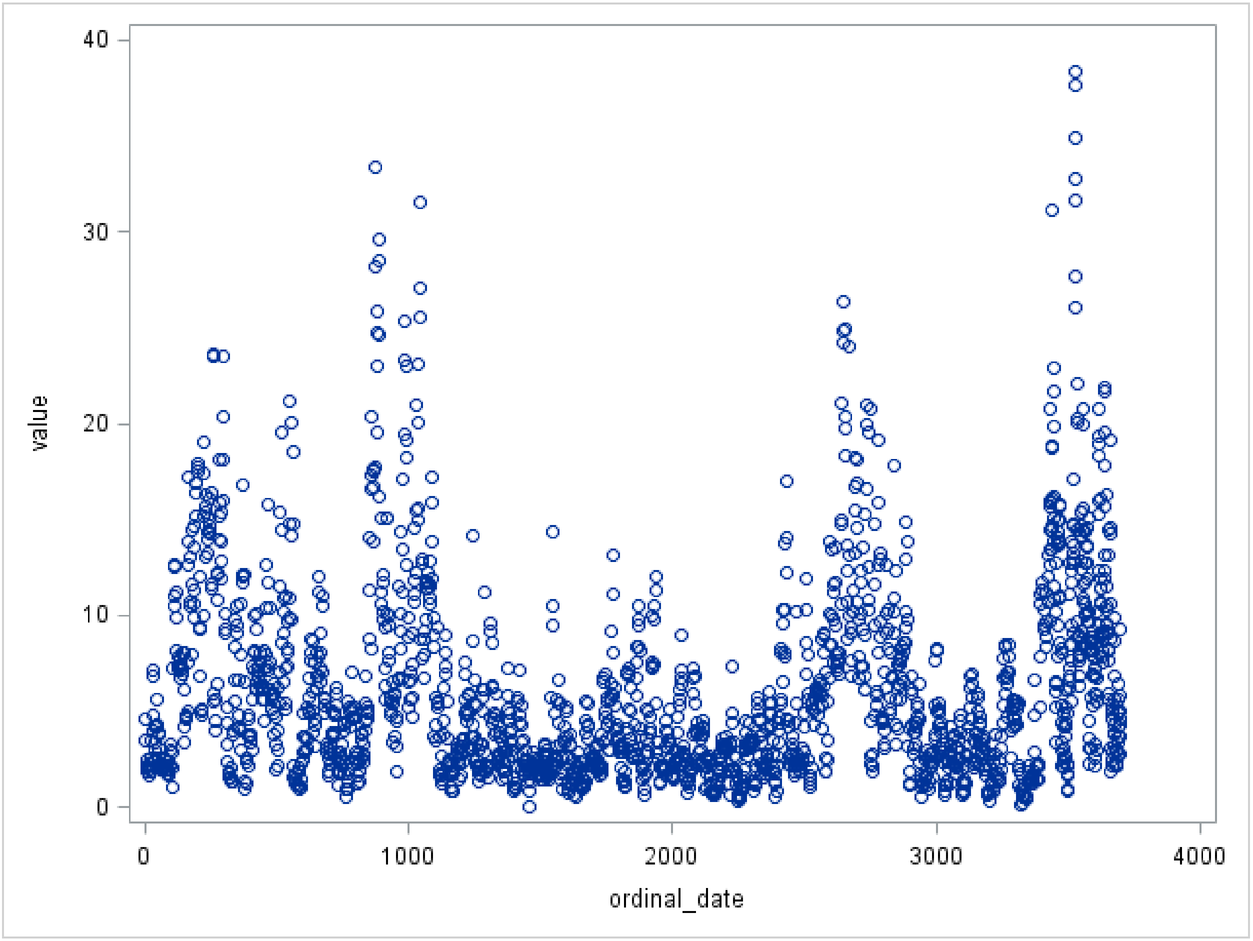
Since we now have measurements, we can plot or otherwise analyze the data. Here's an initial plot of pm25 vs. date*, but it's not clear what the dates are. So before going further, let's prepare the data a bit.

* Interestingly, this plotting code fails with an integer divide by zero message:

```
proc sgplot data=in.meas(where=(parameter='pm25'));
    scatter x=ordinal_date y=value;
run;
```

But this two-step equivalent code succeeds and produced the plot below using data retrieved on 2/4/2019.

```
data pm25;
    set in.meas(where=(parameter='pm25'));
run;

proc sgplot data=pm25;
    scatter x=ordinal_date y=value;
run;
```



Recall that that this site presents hourly data from the last 90 days. Let's plan to show one point per day – particulates at noon – over that period. So how can we select the appropriate records and prepare the date values?

Local dates are presented in this form, as shown in the listing above: 2019-02-04T08:00:00-08:00. The hour is shown right after the T, so we can pull the hour out of this date-time value by taking the two characters starting in place 12. For the subset we want, we will take only records with 12 in that position.

```
    if substr(local,12,2)='12';
```

In terms of the entire date-time string, I looked up SAS informats and found the ymddtt informat that should be able to read dates that look like the ones here. I'll input the current character values with this informat to get a SAS date-time value.

```
    DateTimeAll=input(local,ymddttM15.);
```

All that we really care about for our plot is the date portion of these date-time values, and we can pull that off with the very useful DATEPART function.

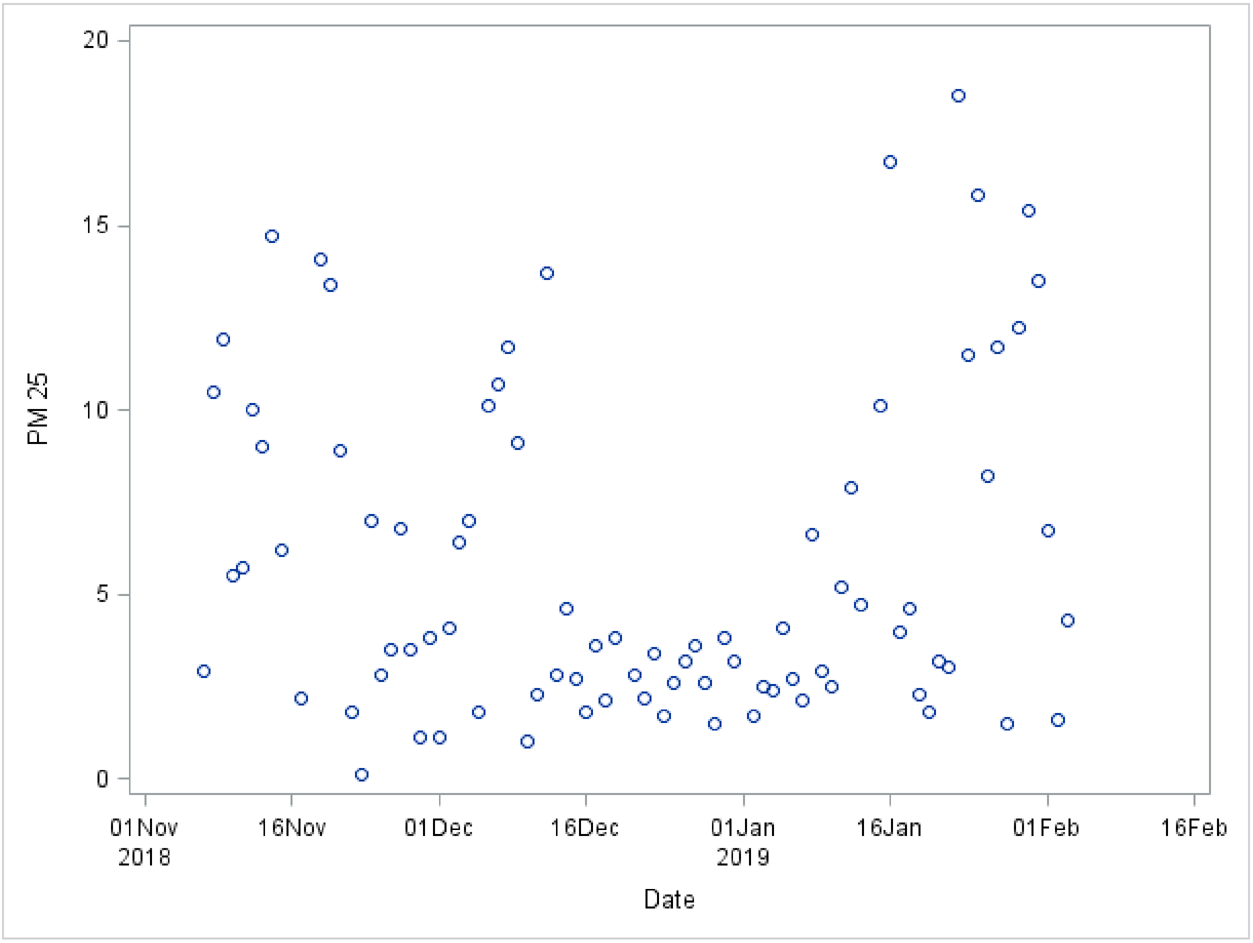
```
    DateOnly=datepart(datetimeall);
```

Finally we will tell SAS to format DateOnly values with the date9. format.

```
    format dateonly date9.;
```

Putting everything together and plotting:

```
data pm25_noon_dates;
    set pm25;
    label value='PM 25'
          DateOnly='Date';
    if substr(local,12,2)='12';
    DateTimeAll=input(local,ymddttM15.);
    DateOnly=datepart(datetimeall);
    format dateonly date9.;
run;

proc sgplot data=pm25_noon_dates;
    scatter x=dateonly y=value;
run;
```



A very cool feature of this plot, I think, is that SAS has automatically put the horizontal axis in the order that we want – earlier to later as we go from left to right – even though the data set itself is in order of later dates at the top to earlier dates at the bottom.

Notice that as stated in the API documentation, this plot does extend back about 90 days from the current day, which as of chapter construction was Feb. 4. To get all 90 days of data, we also needed to specify the value 10000 for the *limit* parameter in our original data request. Only 100 records are returned by default, and the maximum is 10000. In our case, these 10000 records include both o3 records and pm25 records.

Game of Thrones API Example

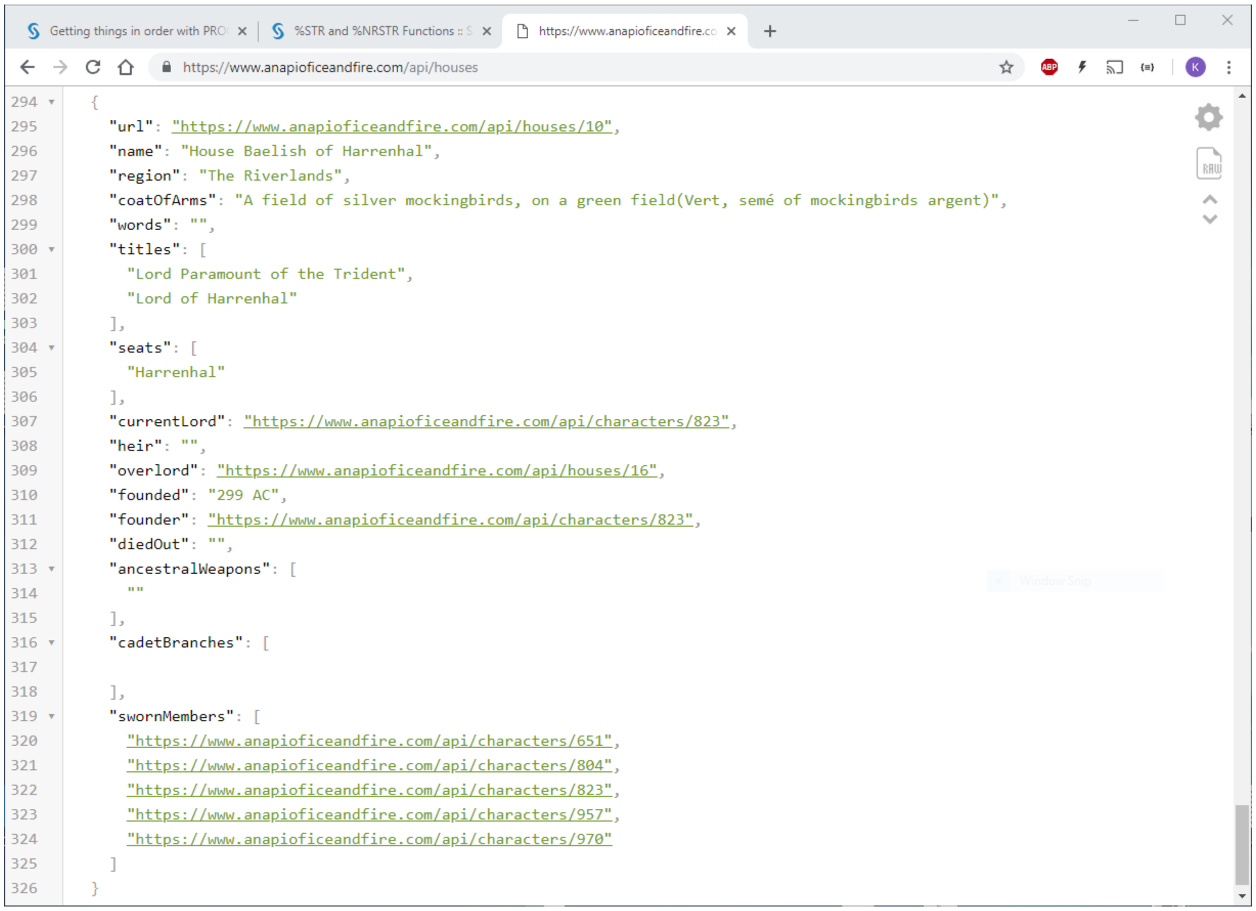
- Using two custom JSON maps and linking data
- A useful URL construction technique

The final season of Game of Thrones will air on HBO starting in April of this year. Many people are excited about this! Some people love Game of Thrones - both the books and the TV series - so much that they maintain an API of Game of Thrones data. The real series by author G.R.R. Martin is called A Song of Ice and Fire, so the website for the API is https://anapioficeandfire.com, with the data made available at https://www.anapioficeandfile.com/api. The first book of the series is called A Game of Thrones, so Game of Thrones became the name attached to the HBO series.

The Ice and Fire books and the Game of Thrones series create an alternative world of powerful houses – hundreds of houses in all, but only 7-12 that really count - that compete for dominance in what's called the Seven Kingdoms. In terms of our current time scale, the basic technology of the series would put Game of Thrones in the Middle Ages. The houses' shifting alliances, varying degrees of morality, and varying house strengths, along with a few strains of the supernatural (such as one house that has dragons), make for an exciting, involving story. Plus the whole of the Seven Kingdoms is threatened by a powerful outside force: an army of zombies called the White Walkers. As we enter the final season (the final book has not yet been written either), will the houses pull together to fight these White Walkers, or will the petty squabbles between houses continue, leading to the fall of all houses to the outside threat?

The ice and fire API is organized into houses, characters, and books. For these notes we will focus on houses and show how you can link to information about the characters in a house. This sequence will require us to make custom user maps for both houses data and characters data. In the end, our sample code will find the name of the current lord for each of six of the most important houses in the Seven Kingdoms: Baratheon, Greyjoy, Lannister, Targaryen, Tully, and Tyrell. Alas, the house with which viewers might most identify, Stark, has no current lord at the time that I compose these notes.

Let's start by taking a look at a portion of the houses data in a browser.



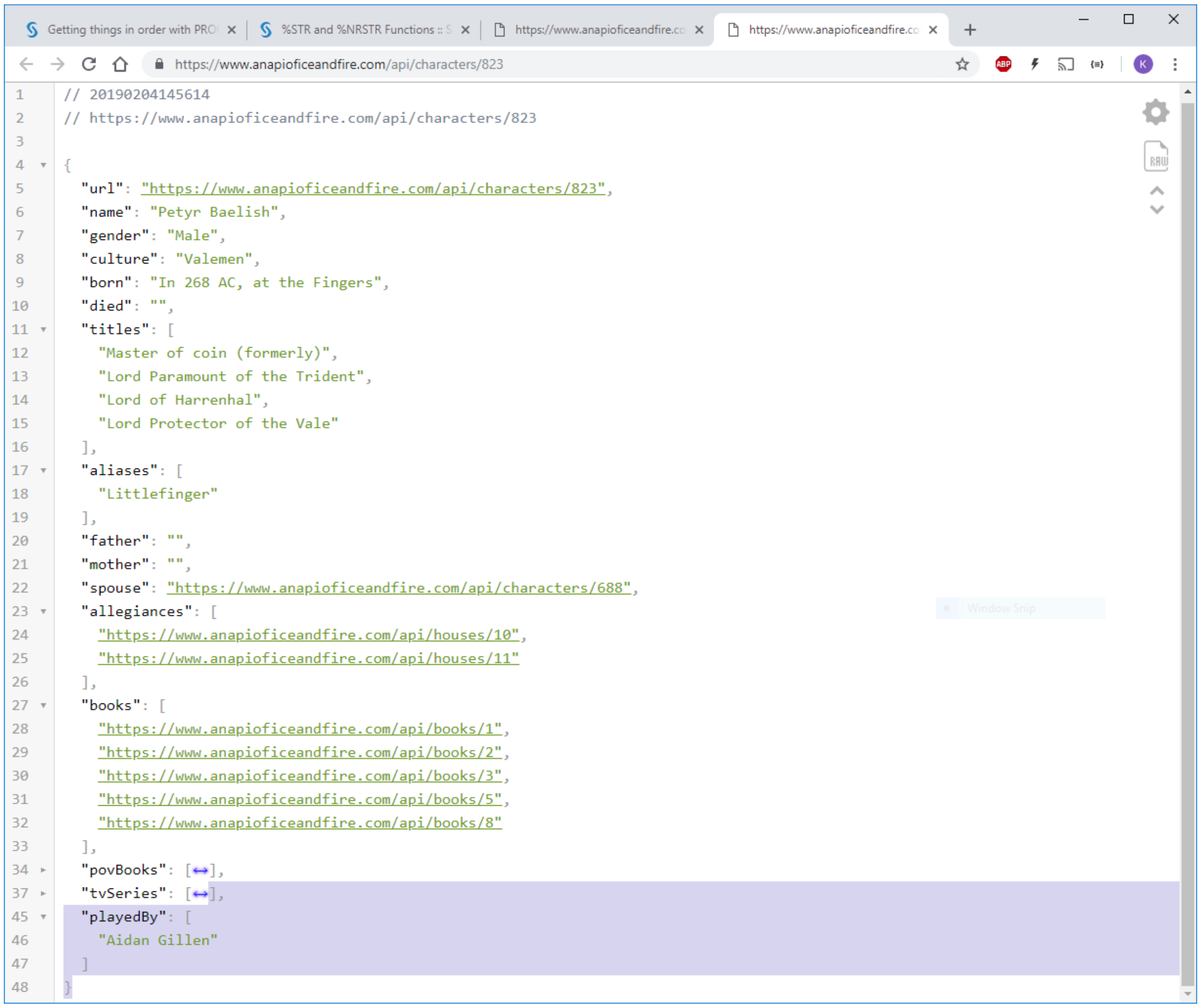
Getting things in order with PRO | %STR and %NRSTR Functions :: S | https://www.anapioficeandfire.co

https://www.anapioficeandfire.com/api/houses

```
  {
    "url": "https://www.anapioficeandfire.com/api/houses/10",
    "name": "House Baelish of Harrenhal",
    "region": "The Riverlands",
    "coatOfArms": "A field of silver mockingbirds, on a green field(Vert, semé of mockingbirds argent)",
    "words": "",
    "titles": [
      "Lord Paramount of the Trident",
      "Lord of Harrenhal"
    ],
    "seats": [
      "Harrenhal"
    ],
    "currentLord": "https://www.anapioficeandfire.com/api/characters/823",
    "heir": "",
    "overlord": "https://www.anapioficeandfire.com/api/houses/16",
    "founded": "299 AC",
    "founder": "https://www.anapioficeandfire.com/api/characters/823",
    "diedOut": "",
    "ancestralWeapons": [
      ""
    ],
    "cadetBranches": [

    ],
    "swornMembers": [
      "https://www.anapioficeandfire.com/api/characters/651",
      "https://www.anapioficeandfire.com/api/characters/804",
      "https://www.anapioficeandfire.com/api/characters/823",
      "https://www.anapioficeandfire.com/api/characters/957",
      "https://www.anapioficeandfire.com/api/characters/970"
    ]
  }
```

This happens to be the data for House Baelish. You can see that many different pieces of data might be available for a house, from a description of the coat of arms to a slogan to the year founded and the founder to a list of sworn members. Also note that a lot of fields are missing. The set of fields is the same from house to house, but most houses have missing data for lots of the fields.

One field that is filled in for House Baelish is the currentLord, with the value https://www.anapioficdandfire.com/api/characters/823. We can also go there in a browser, and we'll see how the characters data is structured.


Getting things in order with PRO
%STR and %NRSTR Functions :: S
https://www.anapioficeandfire.co
https://www.anapioficeandfire.co
https://www.anapioficeandfire.com/api/characters/823

```
// 20190204145614
// https://www.anapioficeandfire.com/api/characters/823

{
  "url": "https://www.anapioficeandfire.com/api/characters/823",
  "name": "Petyr Baelish",
  "gender": "Male",
  "culture": "Valemen",
  "born": "In 268 AC, at the Fingers",
  "died": "",
  "titles": [
    "Master of coin (formerly)",
    "Lord Paramount of the Trident",
    "Lord of Harrenhal",
    "Lord Protector of the Vale"
  ],
  "aliases": [
    "Littlefinger"
  ],
  "father": "",
  "mother": "",
  "spouse": "https://www.anapioficeandfire.com/api/characters/688",
  "allegiances": [
    "https://www.anapioficeandfire.com/api/houses/10",
    "https://www.anapioficeandfire.com/api/houses/11"
  ],
  "books": [
    "https://www.anapioficeandfire.com/api/books/1",
    "https://www.anapioficeandfire.com/api/books/2",
    "https://www.anapioficeandfire.com/api/books/3",
    "https://www.anapioficeandfire.com/api/books/5",
    "https://www.anapioficeandfire.com/api/books/8"
  ],
  "povBooks": [↔],
  "tvSeries": [↔],
  "playedBy": [
    "Aidan Gillen"
  ]
}
```

We see that the name of character 823 is Petyr Baelish.

What we would like to do in our example for these notes is to programmatically retrieve the key pieces of houses data for our six houses of interest, including programmatically reading the characters data for the current lord of each house to get the name of that current lord.

As we did with the air quality data, the first step in making our custom houses data map is to ask SAS to generate a starter map. I also found it useful to print the initial houses data sets to an RTF file to help me decide what variables I wanted to keep in my custom houses map. PROC DATASETS was a help in learning what those data sets were named.

```
%let jsonloc=P:\BIOS 669 2019  logistics\API materials\game of thrones
exercise;

filename hous1 temp;

proc http
    url="%nrstr(https://www.anapioficeandfire.com/api/houses)"
    method="GET"
    out=hous1;
run;


libname in json fileref=hous1 map="&jsonloc\houses.user.map" automap=create;


proc datasets lib=in; run; quit;



ods rtf file="&jsonloc.\houses_data.rtf" style=journal;
proc print data=in.root; title 'root'; run;
proc print data=in.alldata; title 'alldata'; run;
proc print data=in.ancestralweapons; title 'ancestralweapons'; run;
proc print data=in.cadetbranches; title 'cadetbranches'; run;
proc print data=in.seats; title 'seats'; run;
proc print data=in.swornmembers; title 'swornmembers'; run;
proc print data=in.titles; title 'titles'; run;
ods rtf close;
```

In the end, I decided to keep only a subset of variables from the first grouping (original DSNAME “root”), and I changed the DSNAME value to “houses”. So here is my custom houses map (also posted on Sakai with the name user.map.housetest – see this unit’s aq_map video to see how to construct a map):

```
{
 "DATASETS": [
  {
   "DSNAME": "houses",
   "TABLEPATH": "/root",
   "VARIABLES": [
    {
     "NAME": "url",
     "TYPE": "CHARACTER",
     "PATH": "/root/url",
     "CURRENT_LENGTH": 47
    },
    {
     "NAME": "name",
     "TYPE": "CHARACTER",
     "PATH": "/root/name",
     "CURRENT_LENGTH": 27
    },
    {
     "NAME": "region",
     "TYPE": "CHARACTER",
     "PATH": "/root/region",
     "CURRENT_LENGTH": 15
    },
    {
     "NAME": "words",
     "TYPE": "CHARACTER",
     "PATH": "/root/words",
     "CURRENT_LENGTH": 21
    },
    {
     "NAME": "currentLord",
     "TYPE": "CHARACTER",
     "PATH": "/root/currentLord",
     "CURRENT_LENGTH": 52
    },
    {
     "NAME": "founder",
     "TYPE": "CHARACTER",
     "PATH": "/root/founder",
     "CURRENT_LENGTH": 52
    }
   ]
  }
 ]
}
```

To check whether this map works, I'll reference it on my LIBNAME statement as I retrieve some data from the houses API. I'll also take this opportunity to test out the API's page= and pageSize= parameters.

```
%let jsonloc=P:\BIOS 669 2019  logistics\API materials\game of thrones
exercise;

filename hous1 temp;

proc http

url="%nrstr(https://www.anapioficeandfire.com/api/houses?page=8&pageSize=50)"
    method="GET"
    out=hous1;
run;


filename testmap "&jsonloc\user.map.housetest";
```

```
libname in json fileref=hous1 map=testmap;

options nodate nocenter;
ods rtf bodytitle style=journal file="&jsonloc\houses50.rtf";
proc print data=in.houses(obs=5); run;
ods rtf close;
```

It works! Here are the five printed records (words field cut off on right). The seven variables requested in my custom house map are the ones that are included. Note that since I requested the 8th page with a pageSize of 50, I did get the 8th set of 50 houses assuming the houses are sequentially numbered starting at 1, which they are. So if these are the first 5 records of what I retrieved, it makes sense that we are seeing information for houses 351 – 355.

Obs	*url*	*name*	*region*	*words*
1	https://www.anapioficeandfire.com/api/houses/351	House Shett of Gulltown	The Vale	
2	https://www.anapioficeandfire.com/api/houses/352	House Slate of Blackpool	The North	
3	https://www.anapioficeandfire.com/api/houses/353	House Sloane	The Reach	
4	https://www.anapioficeandfire.com/api/houses/354	House Slynt of Harrenhal	None	
5	https://www.anapioficeandfire.com/api/houses/355	House Smallwood of Acorn Hall	The Riverlands	From These Beg

Obs	*currentLord*	*founder*
1		
2		
3		
4	https://www.anapioficeandfire.com/api/characters/761	https://www.anapioficeandfire.com/api/characters/533
5	https://www.anapioficeandfire.com/api/characters/1020	

OK, so we are in great shape as far as reading house data is concerned. What about character data?

I made a custom character map through the same process I followed for the custom house map, although for the character map I took variable definitions from several of the original groupings (url, name, gender, and culture from root; playedBy1 from playedBy; and aliases1 from aliases). Here's the custom character map, which I named user.map.charactertest (also available on Sakai):

```
{
 "DATASETS": [
  {
   "DSNAME": "characters",
   "TABLEPATH": "/root",
    "VARIABLES": [
     {
       "NAME": "url",
       "TYPE": "CHARACTER",
       "PATH": "/root/url",
       "CURRENT_LENGTH": 51
     },
     {
       "NAME": "name",
       "TYPE": "CHARACTER",
       "PATH": "/root/name",
       "CURRENT_LENGTH": 6
     },
     {
       "NAME": "gender",
       "TYPE": "CHARACTER",
       "PATH": "/root/gender",
       "CURRENT_LENGTH": 6
     },
     {
       "NAME": "culture",
       "TYPE": "CHARACTER",
       "PATH": "/root/culture",
       "CURRENT_LENGTH": 8
     },
     {
       "NAME": "playedBy1",
       "TYPE": "CHARACTER",
       "PATH": "/root/playedBy/playedBy1",
       "CURRENT_LENGTH": 14
     },
     {
       "NAME": "aliases1",
       "TYPE": "CHARACTER",
       "PATH": "/root/aliases/aliases1",
       "CURRENT_LENGTH": 24
     }
    ]
  }
 ]
}
```

Notice that it makes a data set named characters.

Let’s see if this map works to retrieve data for our character 823, Petyr Baelish. Notice that I want a single record so I use a pageSize of 1, and I ask specifically for page=823.

```
%let jsonloc=P:\BIOS 669 2019  logistics\API materials\game of thrones 
exercise;

filename char1 temp;

proc http

url="%nrstr(https://www.anapioficeandfire.com/api/characters?page=823&pageSiz
e=1)"
    method="GET"
    out=char1;
run;


filename testmap "&jsonloc\user.map.charactertest";


libname in json fileref=char1 map=testmap;

proc print data=in.characters label; run;
```

It works! Here is what gets printed:

```
Obs  url                                              name           gender  culture   playedBy1     aliases1
  1  https://www.anapioficeandfire.com/api/characters/823  Petyr Baelish  Male    Valemen   Aidan Gillen  Littlefinger
```

As the next step in our attempt to link the current lord name to a specific house for a set of houses of interest, first let's figure out how to do this linking for a single house. If we look at the houses API data in a browser, we see that house 1 does not have a current lord, but house 2 does (happens to be character 298). Let's see if we can write code to read data for house 2, detect that the currentLord number is 298, read data for character 298, and finally join the house and character data so that our house 2 data includes the name of the current lord. Note: looking at the characters API for character 298 in a browser shows the character's name to be Delonne Allyrion, so that's the lord name we will hope to see in our data set in the end.

But first, wouldn't it be useful to have a macro that we could call to get character info of interest, if supplied with a particular character number? Here is such a macro:

```
/*

   macro for retrieving a LordName based on a lord number
   a helpful post in writing and calling this macro:
   https://blogs.sas.com/content/sgf/2017/08/02/call-execute-for-sas-data-
driven-programming/

*/

%macro getchar(charnum= );

    filename chars temp;

    %let url=%nrstr(https://www.anapioficeandfire.com/api/characters/)&charnum.;
    %put url=&url ;

    proc http
        url="&url"
        method="GET"
        out=chars;
    run;

    filename testmap "&jsonloc\user.map.charactertest";


    libname in json fileref=chars map=testmap;

    proc print data=in.characters label; run;

    /* save this bit of data with a name incorporating the character number */
    data char&charnum;
        length LordName $30 gender $10 culture $20 playedBy1 $30 aliases1 $50;
        set in.characters(rename=(name=LordName));
    run;

%mend;
```

Possibly the most interesting part of this macro is construction of the &url variable to send to the URL option of PROC HTTP. Making it a macro variable in the first place allows us to avoid using quotation marks until we are calling PROC HTTP. In constructing &url, notice that we call %NRSTR to allow us to use possibly troublesome characters without SAS misinterpreting things as macro objects, and after the %NRSTR call we insert a reference to our macro parameter &charnum, which we DO want SAS to interpret as a macro object.

Here is some pseudo-code for our goal of adding the name of the current lord for house 2 to the other house 2 data:

```
/* retrieve basic house 2 data */
/* find current lord number within that house 2 data */

/* send that lord number to our macro that retrieves data for a specified
character number */

/* combine the newly-retrieved character data with previous house 2 data */
```

Now that we have broken our task into steps, let's fill in the pseudo-code sections.

```
/* retrieve basic house 2 data */

%let jsonloc=P:\BIOS 669 2019  logistics\API materials\game of thrones
exercise;

filename hous1 temp;

proc http
    url="%nrstr(https://www.anapioficeandfire.com/api/houses/2)"
    method="GET"
    out=hous1;
run;

filename testmap "&jsonloc\user.map.housetest";

libname in json fileref=hous1 map=testmap;

title 'house 2';
proc report data=in.houses nowindows; ... run;
title;
```

OK, there was nothing new in that step.

house 2

url	*name*	*region*	*words*	*currentLord*
https://www.anapioficeandfire.com/api/houses/2	House Allyrion of Godsgrace	Dorne	No Foe May Pass	https://www.anapioficeandfire.com/api/characters/298

```
/* find current lord number within that house 2 data */
```

```
data lordnum;
    set in.houses;

    LordNum = scan(currentLord,-1,'/');

    call symput("LNum",LordNum);
run;
%put LNum=&LNum;

title 'check lord number of house 2 - pulled off correctly?';
proc print data=lordnum noobs; var url currentLord LordNum; run;
title;
```

check lord number of house 2 - pulled off correctly?

url	*currentLord*	*LordNum*
https://www.anapioficeandfire.com/api/houses/2	https://www.anapioficeandfire.com/api/characters/298	298

Did you know that you could call the SCAN function with a negative number in order to pull off words starting from the right rather than the left? That is so useful in this case, when the current lord character number appears as the number being sent to the characters API and therefore is in the final position in a URL such as https://www.anapioficeandfire.com/api/characters/123. If the SCAN function starts at the right end of this string and pulls off the first word using / as the delimiter, the word found here would be 123 (that's right, the string itself is NOT reversed).

Then we write the current lord character number to a macro variable for our later use.

```
/* send that lord number to our macro that retrieves data for a specified
character number */

%getchar(charnum=&LNum)
```

This step is easy since we defined the %getchar macro earlier. Recall that the macro results in a data set named char&charnum (or in this case, char&LNum, so char298).

```
/* combine the newly-retrieved character data with previous house 2 data */


proc sql;
    create table addlord as
        select *
            from lordnum(rename=(url=houseURL)) as h
                left join
```

```
                 char&LNum as c
            on h.currentLord = c.url;
quit;

proc report data=addlord; ... run;
```

Current lord info added to house info

houseURL	*name*	*region*	*words*	*currentLord*	*LordName*	*LordNu*
https://www.anapioficeandfire.com/api/houses/2	House Allyrion of Godsgrace	Dorne	No Foe May Pass	https://www.anapioficeandfire.com/api/characters/298	Delonne Allyrion	298

gender	*playedBy1*	*url*
Female		https://www.anapioficeandfire.com/api/characters/298

This is a great place to use SQL to combine the data since our merging variables have different names. Recall that the URL for the current lord was in the currentLord variable in the original house 2 data, and in the data returned from the character API, that URL is in the url variable. Because I want to keep both the house and character URLs in my final data set, I'll rename the url variable coming from the house data.

Of course in this case both data sets only had a single observation, but we could be doing this operation for a series of houses and then we would need to be careful to match up houses and current lord characters correctly.

OK, as our final example, we want to be able to call the code above for any house name and number and find out the name etc. of the current lord. We'll do it for these six main house names/numbers (OK, I kind of cheated to find out the house numbers – looked at the houses API in batches of 50 looking for familiar house names):

Baratheion	15
Greyjoy	169
Lannister	229
Targaryen	378
Tully	295
Tyrell	397

Assuming that the getchar macro has already been defined, this code should do it:

```
%macro houselord(housename= , housenumber= );

    filename thrones temp;

    * specify house number directly;
    proc http
url="%nrstr(https://www.anapioficeandfire.com/api/houses/)&housenumber."
        method="GET"
        out=thrones;
    run;

    filename testmap "&jsonloc.\user.map.housetest";

    libname in json fileref=thrones map=testmap;

    /* proc print data=in.houses; run; */

    data keep;
        length url $100 name $60 region $20 words $100;
        set in.houses(where=(^missing(currentLord)) drop=founder);

        length FamilyName $20;

        FamilyName="&housename";

        LordNum = scan(currentLord,-1,'/');

        call symput("LNum",LordNum);
    run;

    %put &LNum;

    /* proc print data=keep; title 'has currentLord value'; run; title; */

    %getchar(charnum=&LNum)


    /* now add name of lord to the house */
    /* url is merge variable! use SQL to combine to avoid having to sort or
rename variables */
```

```
    proc sql;
        create table addlord&housenumber as
            select *
                from keep(rename=(url=houseURL))
                    left join
                     char&LNum
                on keep.currentLord = char&LNum..url;
    quit;

    title "&housename addlord&housenumber";
    proc print data=addlord&housenumber;
        var FamilyName LordName houseURL currentLord
    run;
    title;

%mend;

%houselord(housename=Baratheon, housenumber=15)
%houselord(housename=Greyjoy,   housenumber=169)
%houselord(housename=Lannister, housenumber=229)
%houselord(housename=Targaryen, housenumber=378)
%houselord(housename=Tully,     housenumber=395)
%houselord(housename=Tyrell,    housenumber=397)
/* no currentLord %houselord(housename=Stark,     housenumber=362) */

data stack;
    set addlord15
        addlord169
        addlord229
        addlord378
        addlord395
        addlord397;
run;

proc report data=stack;
    columns FamilyName LordName houseURL currentLord;
    define FamilyName  / display style=[cellwidth=3 cm];
    define LordName    / display style=[cellwidth=3 cm];
    define houseURL    / display style=[cellwidth=5 cm];
    define currentLord / display style=[cellwidth=5 cm];
run;
```

All six houses with their current lords

FamilyName	*LordName*	*houseURL*	*currentLord*
Baratheon	Stannis Baratheon	https://www.anapioficeandfire.com/api/houses/15	https://www.anapioficeandfire.com/api/characters/1963
Greyjoy	Euron Greyjoy	https://www.anapioficeandfire.com/api/houses/169	https://www.anapioficeandfire.com/api/characters/385
Lannister	Cersei Lannister	https://www.anapioficeandfire.com/api/houses/229	https://www.anapioficeandfire.com/api/characters/238
Targaryen	Daenerys Targaryen	https://www.anapioficeandfire.com/api/houses/378	https://www.anapioficeandfire.com/api/characters/1303
Tully	Edmure Tully	https://www.anapioficeandfire.com/api/houses/395	https://www.anapioficeandfire.com/api/characters/346
Tyrell	Garlan Tyrell	https://www.anapioficeandfire.com/api/houses/397	https://www.anapioficeandfire.com/api/characters/401

Note: I also plan to post code that uses a cool SAS technique, CALL EXECUTE, to process a series of sequential house numbers rather than calling the houselord macro for each separate house.